Formal Verification of Multimodal Dialogs in Pervasive Environments

Formale Verifikation von Multimodalen Dialogen in Pervasiven Umgebungen
Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation von M.Sc. Stefan Radomski aus Hannover
Tag der Einreichung: 2. Oktober, 2015, Tag der Prüfung: 24. November, 2015
Darmstadt, 2015 — D 17

1. Gutachten: Prof. Dr. Max Mühlhäuser
2. Gutachten: Prof. Dr. Gertrude Kappel

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Telekooperation Group

Formal Verification of Multimodal Dialogs in Pervasive Environments
Formale Verifikation von Multimodalen Dialogen in Pervasiven Umgebungen

Genehmigte Dissertation von M.Sc. Stefan Radomski aus Hannover

1. Gutachten: Prof. Dr. Max Mühlhäuser
2. Gutachten: Prof. Dr. Gertrude Kappel

Tag der Einreichung: 2. Oktober, 2015
Tag der Prüfung: 24. November, 2015

Darmstadt, 2015 — D 17

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den December 9, 2015

(Stefan Radomski)

Vorwort

Diese Dissertation entstand während meiner Arbeit als wissenschaftlicher Mitarbeiter an der Technischen Universität Darmstadt und repräsentiert eine wichtige Zäsur in meiner beruflichen wie auch persönlichen Entwicklung. Als Erster in meiner Familie mit höherer Bildung im eigentlichen Sinne möchte ich kurz innehalten und bitte den Pathos zu verzeihen, wenn ich von dem unwahrscheinlichen Glück berichten möchte, welches mich bis zum heutigen Tag getragen hat und den Menschen, die all dies ermöglicht haben.

Sicherlich will ich meiner Familie danken: Meiner Mutter, die mir stets den hervorgehobenen Wert von Bildung vermittelt hat und für die es immer selbstverständlich war, dass ihr Sohn studieren wird. Meinem Vater, der mir jederzeit die Freiheiten eingeräumt hat in denen sich mein Charakter und meine Selbstsicherheit haben bilden können. Meinen Grosseltern für ihre Unterstützung und ihre Lebenserfahrung die sie, oft vergeblich, versuchten an mich weiterzugeben. Meiner langjähigen Lebensgefährtin, die mir jederzeit liebevoll zur Seite stand und die für unsere gemeinsame Zukunft vielleicht auch mehr in ihrer Lebensplanung angepasst hat als ich das in meiner getan habe.

Aber auch ausserhalb meines familiären Umfeldes gibt es Freunde, die wesentlichen Einfluss darauf hatten, dass ich diese Zeilen als Vorwort meiner Dissertation schreiben kann und denen ich danken möchte: Zum einen Oliver, Mohamed und Peter, mit denen ich jahrelang gemeinsam musiziert habe und die mich in den prägenden Jahren meiner Jugend maßgeblich beeinflusst haben. Zum anderen sicherlich Leif, der mich all die Jahre seit meinem Abitur bis zum erfolgreichen Abschluss meines Studiums begleitet hat und der mich stets motiviert hat: Ohne unser gemeinsames Lernen hätte ich wohl nichtmal das Studium abgeschlossen. Ferner auch Fritz, Darius und André mit denen ich immer wieder zusammen gewohnt und ebenso lange Zeit gemeinsam musiziert habe.

Abschliessend will ich natürlich auch meinen Kollegen und Koautoren danken, für all die kritischen und erhellenden Diskussionen die meine Arbeit hier an der “TK” begleitet haben. Besonders hervorzuheben ist sicherlich Dirk, der mein Interesse an der Forschung rund um multimodale Interaktion in ubiquitären Umgebungen geweckt und unzählige Arbeiten, wissenschaftliche wie auch projektbezogene mit mir durchgeführt hat. Seine Anregungen und Vorarbeiten waren es, welche mich letztlich zum Thema der vorliegenden Dissertation geführt haben. Sowie Max für seine fachlichen Anregungen, aber auch vor allem für seine Geduld mit der vorliegenden Dissertation: So unvorbereitet wie ich mich nach dem Abitur im Studium gefühlt habe, so unvorbereitet fühlte ich mich nach dem Studium als wissenschaftlicher Mitarbeiter an der TK. Von ein paar Rahmenbedingungen durch externe Projekte und der generellen Ausrichtung der TK abgesehen ist das Arbeiten in dieser Gruppe vor allem durch Eigeninteresse und individuellen Gestaltungswillen geprägt. Entsprechend divers sind die Themen, welche sich Doktoranden als promotionswürdig erarbeiten. Hier fällt die vorliegende Dissertation ein wenig aus dem Rahmen bisheriger Beiträge der TK in Bezug auf Interaktion. Wo viele vorangegangene Arbeiten ihre Beiträge rund um neue Interaktionsformen mit Benutzerstudien belegen ist diese Arbeit formalerer Natur und die Ergebnisse mit einer Mischung aus Simulationen und Beweisen belegt.

Alles toll - Danke!

Abstract

Providing reliable and coherent interfaces to end-users in pervasive environments with a wealth of connected sensors and actuators is still an area with little to no support in terms of common methodologies and established standards. The wide range of diverse situations, in which interaction in these environments potentially takes place, prevents any single means of interaction to form a predominant approach. While one class of interaction devices, might be well suited in one given situation, it might be wholly inapplicable in another. Yet, users rightfully expect an interaction session to continue coherently when their situation calls for a change with regard to their means of interaction.

The entailing classes of new requirements for user interfaces, such as distributed interaction logic, guaranteed coherence with regard to the state of the overall interaction and the necessity to effectively support multiple modalities as equals are addressed only insufficiently by the established approaches to model user interface that are popular today. Much research has already been conducted in the last 50 years to develop various conceptualizations applicable to model distributed, coherent, and / or multimodal interfaces, but so far this only lead to a plethora of isolated research prototypes and solutions to specific problems in the design space, with no solution having developed the inertia to succeed in establishing a permanent foothold in industry-supported, commercial end-user applications. While first applications are, indeed, starting to be commercialized, e.g. as seen on IFA 2015 and even earlier fairs, their lack of a standardized approach is still a serious obstacle to commodification: Having a BMW tell a Buderus central heating system that a user expects a warm living room in 30 minutes is appreciated, but the overall usefulness is diminished if one needs to change into the VW only to check on the contents of a Miele fridge, losing all interaction context in between.

In this thesis, we will elaborate on an approach, recently recommended by the "W3C Multimodal Interaction Working Group", to express the interaction logic of user interfaces in pervasive environments as modality-agnostic, nested state-charts controlling modality-specific components. The major contribution is a description of an automated transformation from these state-charts, given in an XML dialect onto the input language of a model-checker. This allows an interface designer to formally guarantee certain behaviors of the state-charts and thereby properties of the interaction described within. This core contribution is evaluated by (i) identifying the subset of state-chart semantics applicable for formal verification, (ii) approaching a proof of correctness of the transformation via the official tests accompanying the state-chart description language and by (iii) showing applicability of the overall approach via transforming a non-trivial state-chart employed to describe the user interface of a commercial consumer product.

Several smaller contributions of this thesis include (i) a proof of the Turing completeness of the employed state-chart semantics as well as the embedding of a push-down automaton, (ii) a transformation onto semantically equivalent state-machines in a slightly extended variant of the state-chart description language with (iii) a closed-form upper bound for the number of states in the state-machine (iv) and extensions to improve the applicability and expressiveness of the proposed state-chart formalism with regard to other established dialog modeling techniques.

All of this is actually implemented in a platform-independent C++ state-chart interpreter, compliant to the respective W3C standards and accompanied by various tools available under a free open-source license. The interpreter is already used in commercial deployments of multimodal dialog systems and was submitted as part of the implementation and report plan for the respective W3C recommendation.

Zusammenfassung

Das Entwickeln und Bereitstellen von verlässlichen und kohärenten Benutzerschnittstellen in pervasiven Umgebungen welche durchdrungen sind mit einer Vielzahl von vernetzten Sensoren und Aktuatoren durch Anwendungsentwickler ist noch immer unzureichend unterstützt, vor allem in Hinsicht auf eine gemeinsame Methodologie und etablierte Standards. Das breite Spektrum von Situationen in denen ein Benutzer in einer solchen Umgebung potentiell interagieren möchte bedingt, dass keine einzelne Interaktionsform jederzeit geeignet oder auch nur verfügbar ist. Nichtsdestotrotz erwarten Anwender zurecht, dass ihr Interaktionskontext unabhängig von der konkreten Wahl der Interaktionform bestehen bleibt.

Entsprechend lassen sich mehrere Anforderungen für eine Interaktionsbeschreibung in solchen Umgebungen ableiten, so zum Beispiel die Notwendigkeit zur Verteilung der Beschreibung, garantierte Kohärenz in Bezug auf den Interaktionskontext und die Notwendigkeit mehrere Interaktionsformen als gleichwertig zu behandeln. In den vergangenen 50 Jahren wurde eine Vielzahl von Ansätzen wissenschaftlich beschrieben, welche den Anspruch haben, entweder vereinzelte relevante Probleme zu lösen oder generell Interaktion in solchen Umgebungen in Form isolierter Forschungsprototypen umzusetzen. Bis zum heutigen Tag gibt es allerdings keinen umfassenden Ansatz, welcher eine breite Unterstützung seitens der Industrie genießt und die einzelnen Lösungen in einem kohärenten Rahmenwerk interoperabel zusammenführt. Wenngleich erste kommerzielle Produkte bereits z.B. auf der IFA 2015 und auch früheren Messen vorgestellt wurden, so bleiben diese bislang doch Insellösungen, auf Produkte der Hersteller in einzelnen Konsortien begrenzt: So hilfreich es beispielsweise auch sein mag, wenn der BMW eines Anwenders der Buderus Zentralheizung daheim meldet, dass in 30 Minuten ein warmes Wohnzimmer erwarte wird, so ärgerlich ist es doch, wenn man in den VW des Partners wechseln muss, um zuvor noch den Inhalt des Miele Kühlschrankes abzufragen und dabei den gesamten Interaktionskontext verliert.

In der vorliegenden Dissertation werden wir im Detail den kürzlich standardisierten Ansatz des W3C für Interaktionbeschreibungen in pervasiven Umgebungen betrachten, worin modalitätsagnostische Zustandsübergangsdiagramme nach Harel modalitätsspezifische Komponenten kontrollieren, um im Zusammenspiel die Benutzerinteraktion formal zu beschreiben. Der wesentliche Beitrag dieser Dissertation ist hierbei eine automatische Transformation dieser Beschreibungen auf die Eingabesprache eines Werkzeuges für die Modellprüfung, um temporallogische Aussagen und Einschränkungen über eben diesen Interaktionbeschreibungen verifizieren zu können. Dieser Kernbeitrag wird hierbei wie folgt evaluiert: (i) Die Teilmenge der Semantik der Interaktionsbeschreibungssprache, welche der formalen Verifikation zugänglich ist wird identifiziert, (ii) ein Beweis für die Korrektheit der Transformation wird durch die offiziellen Tests bezüglich ihrer funktionalen Anforderungen angenähert, (iii) die Anwendbarkeit des Gesamtansatzes wird am Beispiel einer nicht-trivialen Interaktionsbeschreibung eines kommerziellen Produktes gezeigt.

Mehrere kleinere Beiträge ergänzen diesen Kernbeitrag: (i) ein Beweis der Turingvollständigkeit der gewählten Interaktionsbeschreibungssprache, sowie das Einbetten eines Kellerautomaten, (ii) eine Transformation auf eine semantisch äquivalente Zwischendarstellung als vereinfachte Zustandsmaschine durch geringfügige Erweiterungen des Standards, (iii) eine geschlossene Formel für die Obergrenze von Zuständen in der Zwischendarstellung als Zustandsmaschine, sowie (iv) mehrere Erweiterungen im Rahmen des Standards, um Anwendbarkeit und Ausdrucksfähigkeit der Interaktionsbeschreibungssprache in Bezug auf etablierte Modellierungstechniken zu verbessern.

All diese Arbeiten sind tatsächlich umgesetzt und ausprogrammiert in Form eines plattformunabhängigen, standardkonformen C++ Interpreters der betrachteten W3C Interaktionsbeschreibungssprache, welcher unter freier Lizenz quelloffen zur Verfügung steht. Dieser Interpreter wird bereits in kommerziellen, *multimodalen Dialogsystemen* eingesetzt und diente als eine der Referenzplattformen für den “Implementation and Report Plan” im Rahmen des W3C Prozesses eines Standards hin zu einer vollwertigen Empfehlung.

Contents

Part I.
Introduction and Context

1 Introduction

The field of Human-Computer Interaction (HCI) can be defined as the study of the *interaction between people and computers [concerning the] physical, psychological and theoretical aspects of this process* [DFAB03]. If we regard the physical aspect of this interaction to be a series of (however short) turns, wherein input of a user is followed, interrupted or overlapped by output of a computer system, or vice versa, we can conceive this interaction as a kind of *dialog* between a computer and a user. Here, the term dialog is, foremost, used to imply certain expectations of both participants with regard to the coherence of the overall interaction and provides a conceptual framework of user interaction for application developers. If a user interface allows to convey information relevant to the interaction via multiple communication channels, e.g. spoken commands in addition to keyboard input, we can call it *multimodal*. Such multimodal dialog systems are of great interest to complement traditional desktop interaction and can provide HCI in environments where traditional interaction paradigms are inapplicable or at least suboptimal. One prime example where multimodal interaction can be very beneficial is found in the domain of *pervasive environments*, wherein a user is permanently surrounded by a plethora of computing capabilities with various sensors and actuators.

As such, "multimodal dialogs in pervasive environments" identifies a subset of the overall design space of HCI and the claim of this thesis is to enable formal verification for this subset if an interface is modeled with the formalisms proposed. In anticipation to the proposed solution, we can formulate the problem as the question:

> *"Are there dialog management techniques applicable to realize interaction in pervasive environments that are verifiable with temporal logic?".*

Being able to proof temporal properties and constraints of such a multimodal dialog enables application developers to guarantee certain aspects of an interface and will lead to more reliable and trusted human computer interaction, improving the confidence in a system both for a user and the developers. We will consider this question solved, if we can show, or at least provide convincing arguments for the validity of the following chain of propositions, with the chapters of this thesis loosely aligned to elaborate on and assess their validity:

1. *Interaction in pervasive environments necessitates multiple devices and modalities.* (Chapter 2)

   We start to motivate the problem by first developing an intuition of multimodal dialogs in pervasive environments. With this intuition, we will set out to actually define the terms *(multi)-modality*, *dialog* and several related terms. Especially for the term *modality* a good definition is rather problematic as the notion of its root *mode* is ambiguous and changed over the years.

2. *Multimodal interaction in pervasive environments can be expressed in a dialog model on a suitable platform.* (Chapter 3 and 4)

   Having developed a more concise notion of multimodal dialogs, we will introduce a series of reference models for multimodal dialog systems and dialog management techniques to express dialog models as descriptions of interaction in general. With our ultimate goal to make these dialog models accessible to temporal logic formulae, we will consider the W3C State Chart eXtensible Markup Language (SCXML) and compare its expressiveness to the other techniques presented.

3. *These dialog models can be made accessible to the formalisms of temporal logic.* (Chapter 5 and 6)

   This part of the overall argument contains the majority of this thesis's contributions. Starting with SCXML as a description language for dialog models suitable to express multimodal interaction in pervasive environments, we transform a huge subset of its specified syntax and semantics onto PRocess MEta LAnguage (PROMELA) as the input language of the SPIN model-checker[1]. This allows an application developer to formally verify the dialog model with regard to temporal properties and constraints given in Linear Temporal Logic (LTL).

4. *The approach to 2 and 3 is applicable for non-trivial applications.* (Chapter 7.1)

   While we will already have shown functional applicability of the approach via transforming a series of tests accompanying the SCXML standard, we still need to show real-world relevance by applying the approach to a non-trivial dialog model in terms of size. This is achieved by transforming the dialog model of the Nvidia Shield handheld gaming console and performing an exhaustive search of its state-space.

[1] `http://spinroot.com`

This chain of propositions is visualized in figure 1.1, with each arrow representing a claim and will serve as a central theme to guide through this thesis. We do not postulate that every proposition must hold for every entity in the preceding class, e.g., not every dialog management technique is accessible to model checking with temporal logic and there might be interactions in pervasive environments for which a dialog system is unsuited. However, we will attempt to show that the overall argument holds for a fairly large and useful subset of interaction descriptions. Taking the approach outlined in grey, we will present an automated transformation of dialog models given in SCXML with an embedded scripting language onto PROMELA as the input language of the SPIN model-checker and assess its applicability with the dialog model of the Nvidia Shield handheld gaming console. This central argument is complemented by some general considerations for the formal properties of SCXML in chapter 5 and a few selected, not necessarily verifiable applications in chapter 7.

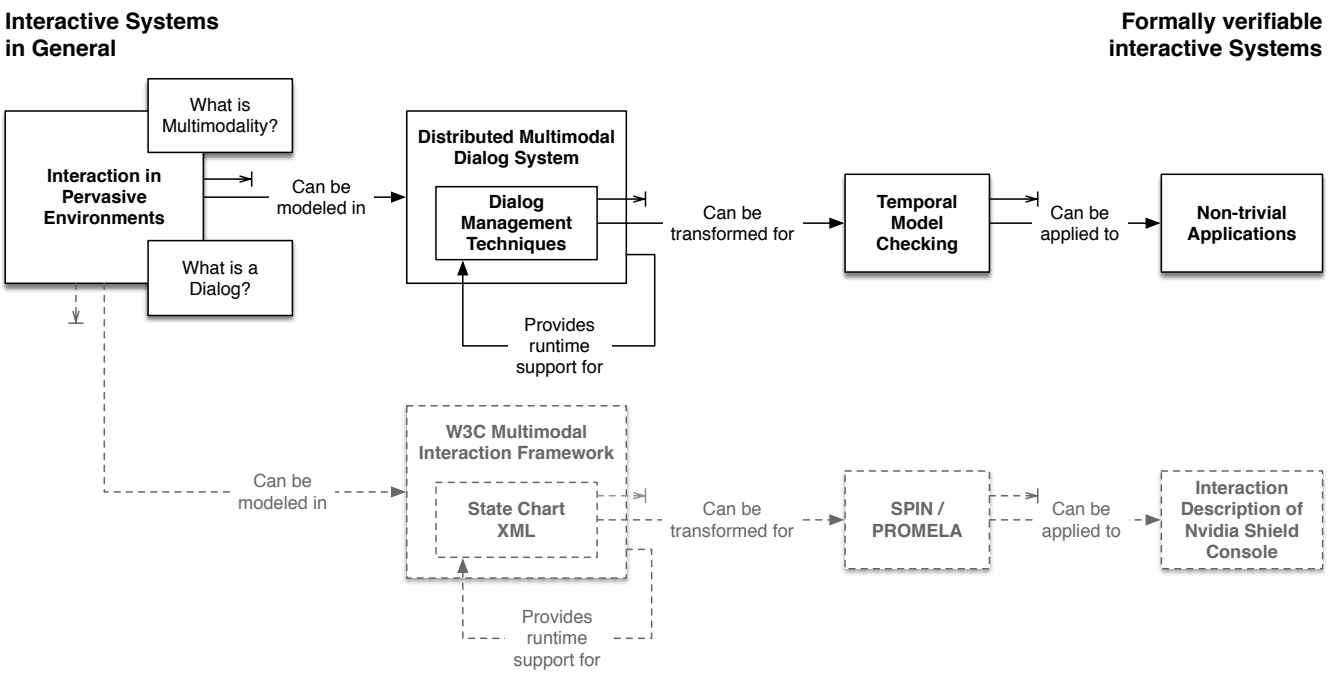


Figure 1.1.: Chain of propositions tackled in this thesis, with the approach to assess their validity outlined in grey.

The individual chapters are also organized into the three parts (I) *Introduction and Context*, (II) *Formal Aspects and Verification* and (III) *Applications and Conclusion* with the following primary functions: The first part introduces important terms and establishes the relevance of SCXML as a possible markup language to express dialog models for pervasive environments. The second part identifies some formal properties of SCXML and describes the transformation of SCXML onto PROMELA programs for the SPIN model checker. In the final part, all of this is applied to the SCXML document describing the dialog model of the NVidia Shield handheld gaming console and optimizations of the transformation are presented.

A Note on the Approach and Technologies

While we also might start with a formally verifiable language and specialize it into a feasible dialog management technique, we would only end up with yet another multimodal dialog system / modeling technique without much impact or proven relevance. Instead, the approach described in this thesis is based on a series of recommendations of the W3C Multimodal Interaction Working Group (W3C MMI WG) for multimodal dialog systems, i.e. we translate their formalisms onto the input language of a model-checker for temporal logic expressions. This has a number of beneficial effects:

- *Instant relevance of the approach.* If we can show that existing dialog models conforming to the standards of the W3C MMI WG can be formally verified with regard to their temporal properties, we have immediate applicability and relevance, if we accept that the respective World Wide Web Consortium (W3C) standards are relevant.

- *No explicit need to show the applicability of the dialog management technique.* If we were to propose our own formalism, we would need to evaluate its applicability to express dialog models for pervasive environments. By

relying on the standardized approach, this burden lies with the W3C MMI WG. However, this is easily shown by listing a set of published applications employing the approach.

- *Real-world examples to evaluate transformation.* If we were to seriously evaluate a custom dialog management technique based on a language intended for formal verification, we would need to write at least one non-trivial application first in order to establish applicability for a real-world dialog. Starting our exploration from an established dialog management technique increases our chances to find a convincing dialog model used in a real-world situation significantly.

- *Applicability of eventual infrastructure.* The implied expectation with employing a standardized approach for multimodal dialog systems is to reuse tools and infrastructure developed by the community and from eventual commercial offerings. There are already authoring environments for multimodal dialogs conforming to the W3C MMI WG recommendations, e.g. developed in-house by Nvidia [KN14b].

1.1 Contribution

The main contribution of this thesis is the description, implementation and evaluation of a largely automated transformation of state-charts, e.g. dialog models, specified in SCXML onto PROMELA as the input language for the SPIN model-checker. This enables the application of formal verification to proof temporal properties given in LTL. In this thesis, we are foremost concerned with the application of SCXML to express multimodal interaction in pervasive environments, but the general approach is just as valid for any other application of SCXML. The contribution is novel as the formal semantics of SCXML differ considerably from other formalization of state-charts, especially with regard to their strictly deterministic behavior. This is highly relevant as the plethora of state-chart formalizations lead to largely incompatible dialects (e.g. in the case of UML state-diagrams) and a W3C recommendation might develop the inertia to standardize the semantics. The contribution is evaluated with regard to:

- The subset of SCXML language features that remains available for formal verification. To this effect, we consider the 182 action-language agnostic tests of functional requirements accompanying the SCXML standard and attempt to transform and verify them (section 6.4.13). This is possible for 132 tests, with the majority of unavailable features being due to the semantics of error-handling at runtime.

- Applicability to real-world problems. Here we consider the dialog model for the Nvidia Shield handheld gaming console as the most convincing SCXML dialog model we could attain (section 7.1.1). This dialog model was developed without its original authors being aware of the approach presented in this thesis.

- The runtime requirements in terms of time and memory usage for an exhaustive verification of this Nvidia Shield SCXML dialog model (section 7.1.4).

This main contribution is enabled by two smaller, but still individually noteworthy, contributions:

- A description, implementation and evaluation of an automated transformation of *any* SCXML document onto a semantically equivalent SCXML state-machine (in section 5.3). Semantic equivalence is shown via the complete set of 232 tests accompanying the SCXML specification. This intermediate state-machine representation can form the basis for a transformation, not only onto PROMELA, but onto many other platforms such as resource constrained devices and eventually even ASICs and FPGAs. It also allows a target platform to ignore the finer points of state-chart semantics, e.g. with regard to composite states, transition preemption and event name matching.

- A PROMELA data-model for SCXML to increase the expressiveness of the formally verifiable subset (in section 6.3). Pending a more thorough analysis of the system descriptions for other model-checkers, and given the selective approach with regard to PROMELA's syntax and semantic we isolated, this more expressive semantics might still be transformable for other model-checkers (e.g. the Symbolic Model Verifier (SMV)).

Several smaller contributions complement this main contribution:

- A proof of the Turing completeness of SCXML by embedding a Deterministic Queue Automaton (DQA) as an equivalent formalism in SCXML (section 5.4.2).

- An upper bound for the number of states in the intermediate state-machine representation, briefly evaluated for its tightness and correctness (section 5.3.8).

- A Prolog data-model for SCXML to allow logic programming and unification-based transition guards. This extends the applicability of SCXML onto more elaborate dialog management techniques, such as actor-based and Information State Update (ISU)-based approaches (see chapter 4.5). Feasibility is shown by embedding an existing dialog move engine [LLC+00] for ISU-based dialog management.

- A data-model agnostic debugger for SCXML, complete with breakpoints, variable inspection and a web-based interface (section 7.2.6.1). It defines a slim REST-based protocol to abstract from any concrete SCXML interpreter.

Finally, though not a scientific contribution, all of this is based on our own SCXML implementation[2] written over the course of two years and made available with a permissive, open-source license. The interpreter implements all of the SCXML recommendation and was submitted as a reference platform for SCXML to attain the "candidate recommendation" status as a phase towards a full W3C recommendation. It is in active use as part of a multimodal dialog system at Cibek, a German SME for home automation and ambient assisted living.

1.2 Publications

Parts of this thesis have already been published in book chapters, journals, magazines, conference proceedings and workshops. A presentation of dialog management techniques was already published in [SWR12b] with an updated list [SWRRA13] published one year later. A general overview of the W3C MMI WGs recommendations and their interdependence was published in [SWR12a].

An early attempt at a transformation from SCXML to PROMELA was described in [RNSW14]. The debugger for SCXML documents was already presented in [RSWS14]. The Prolog data-model to extend the expressiveness of SCXML towards logic programming and unification was presented in [RSWRA13] with an application for vision-impaired users presented in [SWRRAM14] and [SWRM14]. Another application modeled with SCXML employing spatial audio and graphical output as modalities was presented at [RSW13].

We organized two SCXML workshops co-located with the EICS conference, the first in 2014 [RSWL+14], the second workshop in 2015 [SWRBM15].

In earlier approaches to provide interfaces in pervasive environments, we focused on augmenting a spoken dialog system with additional modalities and made sensor data and actuators available to the dialog manager. These attempts can still be found in [RSW10], [SWR10], [SWR11] and [RSW12]. Ultimately we embedded the spoken dialog system into SCXML as a modality agnostic dialog management technique with the approach presented in [SWRMua13] and subsumed as a book chapter in [SWR15].

[2] `https://github.com/tklab-tud/uscxml`

2 Interaction in Pervasive Environments

The purpose of this chapter is, foremost, to establish a common understanding of the terms *pervasive environment*, *multimodality*, *dialog*, their interrelation and to give a first idea for an application of formal verification. With regard to the central argument of this thesis this chapter will provide the arguments as to why interaction in pervasive environment necessitates, or at least greatly benefits from multimodal interfaces (see figure 2.1).

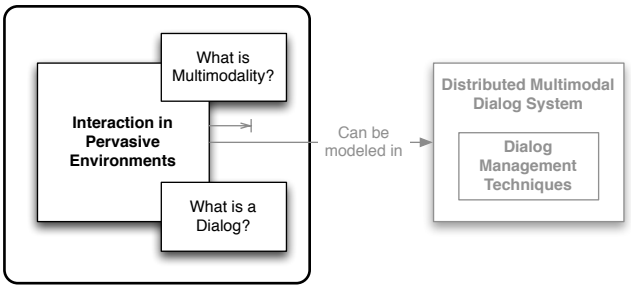


Figure 2.1.: This chapter tackles the proposition "*Interaction in pervasive environments necessitates multiple devices and modalities.*" from the main argument of this thesis (cf. figure 1.1).

We start by developing an intuition of pervasive environments as a variation of ubiquitous computing and regard the problem of providing suitable interaction paradigms to end-users in these environments. This will lead us to the concept of *multimodality* as a generalization of interaction devices and the concept of *dialog* as an abstract conception for Human-Computer Interaction (HCI) to coordinate the modalities and applications in a coherent, overall interaction paradigm. In this first part we will have to resort to some imprecise language related to the terms of *multimodality* and *dialog* as definitions for both are either disputed or not exactly obvious, warranting a much more detailed discussion. In section 2.4.1 and 2.4.2 we will trace both terms more rigorously through the years of scientific research and provide a perspective on the development of their meaning and connotations before we settle for some actual definitions. Finally, the introduction of the term *dialog* will conclude with an early reference model for a dialog system, which sets the stage for the next chapter about multimodal dialog systems.

2.1 Pervasive Environments

> The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.
>
> Mark Weiser 1991

In 1991, Mark Weiser published a widely noted article in the "Scientific American" [Wei91] wherein he described an, at the time, futuristic scenario with computing capabilities surrounding the user ubiquitously. He envisioned these technologies to become so engrained in our environment, so *weaved into the fabric of everyday life* to the point of disappearing from our perception, not unlike written text. Computing capabilities would permeate all objects and support users in a ubiquitous, yet invisible network via casual, natural interactions. Three key areas of research were identified by Weiser in order to realize such a scenario:

1. Technological requirements with regard to **processing power, display sizes and power consumption**. All of which are surpassed by current off-the-shelf technologies, with the possible exception of light-weight, paper-like displays.

2. **Network interfaces** to connect all the devices via wired high-speed interfaces, long-range wireless and tiny-range wireless connections, which find their counterpart in todays Gigabit Ethernet, Wireless LAN and Near-Field-Communication technologies.

3. **Software** for such ubiquitous applications, which can be argued to be the area with the least progress towards reliability, standardization and commoditization up until today.

In retrospect, one of the important contribution of Weiser's article was the implied observation that the cardinality of the human-computer relation was about to change (see figure 2.2): Up until the era of the desktop personal computer starting a few years prior to Weiser's article (at around 1985), the relation of interaction between man and computer was characterized by multiple users *competing for the attention* of a single mainframe computer [VD97] (N:1). For quite some time this could, literally, mean to stand in a line to use a dedicated console. In a later variation this

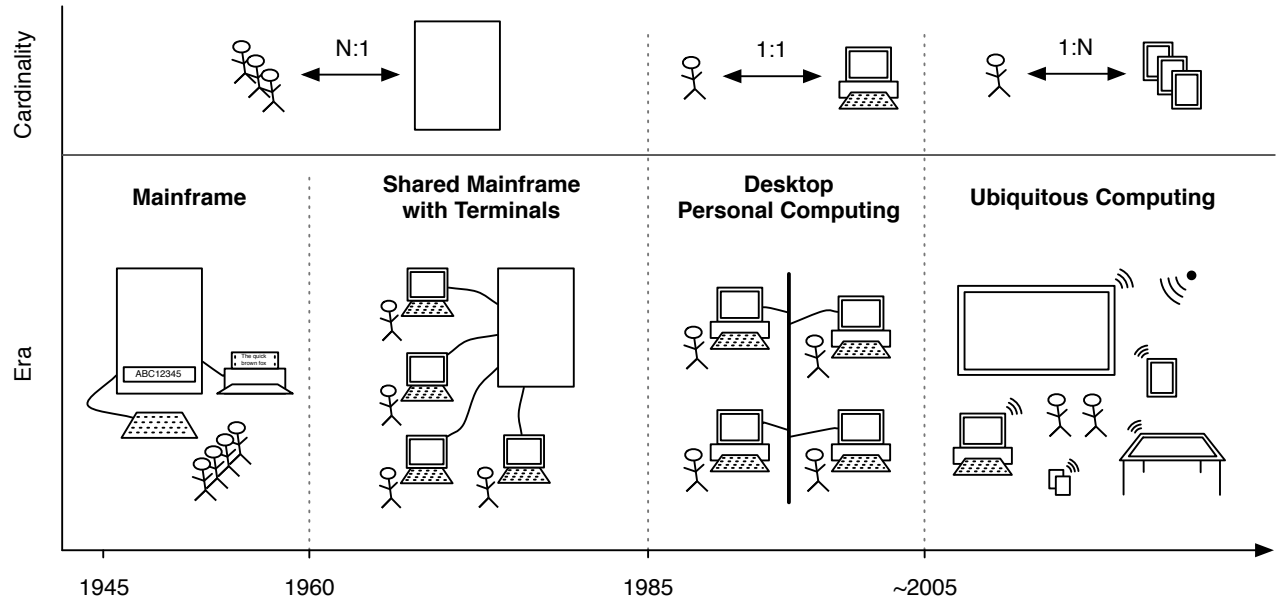


Figure 2.2.: Development of the men-computer relation over time.

was relaxed by utilizing central computing capabilities of a time-shared mainframe computer on a personal terminal. With the advent of the personal computer, the cardinality of this relation was changed for the first time by providing affordable computing capabilities explicit to every individual user (1:1). With ubiquitous computing, this cardinality would change once more to multiple computing devices competing for the attention of a single user (1:N). It is difficult to pinpoint a definite moment in time, when this approach became predominant, or in fact whether it even already has. In figure 2.2, the year 2005 is chosen somewhat arbitrarily as smart-phones became increasingly popular.

The original vision of *Ubiquitous Computing* has, with a different emphasis, been revisited in various variations e.g. as *The Internet of Things*, *Physical Computing*, *Pervasive Environments*, *Ambient Intelligence* or *Smart Environments* [Pos10]. In this thesis, we will concern ourselves with "pervasive environments" as a conception of ubiquitous computing wherein physical spaces are populated by a plethora of interconnected devices or objects that provide access to sensors, actuators, storage and general purpose computations. There are two inherent characteristics that we consider particularly challenging when attempting to provide user interfaces in these environments:

1. Both, the user facing components (i.e. the sensors and actuators), as well as the functional components (i.e. general computation and storage) are assumed to be **distributed** throughout the environment.

2. The wide range of situations in which user interaction may take place, necessitates to **employ many sensors and actuators** concurrently or sequentially to realize reliable interaction.

These characteristics motivate corresponding requirements for a description of interaction in such environments. The distribution of components is reflected by the requirement for an interaction description to explicitly, or implicitly support a distributed execution with network transparent synchronization among the participating components. The wide range of situation and the consequential necessity to employ many sensors and actuators motivates the requirement to support multiple *modalities* as equals. Where traditional Window Icon Menu Pointer (WIMP) interfaces assumed a user sitting in front of a personal computer with dedicated input and output devices connected, no such assumption can be made for interaction in pervasive environments. The user may be engaged in other tasks, move around freely in a potentially noisy or dirty environment and only occasionally interact with a dedicated device. This will, ever again, cause the preferred modality or the concrete set of employed sensors and actuators to change during an interaction and the interactive system will have to maintain the interaction context accordingly.

2.2 Suitable Interaction Paradigms

While the first coherent description of ubiquitous computing with the change in cardinality of the HCI relation is usually attributed to Weiser, the question of suitable means of interaction in these environments is less clearly attributed. Weiser himself only exemplifies interaction with three classes of devices he called *tabs*, *pads* and *boards*.

Later work reinterpreted and generalized these classes into the more abstract concepts of (i) *smart dust* as micro-electro-mechanical systems ranging from nanometers to millimeters in size and lacking visual output, (ii) *smart skin* as (non-planar) display surfaces and (iii) *smart clay* as ensembles of micro-electro-mechanical systems formed into arbitrary three-dimensional shapes [Pos10], but still does not provide a general answer to the question of suitable interaction paradigms in these environments.

At the time of Weiser's publication, personal desktop computers were well established and interaction via mouse, keyboard and graphical displays, organized in the so-called WIMP paradigm was predominant [VD97]. This approach to interaction, however, limits the possibilities for interaction in pervasive environments as it implies dedicated in- and output devices with a focus on graphical displays. If we are to realize the ideal of *disappearing technology, woven into the fabric of everyday life* [Wei91], we cannot constantly remind the users of its presence by requiring them to carry dedicated and specific interaction devices at all times.

In order to identify a general interaction paradigm more suited for pervasive environments and to establish a general design space of HCI, it is helpful to gain a perspective by looking back at earlier interaction research in general.

Even before the term "ubiquitous computing" was coined, many researchers, authors and film directors popularized or at least implied certain expectations as to how humans, in the future, would interact with computer systems [SEB08]. An early vision about the possible relation of humans and machines in general is described as "Man-Computer Symbiosis" by Licklider in 1960 [Lic60]. In this symbiotic relationship, the main responsibility of the computer would be to facilitate formulative thinking and to cooperate with a human to make decisions regarding complex situations. At a time when batch-processing of computations on mainframe computers via punch-card input and line-printer output was the predominant interaction paradigm, Licklider already theorized about the desirability and feasibility of interfaces that would become possible only decades later. He classified some general challenges with one class pertaining explicitly to suitable interaction paradigms:

1. **Desk-Surface Display and Control** to *write notes and equations to each other*. Herein, the human would sketch functions as graphs and generally write instructions to the computer via handwriting or flow-charts and the computer would typeset and evaluate the human's written input.

2. **Computer-Posted Wall Display** to *simultaneously present information to all men* on a shared display to coordinate interaction between a computer and a team of human users.

3. **Automatic Speech Production and Recognition** as the *most natural form of interaction*, allowing computer specialists and non-experts alike, to perform real-time interaction in a symbiotic relation.

The years directly following Licklider's publication saw the advent of time-sharing on mainframe computers and a transition to interaction via dedicated computer terminals. These devices would feature a keyboard for data entry and character-oriented displays to show the results computed on the shared mainframe; users would still share the computing capabilities of a central mainframe, but interaction was more personalized, intensifying research with regard to interaction and usability. A timeline of important contributions between 1960 and 1998 is given by Myers in [Mye98] and summarized in figure 2.3.

To motivate the interrelation of the terms *multimodality* and *dialog* with pervasive environments, it is helpful to differentiate two important aspects of interface research, namely:

1. The **metaphors** employed to borrow syntax and semantics for the interaction from something already familiar to a human user (e.g. direct manipulation of graphical objects or windows as views into an application).

2. The **devices** to provide the means to communicate interaction intents to and from a computer system (e.g. the mouse or a keyboard, but also gesture recognition).

It can be argued that those two form the main dimensions of a design space for HCI. Some authors also include the actual application to span this design space (e.g. [CMR90]). For our motivation, though, the actual application is of no direct consequence: Everything that can be perceived by a user is given by the state of the devices with their interaction syntax and semantics conveyed via the metaphor and, by extension, the application domain.

For instance, in Licklider's work three device ensembles are proposed: (i) interactive screens embedded into a table for user input via an electronic pen and graphical system output on the same surface, (ii) large, wall-mounted displays and (iii) microphone and speakers for interaction via voice. The accompanying metaphors are somewhat less clearly identified, but the choice of the term *symbiosis* in the publication's title and the example interactions given do imply an overall metaphor of a *critical computer clerk*, comparable to an actual person that performs computations, not unlike interaction with early, human computers in the original sense.

It is these two dimensions, metaphor and devices, that can be generalized into the terms *dialog* and *modality* as we will argue below. Do note that the following two sections will only attempt to motivate the interrelation of *modality* and *dialog* with interaction in *pervasive environments*. We will define both terms much more rigorously in section 2.4.1 and 2.4.2 respectively.

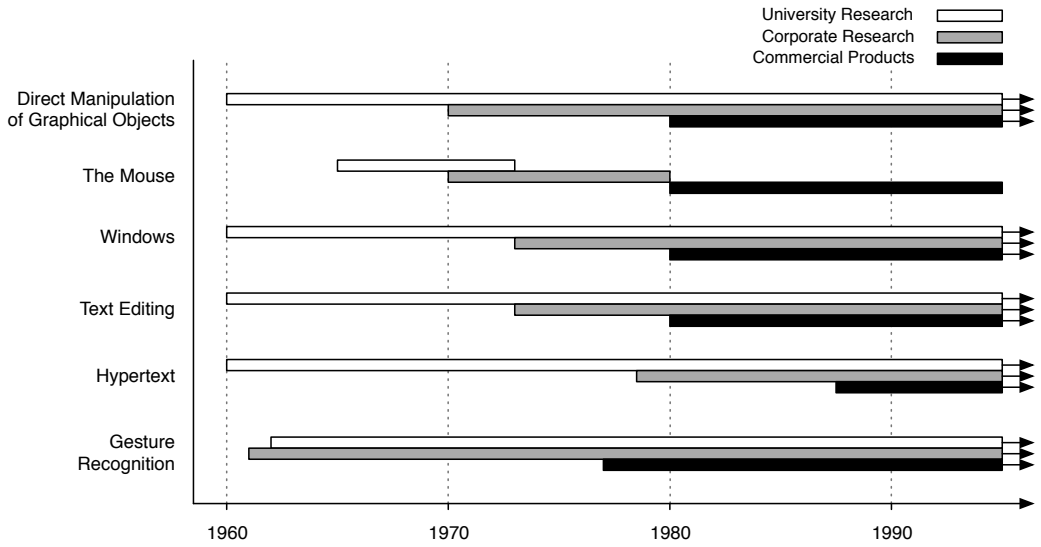
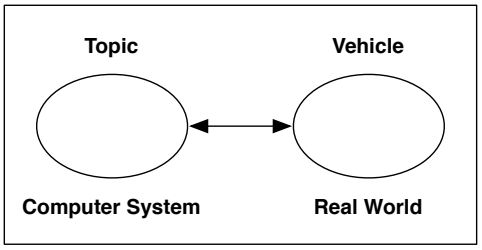


Figure 2.3.: Important contributions to the field of HCI since 1960 (from [Mye98]).

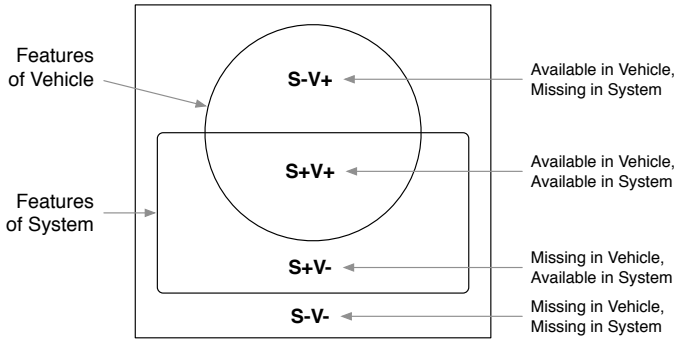
2.2.1 Dialog as a Meta-Metaphor

Anderson introduces the term metaphor as a means to *convey the functional attributes and action-oriented possibilities of a system to the user* [ASK+94]. Following established nomenclature he differentiates between (i) the *vehicle of a metaphor* as something familiar to a user with well established features and behavior and (ii) the *topic of a metaphor* as the object to which the features and behavior of the vehicle are attributed figuratively (see figure 2.4a).

> **Metaphors**: *convey the functional attributes and action-oriented possibilities of a system to the user.*
>
> `- Anderson 1994 [ASK+94]`



(a) Psycholinguistic conception of a metaphor.



(b) Interaction of vehicle and system.

Figure 2.4.: Metaphors as applied to HCI (adapted from [ASK+94]).

When applied to HCI, an interface borrows concepts familiar to a human user with regard to their features and behavior and aligns them with concepts of user interface elements. Usually, there is no exact equivalence and some of the features available in the system are missing from the vehicle and vice versa (figure 2.4b).

If we generalize the term, we can conceive a metaphor as a scaffolding for a mental model of a user to assign and organize functionality and responsibilities to entities of a system; it allows a user to predict the system's interactive behavior. While the intersection of the *functional attributes and action-oriented possibilities* between the metaphor and the system will already allow a user to develop a first intuition for an interactive system, it is an understanding for the complements of both sets that will lead to fluency in interaction. Therefore, any conceptualization of a user interface will, eventually, develop into a *vehicle* for a metaphor on its own as users become intrinsically familiar with its concepts. E.g. in the context of the desktop metaphor, the necessity to group and manage documents related to any specific task established the concept of an *application* with functional attributes and action-oriented possibilities

that are, e.g., found again in *apps* on mobile devices. This extended notion is given additional credence when we consider that the dimensions of a space ought to form an (orthogonal) base and that any given point in the space ought to be expressible as a linear combination of its base dimensions. In this case, every possible user interaction paradigm ought to be expressible as a selection of devices, metaphors (and applications).

If we accept this generalization, we can conceive the metaphor dimension of the HCI design space as the general approach of an interactive system to organize user input and system output into a coherent user interface. In order to describe this interactive behavior, a user interface developer will, in turn, rely on other, more abstract conceptualizations of interaction itself as a scaffolding for a mental model. E.g. a popular approach to describe graphical user interfaces is the Model-View-Controller (MVC) pattern (see figure 2.5). Herein, all of the domain dependent application state is contained in the *model* and the *view* displays a subset of this state to a user. The *controller* will listen for user input conveyed via the interactive elements provided by the view and updates the *model* accordingly, which will cause the *view* to get updated to reflect the changed state and interactive elements.

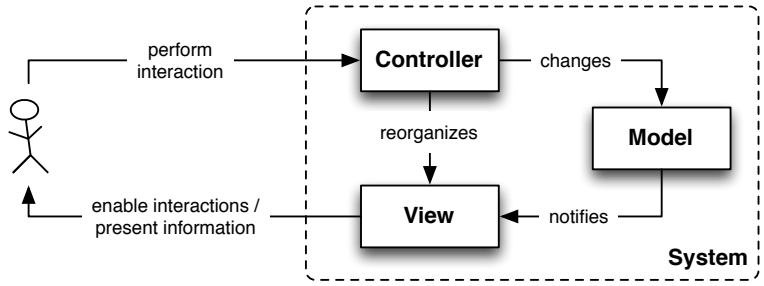


Figure 2.5.: Relations and responsibilities of the Model-View-Controller pattern.

It is on this level of abstraction that the term *dialog* is to be conceived: as an abstract conceptualization to describe interactive systems. I.e., a mental scaffolding for a user interface developer to describe systems adhering to mental scaffolds of a user; a metaphor to describe metaphors in the extended notion of the term. It is *not* to be conceived in its more obvious notion of a spoken interchange between participants, but borrows functional attributes and action-oriented possibilities and applies them for (multimodal) interaction.

In this thesis, we will use the term *dialog* as the central metaphor for user interface developers to model all interactions and conceive any user perceptible metaphors as instances of such a dialog. In this conception, the human and the computer are engaged in a continuous back and forth of turns, communicating interaction intents and information and delivering an interaction intent will change the state of the dialog. In reference to figure 2.4b from Anderson, table 2.1 contains a comparison between the dialog as a general conception for interactive behavior and actual dialogs found e.g. between two humans.

Conceiving all interaction as such a dialog may seem construed at times, e.g.: how would one interpret the dragging of a graphical object as a sequence of turns wherein even minuscule movements of a pointing device by the user would require an immediate system response with a graphical rendering on an updated position? But if we allow both parties to interrupt each other at all times and pose no lower limit on the duration of a turn, we can force us to conceive every other metaphor as a special dialog. This is important as it provides a generic conceptualization for user interface developers that can be refined into any concrete metaphor offered to a human user.

Do note that this conception does not imply or suggest any particular class of devices but abstracts them into a more general notion of "means to communicate interaction intents and information", a point that becomes very relevant in the scope of multimodality below. We will discuss the dialog metaphor as a conception for interaction along with related issues and its historical development more detailed in section 2.4 below.

2.2.2 From Devices to Multimodality

When considering devices suitable for interaction in pervasive environments, it is helpful to have a look at the various definitions that were proposed:

> **Input Device**: *is a transducer from the physical properties of the world into logical values of an application.*
> - Baecker and Buxton 1987 [BB87]

> **Input Device**: *is part of the means used to engage in dialog with a computer or other machine.*
> - Card et al. 1990 [CMR90]

	Found in real-world dialogs (V+)	**Missing in real-world dialogs (V-)**
Found in a dialog system (S+)	Common features (S+V+): • Structured interchange of ideas between two (or more) participants. • Can be conceived as a sequence of (overlapping and interruptible) turns. • Dialog state is progressed by providing information and intents in a turn. • Subject of the dialog can be very specific and concrete or general and abstract. • Set of sensible things to "say" depends on dialog state. • Both partners establish facts as the things they agree upon as the dialog progresses. • Ambiguous or incomplete expressions can be clarified in a sub-dialog or need to be resolvable via the dialog's state.	In system, but not in vehicle (S+V-): • A turn is not necessarily a spoken utterance, any successful conveyance of an interaction intent might be suited to progress the state of the dialog. • Turns are usually discretized, sometimes with a notion for overlapping or interruptions. • Turns may have a duration measured in milliseconds.
Missing in a dialog system (S-)	Not in system, but in vehicle (S-V+): • Domain of dialog is constrained to a given task. • More formal with regard to the syntax and semantics conveyed in a turn. • "Human" aspects of an actual dialog are usually deemphasized.	Neither in system, nor in vehicle (S-V-): • Actual subject of the dialog is not implied.

Table 2.1.: Comparison of real-world dialogs and the dialog metaphor.

While both definitions speak of *input* devices, they are general enough to apply to output devices if we simply reverse the role of computer and user. The latter publication by Card et al. is of particular interest as its definition is followed by a remark that illustrates an important point: *"The dialog is not, of course, in natural language, but is conducted in ways peculiarly suited to interaction between human and machine".*

Even if we exclude natural language as a means to drive the dialog, there would still be other means to express interaction intents or, more generally, convey information, possibly relevant to a human-computer dialog in pervasive environments, which defy classification as a classical interaction device, e.g. gestures, postures or even gazing. Therefore, the notion of *device* itself ought to be generalized into something *suited to convey an explicit or implicit interaction intent or information* and, pending the more thorough definition in section 2.4.1, we will call this a *modality*. As different situations of a user in a pervasive environment will enable or prohibit the interaction via any given modality (e.g. voice interaction is unsuited in noisy environments, multi-touch requires at least one hand to operate), any conceptualization will have to support sequential or parallel use of different modalities [NC93], thus the term *multi*-modality.

Apart from the requirement for multimodal interfaces due to the potential inapplicability of any given modality in a certain situation found in pervasive environments, there are two additional benefits of providing such user interfaces, which can be summarized in two major points:

1. **Combining many error-prone modalities increases overall accuracy:** The main idea is that errors introduced via the shortcomings of one modality can be mitigated by referring to another modality. E.g. spatial references when uttered via voice, such as "the yellow cube behind the pillar" are more easily resolved when deictic gestures, such as pointing, are taken into account as well. [Ovi03]

2. **Providing different means of interaction reduces cognitive load:**

   The term *cognitive load* was coined by John Sweller in 1988 [Swe88] to refer to the mental effort a user experiences when solving a given problem. The term is later differentiated [Swe10] into three distinct categories

that influence the overall cognitive load: (i) *intrinsic cognitive load* as the complexity inherent to the information being presented or the current task at hand, (ii) *extraneous cognitive load* as induced by the manner of presentation, e.g. when unsuited or hard-to-use interfaces divert mental resources from the actual task and (iii) *germane cognitive load* also induced by the manner of presentation, but experienced when the interfaces are supportive and help the user in *schema acquisition*.

With intrinsic and extraneous cognitive load being a burden and germane cognitive load being helpful, we can, informally, define the perceived cognitive load as:

$$\text{Cognitive Load}_{\text{perceived}} := \frac{\text{CL}_{\text{intrinsic}} + \text{CL}_{\text{extraneous}} - \text{CL}_{\text{germane}}}{\text{Human Working Memory}} \tag{2.1}$$

As the perceived cognitive load is something very subjective to a person, the *human working memory* is introduced to normalize cognitive load as defined by Sweller. In 1974, Baddeley and Hitch introduced a model of human working memory [BH74], describing a *phonological loop* and a *visuo-spatial sketchpad* as two modality specific subsystems of human perception coordinated by a *central executive* in which human cognitive processes are performed (see figure 2.6). Later extended to include an *episodic buffer* to integrate information from different sources [Bad00], it has become an important model to explain respective human cognitive processes.

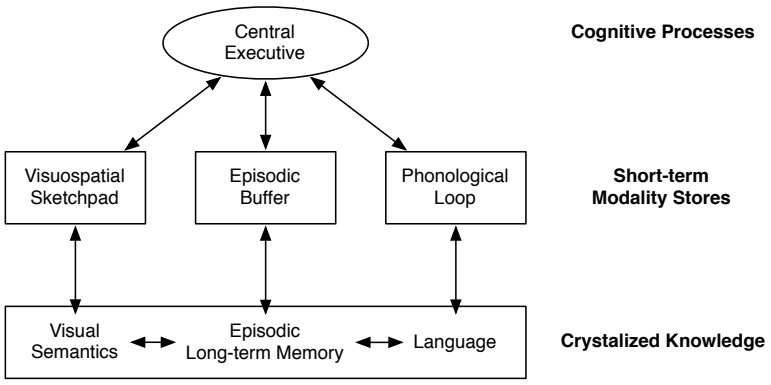


Figure 2.6.: Baddeley's model of human working memory [Bad00].

Thus, in order to reduce the overall cognitive load experienced by a user of an interface in equation 2.1, the following options are available:

a) **Reduce the intrinsic cognitive load:** This is usually not possible as the intrinsic cognitive load is inherent to the complexity of the task at hand. The task itself already implies this part of the cognitive load as it is either hard or easy to perform.

b) **Reduce the extraneous cognitive load:** This part of the overall cognitive load can be reduced by avoiding unsuited presentations of a problem or ill-fitting interaction paradigms. When applied to a user interface, choosing e.g. a textual description for an inherently graphical object or choosing the wrong graphical widget for interaction might be examples.

c) **Increase the germane cognitive load:** This is usually achieved by choosing representations and interaction paradigms that are well suited for the problem and help in *schema acquisition*. Interaction idioms suited to the data, apt metaphors and generally conforming to a user's expectation.

d) **Increase the human working memory:** The reduction of extraneous cognitive load and the increase of germane cognitive load are usually available with any kind of interface. Baddeley's model of human working memory suggests that this part of the overall cognitive load can be increased by making beneficial use of the various modalities, i.e. providing information on the visual as well as the acoustic channel as part of multimodal interfaces. This assumption has also been verified via various experiments, e.g. [Ovi06].

The case for multimodal interfaces is especially strong in pervasive environments, where traditional interfaces are inapplicable or at least a hindrance and the availability of modalities is dependent on the situation or task to be performed. A more thorough introduction and definition of *multimodality* is given in section 2.4.1 below.

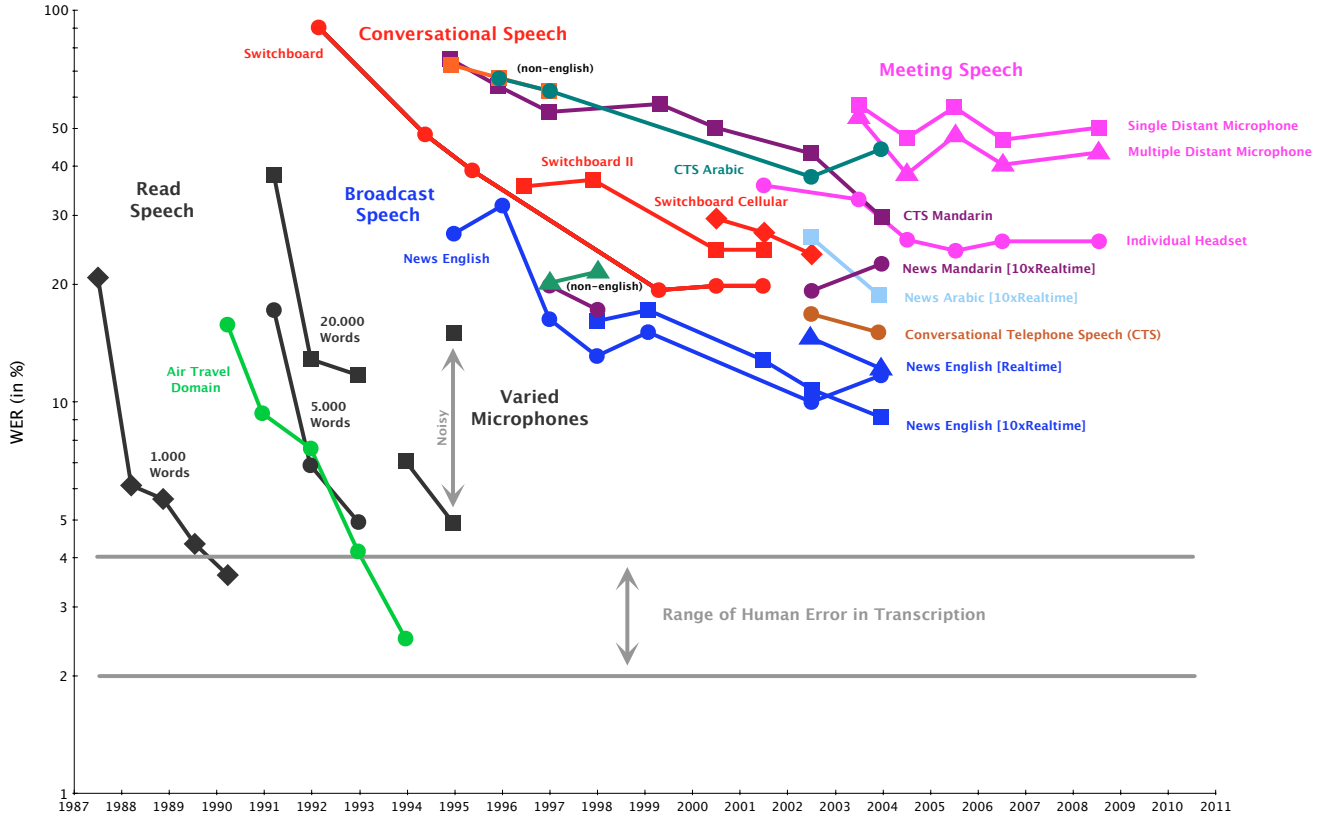


Figure 2.7.: Development of Word-Error-Rate over time [FGRM09].

2.2.3 A Note on Voice User Interfaces

Given the dominance of spoken communication to convey interaction intents and information between humans, speech might seem like the most natural, most obvious modality for an interaction paradigm in pervasive environments. And ever since the first successful applications of automated speech recognition were realized (i.e. [DBB52, Dud55]), researchers and visionaries were excited about the possibilities. Yet, the persistent problems with recognition errors and the inaccessibility of natural language to the formalisms required by a computer system keep the applicability of voice user interfaces limited.

While there was, ever again, some remarkable progress with regard to automatic speech recognition, it is, as an exclusive modality, still too unreliable for dependable user interfaces. Yearly evaluations, coordinated by the National Institute of Standards and Technology up until 2009 concluded with a Word-Error-Rate (WER) of well above 40% for distant speech recognition of spontaneous, overlapping speech (see figure 2.7). While more recent developments with improved acoustic models employing deep neural networks [VGBP13, SKRP15] were able to improve WER for this task considerably, the range of WER for human understanding of distant, continuous and possibly overlapping speech is still unapproachable.

The disappointment with regard to speech as a dominant modality lead many researchers to shift their focus onto multimodal interfaces. While speech was still considered to be important, its role was deemphasized to be one of many modalities available to interact with a computer system in a pervasive environment. However, many of the conceptualizations and solutions developed as part of voice user interfaces are still beneficial when applied to multimodal interfaces. Solutions to cope with the inherent invisibility and transience of spoken utterances, their one-dimensional nature and the entailing requirement of coherent dialogs and the various error-prevention and correction strategies, transcend the modality of speech and provide solutions applicable for multimodal interfaces as well.

2.3 Formal Verification

If we conceive all interaction as a dialog, and a dialog as a series of turns in which interactions change the state of a system, we can formalize such an interaction as a state-transition system. The transitions are driven by user's input, system's output or just about any other event delivered to the system, with the system's state defining the set of permitted events and the system's reaction. In chapter 3.2, we will introduce various *dialog management techniques*

and, in fact, it can be argued that they all follow this basic notion. While they differ considerably in their conception of a dialog state or their approach to transitioning, changing state in response to events is a fundamental commonality.

If we are somewhat careful with the syntax and semantics of a dialog model as a description of interaction, we can employ temporal logic to describe restrictions and soundness properties of the dialogs modeled. E.g. we might formulate that a successful conveyance of an (implicit or explicit) interaction intent will have a specific effect or that some state of the dialog will always entail a specific consequence. If we imagine an elaborate dialog model for all interactions related to a smart home, it would be very beneficial if we could guarantee the following temporal properties:

- "Leaving your house will always turn off your heaters and light".
- "There is always at least one sequence of events that will allow you access to your house" or the related "No sequence of events results in a locked door with the keys still inside".
- "Opening the door of a microwave oven will always deactivate its radiation emitter".
- "Talking on the phone will always mute your stereo if you are in the same room".
- "Your dishwasher will only run if the price of power is below a certain price point".

Obviously, all of the implied physical or logical sensors and actuators would need to be available to the platform interpreting the dialog model. This also establishes the system boundaries for the formal verification: while we may be able to verify these temporal properties on the dialog model, we can not guarantee that this will be the system's actual behavior. The dialog did specify the system to behave as such, but we can not formally verify the interpreting platform itself. Formal verification of specific dialog models constitutes the major contribution of this thesis and is discussed in detail in chapter 6.

2.4 Definition of Multimodal Dialogs

While the previous section established an intuition about the terms *multimodality* and *dialog* - and motivated their necessity, or at least desirability and interrelation for interaction in pervasive environments - the following sections will align this intuition with more established nomenclature and introduce some related concepts. We will attempt to give an overview about the development of both of these terms as different variations introduced throughout the years of scientific research and finally develop a definition for *multimodal dialog* as the subject of formal verification in the remainder of this thesis.

The body of research with regard to either of these topics is vast and spans a timeframe of more than 50 years of scientific publications each. Inconsistent or outright conflicting definitions, often by enumerating examples, implied assumptions, missing delineation to related terms and different narratives prevent any pretense to present a holistic introduction here. A more objective exploration of these terms would require extensive literature review with regard to bibliometrics. As such, this thesis will only regard a select few publications deemed important due to their contribution to the development of the terms.

2.4.1 Multimodal Interfaces

> I intend first to attempt to untangle the terms multimedia, multi-modal and multi-modality. Do they convey important distinctions?
>
> Mayes 1990

In order to establish a more concrete meaning for the concept of a "multimodal dialog", we first need to define the term *multimodal* as a quality of user interfaces. While we already developed a first intuition in the previous sections a good, unambiguous definition is surprisingly hard to find. There seems to be no solid consensus, even among seasoned researchers, with regard to its actual definition. In this section, we will attempt to trace the term through the years of research, present different definitions by various authors to establish connotations the term may carry in related work and finally settle upon one definition. A timeline of important publications relevant for the development of the term or even with explicit attempts at a definition is given in figure 2.8.

The "Put that there" prototype of Bolt [Bol80] as part of the "Media Room" project [Don78] is widely credited as the first multimodal interface (e.g. in [MW98, Ovi03]) but attempts no definition. In fact, the term *modality* is not

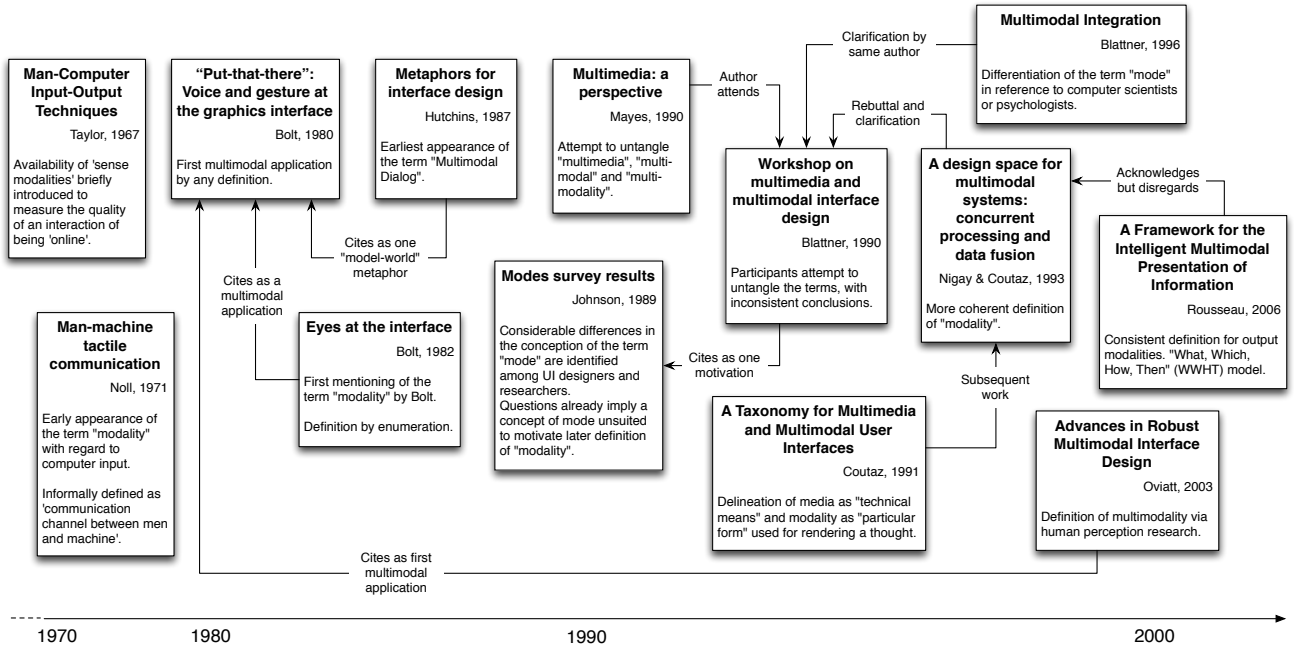


Figure 2.8.: Timeline of selected publications relevant for establishing the term *modality*.

even mentioned in the publication itself. Only in a later publication [Bol82] did Bolt explicitly refer to the concept of multimodality. However, he uses the term rather casually with no citation and, again, no attempt at a definition. The term itself appeared as early as 1967 in a publication from Taylor [Tay67] in the context of a "sense modality" between two humans to argue for a face-to-face meeting being more *online* than a telephone call. Later, in 1971 in the work of Noll [Nol71] an implicit definition is given in reference to haptic interfaces as a *communication channel between men and machine*, but with no citation of related, earlier work. Neither of these publications are helpful to identify any authoritative publication wherein the term is initially introduced to the field of HCI. While the term *modality* and the related terms of *mode*, *medium* and *channel* are subsequently employed in numerous publications by various authors, there were considerable misconceptions about their actual meaning and no clear delineation.

In 1988, a survey to assess the level of consensus regarding the term *mode* was conducted by Johnson and Engelbeck with its results presented in [JE89]. A selection of 32 user interfaces were described and 46 participants, all researchers from the field of HCI, were asked to rate these interface on whether they are *moded* or *modeless*. While there were considerable differences in the classification of these interfaces, the general contemporary notion was aligned with the concept of *mode* described in [Tes81], where e.g. a text-editor would be in a mode such as `INSERT`, `REPLACE`, `DELETE`, or `SEARCH`, causing user input to be interpreted differently with no end of confusion for a novice user.

The problems with coherent definitions were subsequently bemoaned by Mayes in 1990, when he, somewhat jokingly, referred to the whole conglomerate of terms as "the M-Word" [May90]. In an attempt to untangle the various terms, he came up with the following definitions:

> **Mode**: *in the interactive sense may simply be a dimension of dialog, or in the computer science sense, a state of a computer. Yet, another view would be that a mode is defined by the nature of the information being handled.*
>
> **Modality**: *of an interaction can refer either to the particular sensory system the user is engaging: audition, vision, touch; or it also may refer to the essentially spatial or verbal nature of the information.*
>
> **Medium**: *can be any of these, or none. It may be used to refer to the nature of the communication technology. Print is a medium, as is video, or audio.*
>
> `- Mayes 1990`

In the same year, another attempt at coherent, delineating or unifying definitions for *medium*, *modality*, *mode*, *modal* and *channel* was made on a CHI'90 workshop by Blattner et al. [BD90] with Mayes attending:

> **Mode**: *has the meaning of "interface state" determining the way user's actions are interpreted by the system to computer scientists and interaction style to psychologists. [...] there is no general agreement as to what constitutes state. [JE89]*

Modality: *of an interaction refers to the sensory system the user is engaging or the spatial or verbal nature of the information.*

Modal: *is ambiguous and may refer to mode or modality. For this reason the term should not be used.*

`- CHI'90 workshop by Blattner et al. 1990`

No agreement was reached to differentiate *modality* and *channel*, suggesting some similarity in the concepts they refer to. If there was a definition for the term *media* it did not find its way into the published workshop proceedings. The concluding notes, unfortunately, still remark that *the terminology used by those in this area is not clearly defined and differences of opinion still exist as to what words mean.*

Three years later, in reference to the CHI'90 workshop by Blattner, Nigay and Coutaz [NC93] attempt to settle the matter with the following set of definitions:

Mode: *refers to a state that determines the way information is interpreted to extract or convey meaning.*

Modality: *refers to the type of communication channel used to convey or acquire information. It also covers the way an idea is expressed or perceived, or the manner an action is performed.*

Modal: *may cover the notion of "modality" as well as that of "mode".*

Multimodality: *is the capacity of the system to communicate with a user along different types of communication channels and to extract and convey meaning automatically.*

`- Nigay and Coutaz 1993`

The focus on the extraction of *meaning* is identified as the key distinction between a multimedia and a multimodal system when they write: *a multimodal system is able to automatically model the content of the information at a high level of abstraction. A multimodal system strives for meaning.* This publication refers to earlier work by Coutaz and Calean from 1991 [Cou91], wherein a similar notion is introduced and the additional terms of "media" and "multimedia system" are delineated, which will become important below for its definition of "media":

Modality: *may be the particular form used for rendering a thought, the way an action is performed.*

Media: *is a technical means which allows written, visual, or sonic information to be communicated among humans.*

Multimedia System: *is a computer system able to acquire, deliver, memorize, and organize written, visual, and sonic information.*

`- Coutaz and Calean 1991`

In 1996, six years after the CHI'90 workshop and, presumably, unsatisfied with the incoherent definitions, Blattner herself published an article in *IEEE Multimedia* [BG96], explicitly identifying the ambiguity of the term *mode* and *modality*, depending on the reference to either computer scientists or perceptual psychology, when she remarks:

Modality: *has a more ambiguous meaning. Computer scientists often use the term mode as a synonym for state; for example, an editor might be in cut-and-paste mode as opposed to input mode. Psychologists, on the other hand, refer to human sensory modalities of vision, hearing, touch, smell and taste. We use this latter sense when talking about human-computer interaction. We can also speak of modes of interaction or interaction styles. For example, menus and natural language are interaction styles or modalities. Language may be both a medium and a modality; in this case, our interpretation of the representation determines what we call it.*

Multimodal interface: *exploits human sensory modalities in human-computer interaction; by implication, therefore, these modalities comprise integral and essential components of the interface language.*

`- Blattner and Glinert 1996`

Even though, the definition of *modality* suggests a direct correspondence to the human senses, the examples given in [BG96] and the actual article do differentiate between various dimensions within these sensory systems capable to convey meaning. This distinction is also central to the definition given by Oviatt in 2003 [Ovi03]:

Multimodal Interfaces: *process two or more combined user input modes — such as speech, pen, touch, manual gestures, gaze, and lip movements — in a coordinated manner with multimedia system output.*

`- Oviatt 2003`

Here, the term *mode* carries hardly any of its original connotations as the "state of a computer interface" implied in the survey of Johnson and Engelbeck [JE89], but is in explicit reference to a concept described e.g. in the work of Pick and Saltzman [PJS78] as the *perceptual mode* of a human. Starting from experimental evidence for two different

cognitive systems to process visual stimuli, one for general orientation and another one for focused manipulative behavior, they generalize a perceptual mode as:

> **(Perceptual) Mode**: *is implied: (1) when one type of information rather than another is extracted from a given pattern of stimulation; and (2) when a specific type of information is extracted from very different patterns of stimulation. It is defined in terms of the information extracted by the receiver [...] the engaging of a general mechanism, and perhaps initiation of complex behavior for the guidance of perception.*
>
> - Pick and Saltzman 1978

The perceptual mode would set the *direction of perception* of a human to *process and extract specific information for different functions*; the manner in which a stimulus is interpreted as the cognitive subsystem employed. As an example and concretization with regard to user interfaces, imagine a stimulus of the human eardrum: The selection of a modality is the semi-conscious classification with regard to the kind of audio received: does it carry any information at all, is it to be ignored, interpreted as a sound indicating some event, an utterance of words, maybe music or is it even a mixture of those things? To be applicable in the definitions of Blattner or Oviatt, this notion would need to be transferred into the *perceptual mode of a computer* as a selection and processing of those aspects in a given stimulus that carry information relevant to an interaction. This process of the *extraction of meaning* is also in line with the definition of modality by Nigay and Coutaz [NC93] from above. With communication among humans, we expect to be able to communicate by triggering these perceptual modes in others, as *human to human communication is naturally multimodal* [Cou91], making multimodality a necessary quality for any natural interface.

It is noteworthy, that the definition of multimodal interfaces by Oviatt is apparently asymmetrical: while the ability to trigger different perceptual modes is a required quality for user input, system output has to be delivered via more than one *medium*. Unfortunately, Oviatt did not feel the need to define the term *multimedia* as rigorously as she defined *mode* in reference to perceptual psychology. A definition for *media* and thus *multimedia* as given by Coutaz and Calean [Cou91] above is unsuited as it unnecessarily limits Oviatt's definition by enumerating *media* as *written, visual, and sonic information*. An adaption of the definition given above by Mayes [May90] as *any of these [mode and modality], or none* is even less helpful, as the definition gets too broad to be useful and *mode* is still defined in the notion of computer scientists.

The original wording of the definition of Oviatt, without the explicit reference to *perceptual modes*, was already published in her earlier work [Ovi99], wherein she cites e.g. Neal and Shapiro [NS88] with a book chapter called "Intelligent Multi-media Interface Technology". Unfortunately, even though both terms are being used by Neal and Shapiro to refer qualities of in- and output, no delineating definition is given. However, it is clear that the term cannot be defined pertaining to the sensory apparatus of a human as e.g. text, graphics and animations are all visual representations and consistently referred to as different *media*. Maybe the distinction could be aligned with the delineation of multimedia and multimodal given by Nigay and Coutaz [NC93], wherein a representation and understanding of the actual meaning in a *rendered thought* differentiates a mode from a medium. But most likely, the requirement for multimedia system output in Oviatt's definition was meant as a stronger requirement than just addressing different perceptual modes in a human, as would have been the case with a symmetrical definition, making a distinction as introduced by Nigay and Coutaz unlikely.

However, if we compare Oviatt's definition of multimodal interfaces to the one given by Blattner, we see that this distinction does not seem to be essential for a concise definition. In fact, it is unclear which class of interfaces would be included if we substitute the requirement for "multimedia system output" in Oviatt's definition with a more general requirement to "address multiple perceptual modes". Given the additional semantic problems the former term entails with no obvious discriminative power and the apparent usefulness of *perceptual mode* with a concise meaning grounded in perceptual psychology, we will settle for the following definitions for the scope of this thesis:

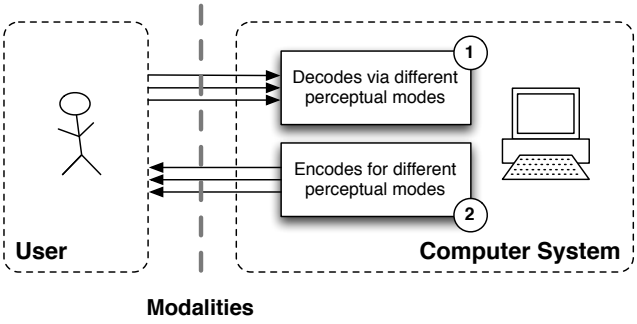


Figure 2.9.: Requirements for an interface to be called "multimodal".

Definition 1 (Modality): *(as a quality of user-interfaces) identifies a perceptual mode of a human or a comparable concept simulated by a computer.*

Definition 2 (Multimodal Interface): *A human-computer interface is multimodal if the computer supports two or more perceptual modes (1) to decode information relevant to the interaction and (2) to encode its response.*

This definition of multimodal interfaces acknowledges the important differentiation of the term *mode* into (i) the notion employed by computer scientists (e.g. [Tes81]) and (ii) the more useful notion employed by psychologists as recognized by Blattner [BG96], and employs the concrete concept of *perceptual mode* [PJS78] introduced into the field of HCI by Oviatt [Ovi03], while avoiding unnecessary ambiguous terms.

We do not postulate any requirement for a human user in this definition as their interactive behavior is *naturally multimodal* [Cou91], with variances in linguistic expressions, intonation and gestures for spoken utterances all carrying semantics. Indeed, it might be argued that a human is incapable to observe or produce interactive behavior without conveying, at least implicit, information on different perceptual modes. Even if we just read a typewritten sentence from another human, the choice of words, the presence / absence of grammatical errors, the general layout, even the circumstances of its arrival will influence our perception of the interaction and the totality of information we extract. With spoken utterances, we will infer e.g. the sender's gender, approximate their age, their general confidence, maybe even assess their health. Indeed, it may be impossible for a human *not* to extract all this additional information. Therefore, the second condition in the definition is rather weak as just about any system output will be subjected to the different perceptual modes of a human. We might even argue that there is, in fact, no unimodal system output, just system output with controlled and uncontrolled modalities. With this distinction, we can yield discriminative power from this second condition by requiring two or more modalities for which the information is controlled, which is meant by "encoding information" to be extracted via perceptual modes in a human.

However, the first condition is the most important criterion for an interface to be called multimodal: We require such a system to extract information contained in a human user's behavior via more than one perceptual mode. There are obviously different levels of *semantic richness* encodable for any given perceptual mode and we might further require that two *non-trivial* modalities are employed to fulfill this condition.

There were many more definitions for the terms *mode*, *medium* and *modality* proposed in the more that 50 years since the terms were first employed in the context of HCI: e.g. Schomaker et al. [SMH95], Bernsen [Ber96], Bordegoni et al. [BFF+97], Rousseau et al. [RBVB06], even the publication of the Extensible MultiModal Annotation Markup Language (EMMA) recommendation [JBB+09] by the W3C Multimodal Interaction Working Group (W3C MMI WG)[1] and any attempt at a unifying definition seems futile by now. Most definitions suffer from similar problems: definitions by enumeration, not discriminative enough to be useful, relying on unresolvable terms, recursiveness or appeals to intuition. After carefully examining the selection presented above, I have to conclude that there is no possible set of well-defined classes between the perceptual modes as the most generic conception to extract meaning via different cognitive subsystems and the class of human senses.

Maybe surprisingly, the above definition of *modality*, based on the concept of *perceptual mode* from psychology is, again, very much aligned with the implied notion of the term in the very early publication from Taylor in 1967 [Tay67]:

> *If you and I are having a face-to-face conversation, we are on-line; if we are communicating by mail, we are off-line. This suggests that there may be degrees of on-lineness. In our face-to-face conversation we are exchanging information through more than one sense modalities.*
>
> - Taylor 1967

2.4.2 Dialogs and Dialog Management

> What makes the man-computer interaction qualitatively different from other types of man-machine interaction is the fact that it may be described, without gross misuse of words, as a conversation. That is to say, the interaction involves a two-way exchange of information in the form of commands, requests, queries, answers to queries and messages of sundry sorts.
>
> Nickerson 1969

[1] http://www.w3.org/2002/mmi/

The term *dialog*, at least in the context of HCI, is much less contended then the previous term of *modality*. As such, we do not need a similarly thorough consideration of the controversies in its historical development. The overall expectation, to align human-computer interaction with the phenomena observed with human to human interaction, most notably spoken communication, can be found throughout the history of computers and was usually entangled with issues of artificial intelligence. Considerations as early as Descartes assumed that all interactive automatons will strive for interaction comparable to a human when he writes:

> *For we can easily understand a machine's being constituted so that it can utter words. [...] But it never happens that it arranges its speech in various ways, in order to reply appropriately to everything that may be said in its presence.*
>
> - Descartes 1637

Later, in 1950, Turing used the ability for a machine to converse coherently as a definition for intelligence via his famous "imitation game". Herein a computer and a human are engaged in a written conversation with a human interrogator whose task is to identify the computer and the human dialog partner [Tur50]. While the prospect of artificial intelligence and this puristic, human-oriented notion of a dialog was ever again an exciting motivating vision for research in HCI and might still be the ultimate goal, the technical confines of early computer systems also lead the development of a more pragmatic, system-oriented conception of the term in what can be distinguished as follows[2]:

- *Human-oriented approaches* started the exploration of the HCI design space from the human perspective and tried to work their way from interaction concepts natural to a human user towards interfaces expressible by the affordances of a computer system. Early work in this regard is found e.g. by McCarthy [McC59] or Raphael [Rap64]. This conception was later famously "mocked" by Weizenbaum in 1966 with his ELIZA system [Wei66, Wei76] and the implied pretense criticized by Searle in 1980 with his "Chinese Room Argument" [Sea80].

- *System-oriented approaches* dropped all pretense to resemble human-to-human interaction in the literal sense and started their exploration from interaction paradigms natural to a computer system. Formal syntax and semantics would establish clear boundaries and expectations for a system's capabilities and a user would have to adapt to the means of interaction offered by the computer. While there was always an attempt to accommodate the human user, the resulting interfaces were reliable and unambiguous first with usability and interaction natural to a human user only as a secondary consideration.

Research in the latter conception did not employ the term *dialog* or *conversation* until the early 1960, when "on-line" interfaces [LC62], e.g. in the form of query-response systems [Sha64], became more prevalent [Nic69]. But even the earliest approaches to programming computer systems can be conceived as a written dialog, wherein a human dialog partner writes formal letters regarding the operations of a central processing unit on memory addresses to a computer system. While this might seem construed, there were already considerations with regard to the "usability" of these early formal languages, e.g. in [Buc58].

The notion of all interaction as a conversation (or dialog) between a human user and a computer continued to play a role, even when written "on-line" communication was complemented by more elaborate means of interaction. When Sutherland published the original description of his "Sketchpad" system in 1964 [Sut64], he employed the notion even prominently in the publication's introduction when he writes:

> *The Sketchpad system makes it possible for a man and a computer to converse rapidly through the medium of line-drawings. Heretofore, most interaction between man and computer has been slowed down by the need to reduce all communication to written statements that can be typed; in the past, we have been writing letters to rather than conferring with our computers.*
>
> - Sutherland 1964

An early authoritative work to establish the term *dialog* as a conception for all things HCI, even via various modalities, is found in the book "Design of man-computer dialogues" by Martin in 1973 [Mar73]. Herein, Martin describes different design considerations when providing interfaces for human users with varying levels of expertise.

By the time of Martin's book, the requirement for an explicit subsystem to bridge application logic to the user interface was already well established with respective approaches dating back as early as Newman's "Network Definition Language" to describe graphical user interfaces as state transition diagrams in 1968 [New68]. A later survey from Maquire [Mag82] attempts to organize the wealth of recommendations for the design of dialogs proposed or implied by various authors since 1966 into 16 *areas of considerations*, providing a more thorough overview of contemporary

[2] Martin made a similar differentiation as designing systems from the *outside in* and *inside out*, respectively [Mar73].

research. Maquire concludes his survey with a call to develop a *framework for the presentation of dialog design information.*

Coincidentally, it was the same year that Kasik coined the term User-Interface Management System (UIMS) to refer to a software component with the responsibility to *decouple physical interaction handling from logical function performance* [Kas82]. In his publication, Kasik introduced the "Tiger Interactive Command and Control Language" as an *application independent dialog specification language* and a respective run-time interpreter to describe graphical user interfaces. In hindsight, this was an important contribution as it allowed to identify and classify *dialog management techniques* [Hix90] as different conceptualizations to express *dialog models* and spurred interest in the development of respective reference models.

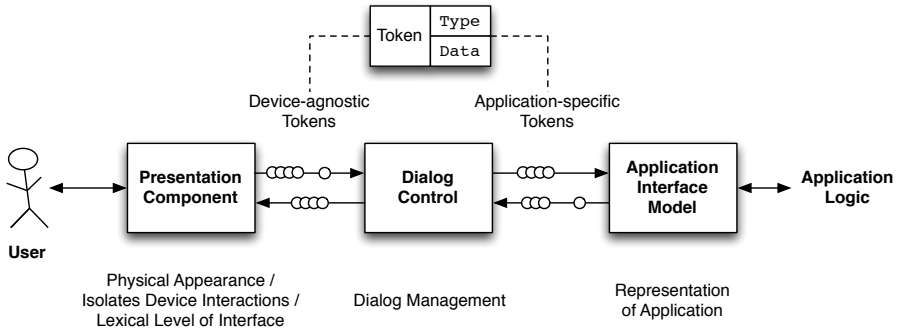


Figure 2.10.: The Seeheim reference model for a user interface management system [PtH85, Gre85].

One early reference model for UIMSs and an important focal point for future research in the field is found in the Seeheim model (figure 2.10). Herein, a UIMS is separated into three components with different responsibilities (from [Gre85]):

- The **Presentation Component** *is responsible for screen management, information display, input devices, interaction techniques and lexical feedback.*
- The **Dialog Control** *manages the dialog between the user and the application. This component converts the stream of input tokens originating in the presentation component into a structure representing the commands and operands intended by the user. This structure is then converted into a sequence of input tokens sent to the application interface model in order to execute the command. Similarly, the output tokens sent by the application interface model are interpreted by dialog control and a sequence of output tokens for the presentation component is generated.*
- The **Application Interface Model** *is the user interface's view of the application. It contains descriptions of all the application's data structures and routines that are accessible to the user interface.*

The original conceptualizations of the Seeheim model were widely adopted in other reference models of the time, but came under pressure as being too simplistic [tH91]. An attempt at a unifying reference model for a UIMS runtime along with a set of design criteria for such models was proposed in 1991 at the "CHI'91 UIMS Tool Developer's Workshop" [BFL$^+$92] as the "Arch" reference model and its generalized variant, the "Slinky" metamodel (figure 2.11).

The responsibilities of the different components are given as follows (from [BFL$^+$92]):

- The **Domain-Specific Component** *controls, manipulates and retrieves domain data and performs other domain-related functions.*
- The **Domain-Adaptor Component** is *a mediation component between the Dialog and the Domain-Specific Components. Domain-related tasks required for human operation of the system, but not available in the Domain-Specific Component, are implemented here. The Domain-Adaptor Component triggers domain-initiated dialog tasks, reorganizes domain data (e.g., collects data items in a list), and detects and reports semantic errors.*
- The **Dialog Component** *has [the] responsibility for task-level sequencing, both for the user and for the portion of the application domain sequencing that depends upon the user; for providing multiple view consistency; and for mapping back and forth between domain-specific formalisms and user-interface-specific formalisms.*
- The **Presentation Component** is *a mediation, or buffer, component between the Dialog and the Interaction Toolkit Components that provides a set of toolkit-independent objects for use by the Dialog Component [...]. Decisions about the representation of media objects are made in the Presentation Component.*

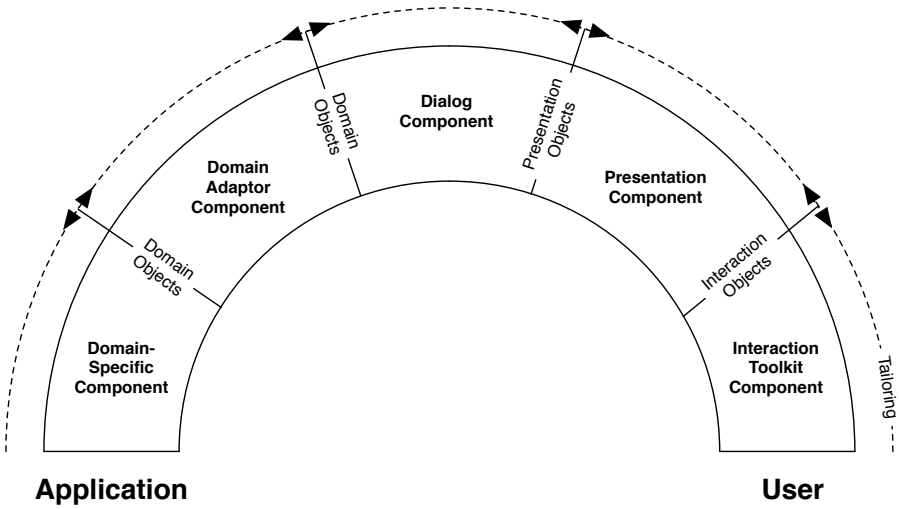


Figure 2.11.: The "Arch" reference model or, with tailoring, the "Slinky" metamodel for a UIMS (from [BFL+92]).

- The **Interaction Toolkit Component** *implements the physical interaction with the end-user (via hardware and software).*

These components are very much aligned with the components from the earlier Seeheim model with only an additional adaptation layer between the application and the dialog manager and a more differentiated presentation component, suggesting conceptual stability and consensus among researchers at the time. However, while the Seeheim model only spoke of in- and output tokens with a general type field and arbitrary data attached, the Slinky model does go into more detail with regard to the information contained in the tokens passed between the components. It is important to note, that neither reference model specifies any modality-specific conception in the presentation component(s), though the term used at the time was still *media* (from [BFL+92]): *"The medium used in the presentation or event generation is not defined"*. And, indeed, most elements of later reference models for multimodal dialog systems can be traced back to these early reference models for UIMSs, most notably the "W3C Multimodal Interaction Framework" [LRR03] as we will see in chapter 3.

By this time at the latest, the concept of a *dialog* and the related issues of *dialog management* and *dialog models* were well established with the same connotations found today. As such, the various attempts to define these terms differed only slightly with regard to their granularity and perspective. The term's arguably intuitive definition is given by its entry in the Merriam Webster dictionary:

> **Dialog**: *A conversation between two or more persons; also: a similar exchange between a person and something else (as a computer).*
>
> **Conversation**: *An oral exchange of sentiments, observations, opinions, or ideas.*
>
> `- Merriam Webster`

We have seen above, that this general notion is to be adapted in the scope of system-oriented approaches to HCI towards the conceptualizations introduced i.e. via the UIMSs reference models. A respective definition aligned with this notion is given e.g. by Nielson [Nie87] as:

> **Dialog**: *A recursive sequence of inputs and outputs necessary to achieve a goal.*
>
> `- Nielson 1987`

Nielson himself already identified problems with this definition, such that user input can not always be *"chopped up into sets of discrete interactions"*. But this very notion is still central to all established dialog management techniques. The user inputs and system outputs are conceived as turns in a dialog, initiated either by the user or the system respectively. This might seem construed at times, e.g. when the user is dragging an object on the screen. But conceiving these turns to be potentially as short as a few milliseconds and interruptible by either dialog partner allows for the application of the various dialog management techniques we will see in chapter 3 and, thereby, can ensure coherent interaction. A later definition of Bunt [Bun95] does explicitly account for the possibility to deliver communicative behavior in these turns via multiple modalities.

> **Multimodal Dialog**: *[is] a sequence of complex elements of communicative behavior intended to change the dialog context.*
>
> - Bunt 1995

Without elaborating on the term *dialog context*, the definition of Bunt generally fits into our definition of *modality* from above and he does imply a sequentialization of interaction events into turns.

While we initially divided early HCI research into system- and human-oriented approaches and followed the first approach to motivate the term *dialog*, it is remarkable to note that these reference models had a unifying effect as research with the latter approach adopted (or developed) the same conceptualizations. This is evidenced e.g. in a publication of Traum [Tra96], wherein he describes a dialog system with natural language understanding employing software agents in an extended Beliefs, Desires and Intentions (BDI) approach:

> **Dialog Manager**: *is that part of a dialog system that connects the I/O devices and translators (whether they be spoken or typed language, a command language, menu selection, graphical presentation, etc.) to the parts that do the domain task reasoning and performance.*
>
> - Traum 1996

Another definition of a dialog manager's responsibility as seen from the human-oriented approach is found in Rudnicky [Rud99]:

> **Dialog Management**: *provides a coherent overall structure to interaction that extends beyond a single turn [. . .].*
>
> - Rudnicky 1999

With the perspective gained above, we can conclude with the following definitions for the scope of this thesis:

Definition 3 (Multimodal Dialog): *is a sequence of interleaved, communicative events between a human and a computer in which information pertaining to two or more perceptual modes is conveyed.*

Here, turns are concretized into discrete *communicative events* and it is left intentionally undefined at which point in time a turn constitutes such a communicative event. E.g. with automatic speech recognition, the communicative event might only be emitted once the user's spoken utterance is finished or intermediate events with preliminary recognition hypothesis might be generated. We will refer to communicative events originating from the user as *input events* and those originating from the system as *output events*.

Definition 4 (Dialog Manager): *is a software component responsible for maintaining the dialog's state and driving the interaction by mapping relevant user input events onto system responses as output events. Performing these responsibilities is also referred to as dialog management.*

Definition 5 (Dialog Management Technique): *(also Dialog Strategy, or Dialog Management Strategy) is a conceptualization of a dialog for an operationalization in a computer system. It defines the representation of the dialog's state and respective operations to process and generate events relevant to the interaction.*

Definition 6 (Dialog Model): *is a formal description of interaction for a concrete user interface employing the syntax and semantics of a specific dialog management technique.*

A Preliminary Formalization of a Dialog

In order to motivate the subsequent contribution of a formal verification of a dialog's temporal properties, we will provide the following, preliminary formalization of the function performed by a dialog manager:

$$f_{DM} : \texttt{state}_t \times \texttt{in}_i \rightarrow \texttt{state}_{t+1} \times \texttt{out}_j \tag{2.2}$$

That is, a system's response $\texttt{out}_j$ and subsequent dialog state $\texttt{state}_{t+1}$ are a function of the user input $\texttt{in}_i$ and the current dialog state at time t, regardless of the employed dialog management technique. If we adapt this formalism to (i) allow $\texttt{in}_i$ to be not only user input, but any event relevant to the interaction (e.g. user input, any timed event or input from external systems) and (ii) conceive the creation of system output $\texttt{out}_j$ to be a side effect of assuming a dialog state, we can interpret the dialog manager as something like a state transition system:

$$Q := \{(p_1, \ldots, p_n) \mid p_i \in \{0, 1\}, \text{a binary encoding of the dialog state}\} \tag{2.3}$$

$$\Sigma := \{\text{any event relevant to the interaction}\} \tag{2.4}$$

$$f_{DM} : Q \times \Sigma \to Q := \text{The dialog management technique} \tag{2.5}$$

The question about the specific computational model of f_{DM} is untouched in this formalization as we made no assumption about the the size of Q. It might still be anything from a simple Deterministic Finite Automaton (DFA) to a Turing complete formalism. And indeed, this will become an important consideration as we would not be able to formally verify temporal properties of a dialog model if we can embed a Turing machine in the dialog management technique due to the halting problem [Tur36]. In fact, we will see in chapter 6 that our approach requires Q to be finite enumerable and thus f_{DM} to be embeddable in a DFA.

This preliminary formalization does describe the interface to the actual application logic only implicitly. Formally, invoking application logic would either (i) be a side effect of entering a state $q \in Q$ and results from the application delivered as input events triggering transitions just as with input from a user or (ii) we could conceive all of the actual application logic to be modeled in f_{DM}. Pragmatically, the former approach is often preferred as it encapsulates the application logic and allows for different user interfaces with the same application. However, it is perfectly possible to model an actual application with f_{DM} simply by embedding its state and operations. This de-emphasis of the actual application interface can be found throughout many of the reference models for Multimodal Dialog System (MDS) and dialog management techniques we will see in the next chapter.

The previous chapter argued for the necessity, or at least desirability, of multimodal interfaces in pervasive environments and defined the terms *modality* and *dialog*. In this chapter we will explore reference models for Multimodal Dialog Systems (MDS) as platforms to interpret respective dialog models and introduce some established *dialog management techniques* to express such dialog models.

We start by regarding the development of early reference models for MDSs, as successors or extensions of the early Seeheim / Arch model introduced in the previous chapter, and work our way to more modern models. We will place a special focus on the framework proposed by the W3C Multimodal Interaction Working Group (W3C MMI WG) as it describes the scope of the group's activities and establishes the relevance of State Chart eXtensible Markup Language (SCXML) as the dialog management technique chosen for our approach to formal verification of respective dialog models.

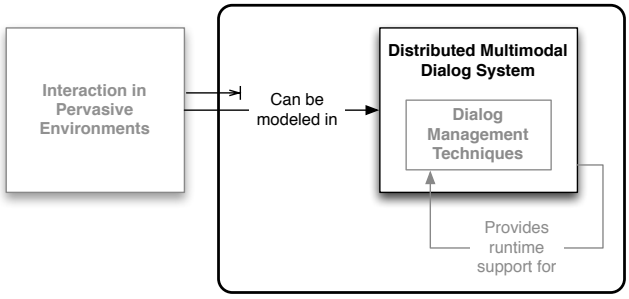


Figure 3.1.: This chapter tackles the first part of the proposition "*Multimodal interaction in pervasive environments can be expressed in a dialog model on a suitable platform.*" from the main argument of this thesis (cf. figure 1.1).

For the main argument of this thesis, the function of this chapter is to establish that there are MDS that, indeed, constitute a suitable platform to express multimodal interaction in pervasive environments and we will, in regard to our approach from figure 1.1, claim for the W3C Multimodal Interaction Framework (W3C MMI framework), and its concrete instantiation in the form of the World Wide Web Consortium (W3C) "Multimodal Architecture and Interfaces" recommendation, to be a such a platform. In fact, this very claim is made already in the W3C MMI WG group charter[1] via their vision of:

> *"Extending the Web to allow multiple modes of interaction (GUI, Speech, Vision, Pen, Gestures, Haptic interfaces, ...). [For] Anyone, Anywhere, Any device, Any time. [...] Accessible through the user's preferred modes of interaction with services that adapt to the device, user and environmental conditions".*

That is, the major ambition of the W3C MMI WG is to establish a suitable platform for multimodal interaction in pervasive environments. Nevertheless, aligning their approach with other reference models established in research will allow us to identify its applicability and eventual short-comings more clearly.

3.1 Reference Models and Implementations

Various researchers presented a plethora of reference models for MDSs over the years. Starting with general considerations for an explicit description of interaction in the scope of User-Interface Management System (UIMS) and the Seeheim / Arch model presented in the introduction, subsequent research refined the conception to account for the issues specific to multimodality, such as sensor and semantic fusion of user-input [LNP+09] and modality selection and coordination for the system's response. Around the year 2000 at the latest, conceptual stability in the general architecture can be observed and standardizations efforts were increased to commoditize multimodal interaction, not unlike Hypertext Markup Language (HTML) for Hypertext and graphical interaction earlier. This process is still ongoing and resulted, so far, in a series of W3C recommendations for which SCXML was proposed as a modality-agnostic dialog model description language.

This selection of MDSs is, by no means, exhaustive. A more complete list up until 2012 is given by Oviatt in [Ovi12] and even more reference models are implied in the various implementations published over the years. However, as we will see, there are striking conceptual similarities and they all can be traced back (more or less directly) to the earliest conceptions of the Seeheim model.

3.1.1 Early Reference Models

We will present only two early reference models. Foremost, to motivate the development from the Seeheim / Arch model to the modern, more generalized reference models introduced later and ultimately the W3C MMI framework.

[1] `http://www.w3.org/2011/03/mmi-charter` (accessed July, 2015)

CUBRICON (1989)

One of the earliest systems that already included most elements found in modern MDSs is the CUBRICON system [NTD+89]. It was developed for military situation assessment as a multimodal map system, allowing a user to point towards a location on a map and pose additional queries via spoken language. It is related to earlier, similar systems such as XTRA and the "Integrated Interface System", both listed in the references of [NTD+89].

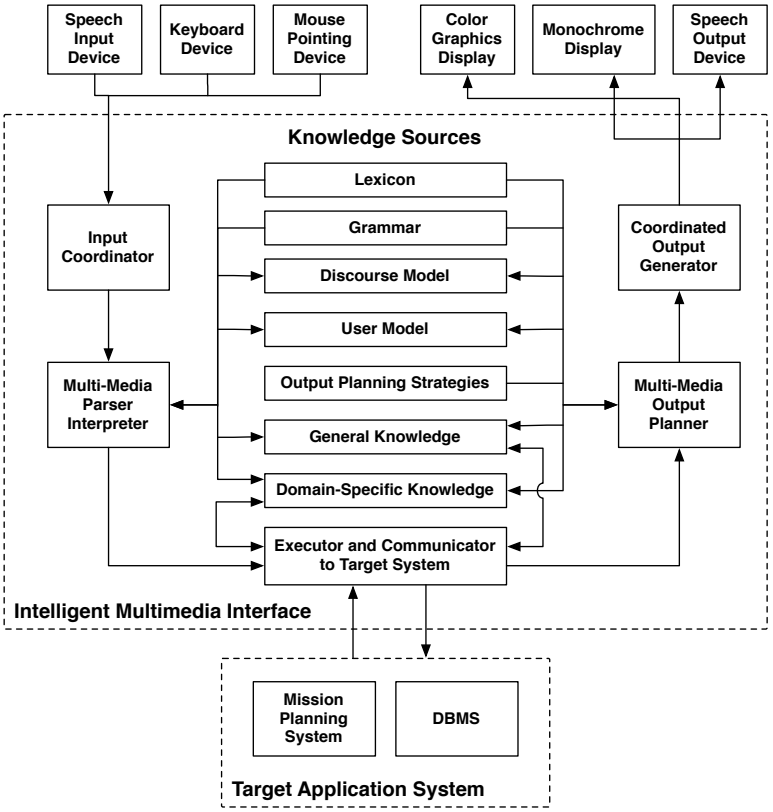


Figure 3.2.: System overview of the CUBRICON dialog system (from [NTD+89]).

The CUBRICON system is noteworthy as it explicitly defines an architecture (figure 3.2) and clearly separates input and output, as opposed to the Arch reference model (figure 2.11) for UIMS. Nevertheless, many responsibilities of components in the Arch reference model can also be found in the CUBRICON architecture:

- The *Domain-Specific Component* and the *Domain Adaptor Component* as the interface into the application logic are deemphasized and subsumed into a general *Executor and Communicator to Target System*.
- The *Dialog Component* is considered in more detail and separated into many different components. Actual dialog management is performed via the *Discourse Model* but many additional knowledge sources are identified.
- Where the Arch model did only specify a *Presentation Component*, CUBRICON distinguishes between (i) interpreting the user input in a processing chain from the devices to a *Multi-Media Parser Interpreter* which feeds refined interaction representations into the dialog components and (ii) a *Multi-Media Output Planner* as the start of a processing chain ending in the output devices.
- The *Interaction Toolkit Component* has no direct equivalence, but we will assume that the *Input Coordinator* and the *Coordinated Output Generator* do employ something similar to a toolkit to control the actual input and output devices.

The clear distinction between the responsibilities for refining user input into modality-agnostic interaction representations on the one side and rendering the system output via selected modalities on the other side, as well as the de-emphasis of the actual interface to the application are found in most later reference models. This may be thought of as a shift of focus from application development towards the issues central to interaction representation as a dialog.

The process of refining sensor data from the various input devices into a compound representation of the user's (implicit or explicit) interaction intent later became known as *multimodal fusion*, whereas the selection of output modalities and the synchronization of respective devices became known as *multimodal fission* [CNS93, NC93, BG96].

PAC-AMODEUS (1991)

The PAC-AMODEUS reference model for multimodal UIMS [NC91] was created as an application of the Presentation-Abstraction-Control (PAC) [Cou87] implementation model for interactive systems in the AMODEUS project. In the PAC model, an interactive application is decomposed into a layered hierarchy of concurrent *interactive objects* (see figure 3.3), each with their own state and interactive behavior. Such an interactive object consists of three components (from [Cou87]):

1. The **Presentation** defines the concrete syntax of the application, i.e. the input and output behavior of the application as perceived by the user.

2. The **Abstraction** corresponds to the semantics of the application. It implements the functions that the application is able to perform. These functions are supposed to result from a thorough task analysis.

3. The **Controller** maintains the mapping and the consistency between the abstract entities (involved in the interaction and implemented in the Abstraction) and their representation to the user. It embodies the boundary between syntax and semantics. It is intended to hold the context of the overall interaction between the user and the application.

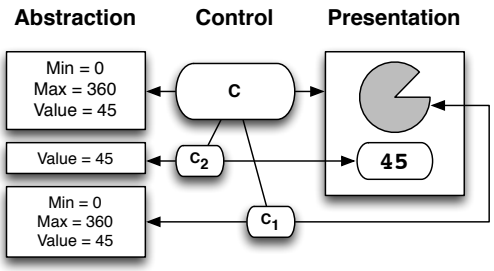


Figure 3.3.: Compound interactive object in the PAC conception (from [Cou87]).

A topmost, compound object represents the interactive application and delegates more specific interaction concerns, i.e. a sub-dialog, an interaction widget or interaction with regard to a specific modality to nested interactive objects. Coutaz herself proposed finite state automatons to maintain the dialog state of an interactive object, but this is ultimately a question of the employed dialog management technique. Using the PAC implementation model, a dialog can be distributed among many concurrent and largely independent entities that cooperate by sending events to each other. This is an important consideration if we are to apply such an architecture for interaction in pervasive environments.

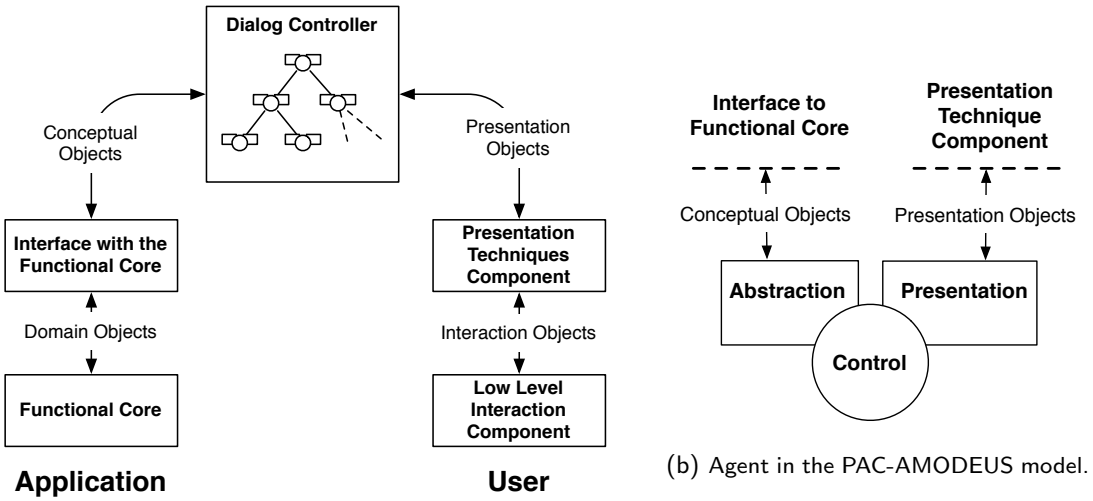
The PAC model was subsequently applied as part of the AMODEUS project to develop a distributed, multiagent reference model for a multimodal UIMS (figure 3.4a). It is in direct reference to the Seeheim and later Arch reference model and of special relevance as its conception of nested interactive objects is closely related to the structure of Modality Components (MC) and Interaction Managers (IM) found in the W3C Multimodal Architecture and Interfaces (W3C MMI architecture).

It can be argued, that the PAC model is similar to the Model-View-Controller (MVC) pattern for organizing interactive systems with the abstraction corresponding to the model, the presentation as the view and the controller as the likewise named component from MVC. In fact, Coutaz herself attempts to delineate PAC from MVC in her discussion about related work in [Cou87]:

- *PAC distinguishes but encapsulates functions and presentation into a single object. A local controller encompasses the boundary between local semantics and local syntax. At the opposite, MVC makes an explicit use of three SmallTalk objects which must maintain their consistency through message passing. In MVC the notion of control is diluted across three related objects, whereas it is explicitly centralized in PAC.*

- *PAC combines input and output behavior into one component, whereas MVC distributes them across two objects. The distribution has the advantage of flexibility (one can change the input syntax without disturbing the component dealing with the output syntax). Unfortunately, it is often the case that, at the fine grained level of the interaction, input events are strongly related to immediate output feedbacks.*

The latter point will become relevant again when we introduce the W3C MMI architecture as part of the W3C MMI framework below: Oftentimes, system output and user input are closely coupled and it is not always practical to separate them into distinct components for fusion and fission.



(a) Components in the PAC-AMODEUS reference model.

(b) Agent in the PAC-AMODEUS model.

Figure 3.4.: The PAC-AMODEUS reference model and the agents constituting its dialog controller (from [NTD+89]).

3.1.2 Generalized Reference Models

Around the year 2000, most reference models proposed for (multimodal) dialog systems were rather similar with only slight variations in their scope, perspective and nomenclature. They all share an approach still traceable to the early Seeheim / Arch model, but most often with user input and system output handling separated into distinct (and sometimes isolated) processing chains and the application interface deemphasized. Three important responsibilities can be identified with virtually every reference model:

- **Multimodal Fusion** (Extract semantics from user input):
  Information regarding a user's behavior is sensed by various sensors and synchronized in the time-domain. Subsequent components for modality specific interpretations attempt to refine this raw sensor data into a suitable representation of the user's interaction intent as input events.

- **Dialog Management** (Act depending on dialog state):
  The representation of the user's interaction intent, as the result of the multimodal fusion, establishes the input tokens for the dialog manager. Depending on the state of the dialog, this component will act upon the user's input and select an eventual (abstract) system response.

- **Multimodal Fission** (Concretize and render system output):
  The abstract system response is concretized with regard to available actuators and suitable modalities. It is, again, synchronized in the time domain and rendered to be perceived by the user.

These three common responsibilities are sometimes accompanied by various knowledge sources or overarching components such as *context*. Occasionally the responsibilities to read sensors and control actuators is also decoupled from fusion and fission respectively. A noteworthy exception are the reference models describing agent-based architectures. These models generally feature a facilitator to coordinate and compound small software agents. Ultimately, though, the responsibilities are comparable to a dialog manager, but distributed among these isolated components that compete for *suitability* at the facilitator in a given situation.

The following paragraphs, rather indiscriminately, list various reference models proposed throughout scientific literature to establish that these three core responsibilities were introduced with virtually every modern reference model. We, explicitly, do not attempt to explain their concrete manifestation in the scope of any given model but merely argue that their persistent reoccurrence strongly warrants some form of standardization in order to reuse individual components with other MDS. This will provide us with the perspective to, ultimately, align and evaluate the W3C MMI framework as a proposed, standardized platform for a MDS in the next chapter.

It is noteworthy, that the conceptualizations for interfacing with the actual application logic are progressively deemphasized to the point of barely being considered at all. As already mentioned when we introduced our preliminary formalization of dialog management at the end of section 2.4.2, the interface to the application logic became implicit

in the responsibilities of dialog management and we will detail possible realizations for the state-chart approach in section 4.2.

Quickset (1997)

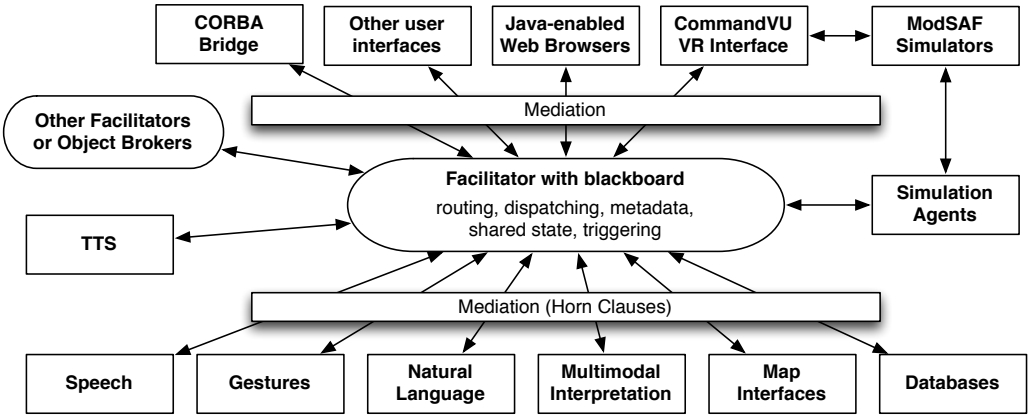


Figure 3.5.: Blackboard as a facilitator to route queries to appropriate agents in the QuickSet system (from [CJM+97b]).

Quickset is an early MDS developed by DARPA funding for the training of military personnel in combat situations. It employs a *distributed, multiagent architecture to integrate not only the various user interface components, but also a collection of distributed applications* [CJM+97a]. It is noteworthy for its application of logical unification to merge *partial meaning representation fragments* from the various modalities into the best *joint representation*. It only implies a reference model as the structure in figure 3.5 does indeed show the actual architecture of the QuickSet system.

As an agent-based system, the components responsible for multimodal fusion, fission and dialog management are not a clearly identifiable as with other reference models, but implied in the selection of agents by the facilitator.

Maybury & Wahlster (1998)

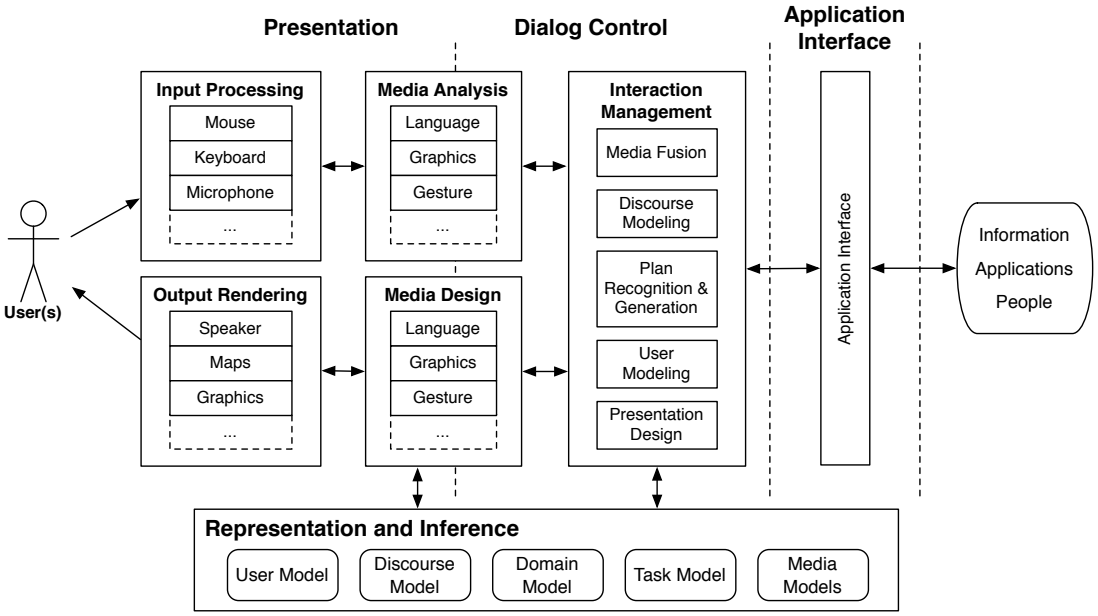


Figure 3.6.: Architecture of Intelligent User Interfaces (from [MW98]).

The reference model of Maybury & Wahlster is presented in the introduction of the book "Readings in Intelligent User Interfaces" [MW98] as a *high-level architecture of intelligent user interfaces* to organize the individual challenges discussed in the remainder of the book. It is not explicitly introduced as a reference model for MDSs but can very

well interpreted as such. It clearly features all the distinct responsibilities with regard to fusion, fission and dialog management.

TrindiKit (2000)

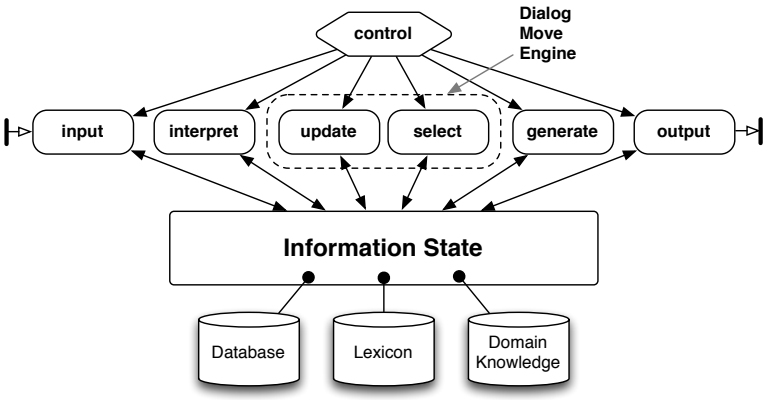


Figure 3.7.: TrindiKit dialog move engine architecture (from [LT00]).

TrindiKit is a general platform for dialog management with a special focus on the *information-state-update* approach, which we will see again when discussing dialog management techniques later in this chapter. While the core of TrindiKit, and its relevance for dialog management, originates in its *dialog move engine*, the other responsibilities of fusion and fission can be seen as peripheral tasks in figure 3.7. It is noteworthy, that TrindKit was originally introduced as a Spoken Dialog System (SDS), and later adapted into a platform for multimodal interfaces.

COMIC (2003)

Figure 3.8.: Simplified architecture of the COMIC system for multimodal dialog management (from [CSW+03]).

The COMIC system was developed as an application of *conversational multimodal interaction with computer*. It is relevant in the presentation of reference models for MDSs for its description of components with the literal responsibilities identified above. Even though only a simplified architectural drawing of the COMIC system, fusion, fission and dialog management are clearly identified in figure 3.8.

SmartKom (2004)

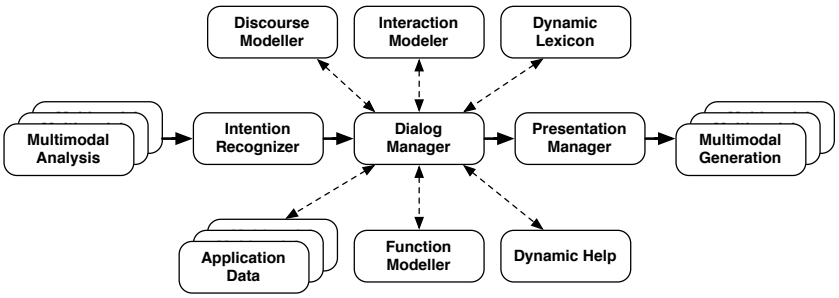


Figure 3.9.: Modules in direct communication with the dialog manager in the SmartKom system (from [Löc04]).

The SmartKom project implemented a *multi-modal task-oriented dialog system* allowing for *a uniform communication model [for] equal treatment of the user, other dialog processing modules, and applications as dialog participant* [Löc04]). The responsibilities with regard to fusion are realized by the *multimodal analysis* component and the *intention recognizer*, fission is implemented in the *presentation manager* and the *multimodal generation* components (see figure 3.9).

Jaspis (2004)

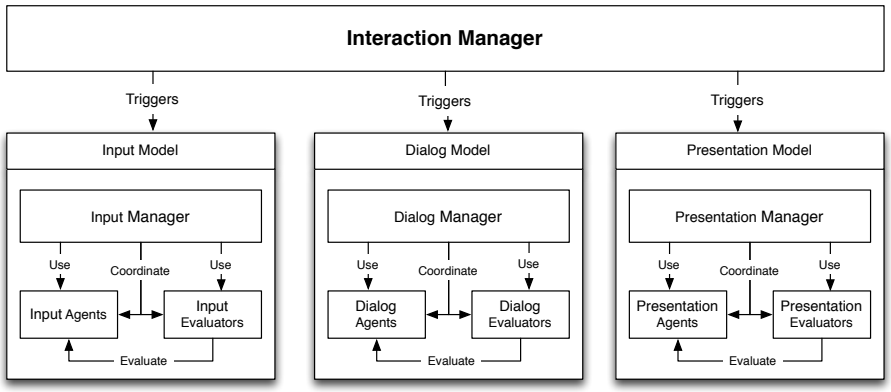


Figure 3.10.: General interaction management architecture in the Jaspis system (from [Tur04]).

Jaspis is another agent-based conception of a dialog system. Initially with a focus on SDS, it was later extended to multimodal interaction in general. Here, the responsibilities with regard to fusion, fission and dialog management are encapsulated in respective agents who are evaluated for their suitability to further the dialog by a central interaction manager (figure 3.10). As with the Quickset system above, the responsibilities are distributed among software agents, however, a clear assignment can be interpreted as these agents are classified into the respective domains of *input*, *dialog* and *presentation*, corresponding to fusion, dialog management and fission.

MIMUS (2007)

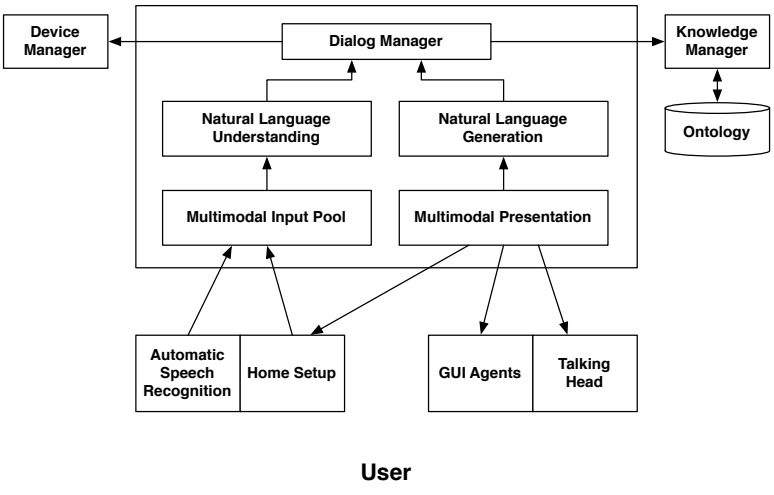


Figure 3.11.: Architecture of the MIMUS multimodal dialog system for the home domain (from [APM07]).

The Mimus project is an implementation of a *multimodal and multilingual dialog system for the in–home scenario, which allows users to control some home devices by voice and/or clicks* [APM07]. It employs an information-state update approach to dialog management (we will introduce later) and describes corresponding components for all the responsibilities of fusion, fission and dialog management.

Oviatt (2012)

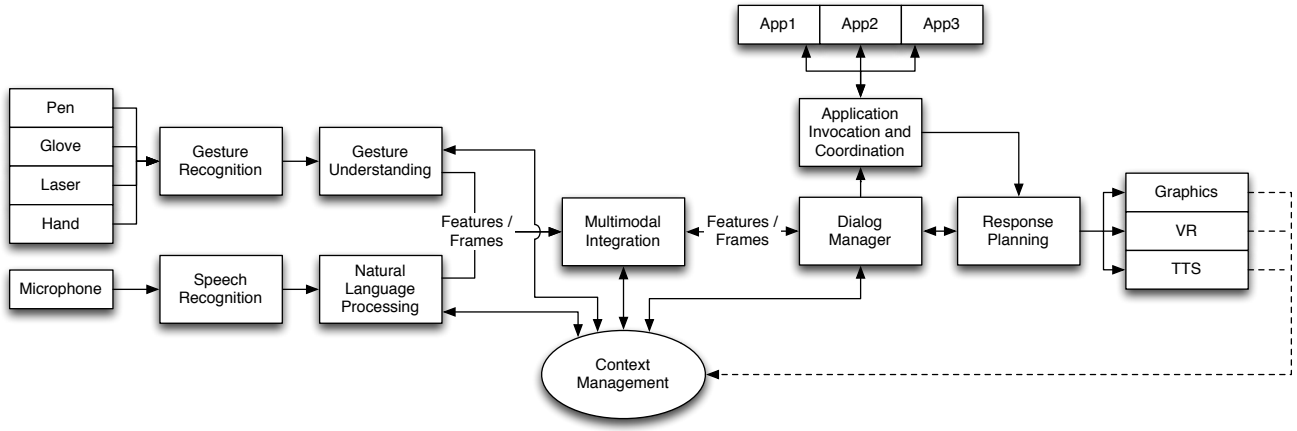


Figure 3.12.: Typical information processing flow in a multimodal architecture designed for speech and gesture (from [Ovi12]).

Finally, this reference model of Oviatt was presented in the book "The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications" [Ovi12]. Even though only a specific instantiation of a MDS, it does contain components for all of the three core responsibilities (figure 3.12).

3.1.2.1 Summary

In one form or another, the three core responsibilities of (i) **multimodal fusion** to refine raw sensor input into an abstract user interaction intent, (ii) **dialog management** to decide upon the system's response given the state of the dialog while considering eventual additional knowledge sources as well as (iii) **multimodal fission** to concretize the system's response with regard to available and suitable output devicescan be found with virtually every reference model. While many researchers, obviously, thought it a good idea to encapsulate these responsibilities, the actual representation of the information passed between the components is quite diverse, inhibiting reuse and forcing each concrete MDS to reimplement the respective functionality. It is this representation of information between the components and a common approach to instantiate and manage their life-cycle, were the W3C attempted to standardize a set of recommendations we will introduce in the next section.

3.1.3 The W3C Multimodal Dialog System

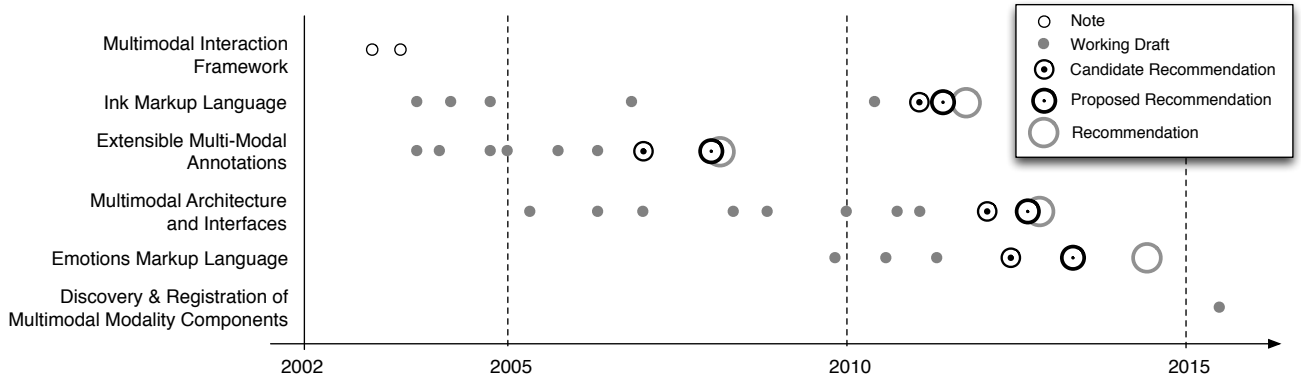


Figure 3.13.: Timeline for publications of W3C standards by the W3C MMI WG.

Since its inception in 2002, the W3C MMI WG published a series of specifications with the ultimate goal to standardize interfaces for the most important components of a MDS. Their earliest publication was the "W3C Multimodal

Interfaces Framework", a W3C note with its latest iteration from 2003. In a sense, it can be thought of as an early declaration of the scope of the working group's ambitions for a standardized MDS. It is explicitly not presented as an actual architecture but merely identifies the major components of a MDS with their respective responsibilities and exemplifies *the markup languages used to describe information required by components and for data flowing among components*. The components are organized into three general sections *input*, *output* and *interaction* with a few cross-cutting concerns, such as *sessions*, *applications* and *environments*. The document itself does not have any normative function, but only provides the context for the subsequent specifications that actually comprise the W3C Multimodal Interaction Framework (see figure 3.13). Not all relevant specifications for the proposed W3C MDS originated with the W3C MMI WG as many established recommendations (e.g. as Voice eXtensible Markup Language (VoiceXML) or HTML) predate the working group's efforts. A more complete list of recommendations applicable for the various responsibilities of a MDS will be given in figure 3.15 below.

The framework describes ten classes of components more closely, three for the the responsibilities input and output each, one logical component as an Interaction Manager (IM) which *coordinates data and manages execution flow* and three more general components for external systems and general concerns (bold in figure 3.14):

- *Input* modality components to refine and abstract user input
  1. **Recognizer** components to digitalize/formalize device sensor data:

     This component *captures natural input from the user and translates the input into a form useful for later processing*. It merely collects sensor data from the various devices and transforms it into a form that can be further processed by the dialog system. The scope of this transformation differs per employed device class and modality. E.g. for spoken input, the result of this component would already be a lattice or an N-best list, whereas e.g. keyboard strokes would be emitted in a more literal representation.
  2. **Interpretation** components to distill semantic information:

     This component distills, unifies and enriches the output of a recognizer component into a representation of its semantics relevant for the task. It would, e.g. transform various spoken affirmative utterances into a simple `yes` or depend on other knowledge sources to interpret a given representation of user input.
  3. **Integration** components to fuse semantics from various sensors:

     In this component, the interpreted representations of the user's input from the various sensors are integrated into a single, coherent interaction intent for the actual dialog manager.
- *Output* modality components to select and concretize system output
  4. **Generation** components to concretize an abstract system response:

     This component will select the modality or modalities to convey the *internal representation* of the system response to the user and concretize the representation with regard to the specifics of the actual runtime platforms for any chosen modality (e.g. XHTML or VoiceXML).
  5. **Styling** components to mold and polish a system response:

     The styling component will enrich the modality specific markup from the generation component with additional information, such as the layout for graphical representation or the persona and prosodic information for spoken output. It corresponds in its responsibilities to the Cascading Style Sheets (CSS) from the HTML specification.
  6. **Rendering** components to direct device actuators:

     This component will interpret the representation from the generation and styling components to actually transform the respective descriptions into phenomena perceptible by the user. For all practical intents and purposes, this component represents the runtime platforms for the various markup languages, e.g. an HTML browser or a VoiceXML platform. But also modalities which are not described via corresponding markup languages from the W3C can be employed, e.g. spatial audio.
- 7. **Interaction managers** to coordinate modality components and general concerns:

     Interaction managers are logical components contained in a *host environment* and coordinate the actual modality components that constitute the input and output of the dialog system. Different such host environments are possible, e.g. interpreter of markup languages itself, such as SVG, Extensible HyperText Markup Language (XHTML), SMIL or dedicated platforms for multimodal interaction. With some host environments, it is possible to nest interaction managers, a concept for which the specification stresses the *russian doll* metaphor, wherein outer interaction managers contain inner interaction managers for more fine-granular responsibilities of interaction, such as *barge-in* with spoken communication or modality specific error correction.

- General components

  8. The **session** component:

     This component ensures the coherence and persistence of the interaction session, especially with regard to a distributed platform. It is not necessarily an explicit component, but might as well be a property of the other components. Its main responsibility is to abstract the issues with regard to resource acquisition and message passing to synchronize the overall interaction.

  9. Information about the **system and environment**:

     This component provides information about the interaction context, e.g. available devices, their capabilities and the user's preferences as well as other general information, e.g. ambient noise level or just about any information relevant to the specific instantiation of the interaction.

  10. The **application functions** interface:

      This component provides the bridge between the MDS and the actual application with its *business logic*. It is only implied in the figures and text, but not described in any detail.

Of these components, all but the three general components can be conceived as Modality Components (MC), which can be nested to form compound MCs. To this effect, an MC would take on the responsibilities of an IM and coordinate nested MCs. In fact, all but the topmost IM constitute compound MCs, similar to the notion of *interactive objects* from the PAC-AMODEUS reference model. The ability to decompose multimodal interaction into compound MCs depends on the host environment.

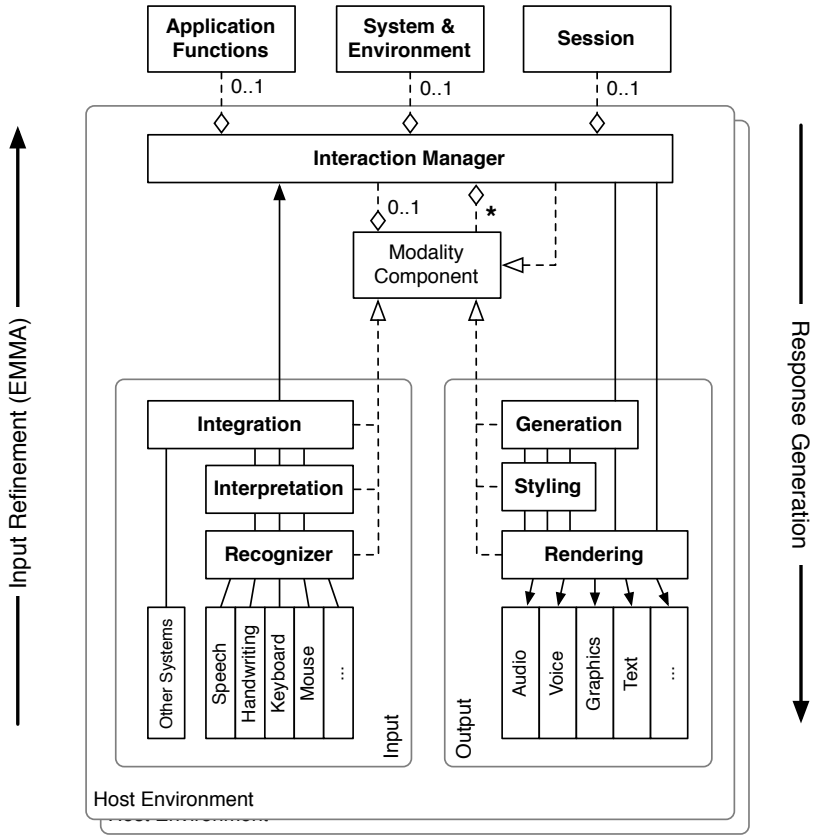


Figure 3.14.: Combined collaboration / class diagram of components in the W3C MMI framework (adapted from [SWRMua13]).

The W3C MMI framework specification insists on not describing an actual architecture but only general considerations for a concrete architecture. As such it lists a set of fundamental properties any actual architecture adhering to the principles of the framework has to follow:

1. *The multimodal architecture contains a subset of the components of the W3C Multimodal Interaction Framework.* A multimedia architecture contains two or more output modes. A multimodal architecture contains two or more input modes.

2. *Components may be partitioned and combined.* The functions within a component may be partitioned into several modules within the architecture, and the functions within two or more components may be combined into a single module within the architecture.

3. *The components are allocated to hardware devices.* If all components are allocated to the same hardware device, the architecture is said to be centralized architecture . For example, a PC containing all of the selected components has a centralized architecture. A client-server architecture consists of two types of devices, several client devices containing many of the input and output components, and the server which contains the remaining components. A distributed architecture consists of multiple types of devices connected by a communication system.

4. *The communication systems are specified.* Designers specify the protocols for exchanging messages among hardware devices.

5. *The dialog model is specified.* Designers specify how modules are invoked and terminated, and how they interpret input to produce output.

When we relate the implied reference model to the earliest Seeheim / Arch model and the subsequent reference model presented above, we can see that there is a traceable mapping of conceptions (see figure 3.15). The W3C MMI framework does deemphasize the issues with regard to the actual application interface as do most other MDS reference models. The responsibilities for the components constituting the input and output of the system are more differentiated and differ from comparable reference models, but this is no fundamental property as *components may be partitioned and combined.*

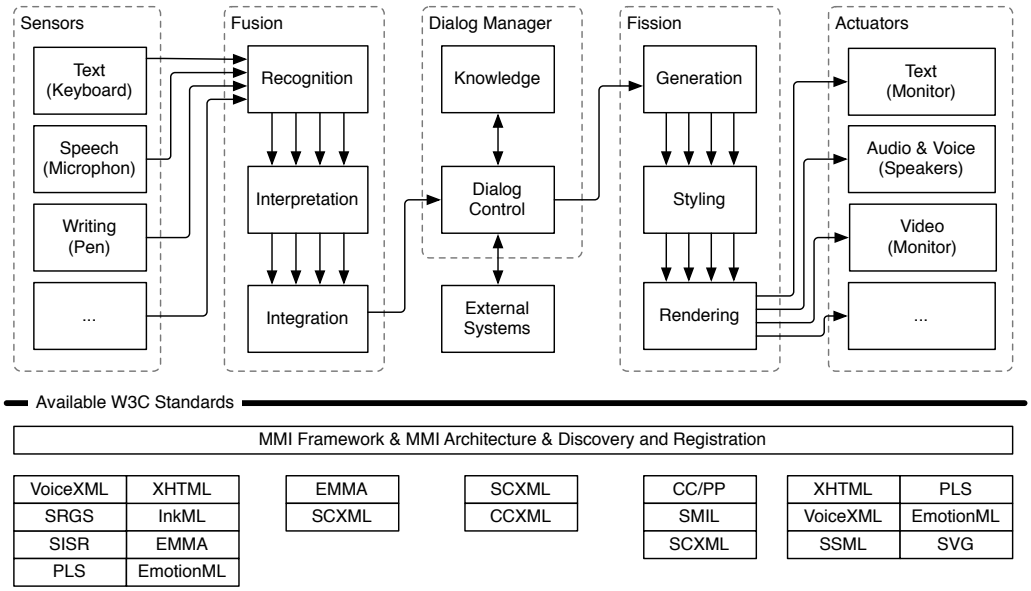


Figure 3.15.: W3C MMI framework as a MDS reference model and available W3C standards.

The W3C Multimodal Architecture

One of the architectures following the principles outlined in the W3C MMI framework is the W3C MMI architecture recommendation [Dah13]. It concretizes the framework into a *loosely coupled architecture [...], which allows for co-resident and distributed implementations.* Foremost, by defining a *life-cycle event interface* and related properties for messages passed between IMs and MCs (table 3.1, figure 3.16). The actual means of delivery as well as the specific encoding of these messages is undefined. In fact, even the actual payload of the messages is intentionally undefined and only the bare minimum with regard to interoperable life-cycle events is specified. The idea is to allow for a maximum of flexibility within individual components while still providing a common basis for potential interoperability. There are five design goals explicitly listed within the specification that help to set its scope and ambitions:

- **Encapsulation:** *The architecture should make no assumptions about the internal implementation of components, which will be treated as black boxes.*

Event	Description
Paired Messages (Initiated by Modality Component)	
NewContextRequest	Request for a new context from the interaction manager.
NewContextResponse	Acknowledgement of success or failure for a NewContextRequest.
Paired Messages (Initiated by Interaction Manager)	
PrepareRequest	Initialize and preload data. Can be sent multiple times prior to starting.
PrepareResponse	If successful, the MC must respond with minimal delay to start requests.
StartRequest	Initiate processing of the document given as part of the request or per URL.
StartResponse	Acknowledgement of success or failure.
PauseRequest	Suspend processing of the current start request.
PauseResponse	Acknowledge suspension.
ResumeRequest	Resume processing of the current start request.
ResumeResponse	Acknowledgement of success or failure.
CancelRequest	Cancel processing of the current start request.
CancelResponse	Acknowledgement of cancellation.
ClearContextRequest	Context no longer needed, free resources and terminate if appropriate.
ClearContextResponse	Acknowledge end of context.
StatusRequest	Keep-alive request.
StatusResponse	Keep-alive response if context is known, undefined otherwise.
Unpaired messages (from Modality Component)	
DoneNotification	Modality component signals end of processing reached.
Unpaired messages (Any Direction)	
ExtensionNotification	Asynchronous event with arbitrary, application-specific data.

Table 3.1.: Life-cycle events in the "W3C Multimodal Architecture and Interfaces" recommendation (from [SWRMua13]).

- **Distribution:** *The architecture should support both distributed and co-hosted implementations.*
- **Extensibility:** *The architecture should facilitate the integration of new modality components. For example, given an existing implementation with voice and graphics components, it should be possible to add a new component [...] without modifying the existing components.*
- **Recursiveness:** *The architecture should allow for nesting, so that an instance of the framework consisting of several components can be packaged up to appear as a single component to a higher-level instance of the architecture.*
- **Modularity:** *The architecture should provide for the separation of data, control, and presentation.*

Within the architecture itself, only four components are described (figure 3.17). Though, without an explicit reference to their function within the W3C MMI framework, most of them have an obvious correspondence:

- The **Interaction Manager** has the same responsibilities as described in the framework above: To manage and synchronize the various MCs. The architecture mandates that *all life-cycle events that the MCs generate must be delivered to the IM* and that *all life-cycle events that are delivered to MCs must be sent by the IM.*

  While this suggests a single, central IM the architecture explicitly allows for *nested MCs*, wherein an MC assumes the responsibilities of a IM for more specific concerns related to the interaction and manages a set of MCs in turn. Again, the metaphor of the *russian doll* is stressed for this composition. In fact, only the topmost IM is a pure IM, all others, nested more deeply in the input or output layers, will need to behave as a MCs for their parent IM and as an IM for the MCs in the processing chain below. This relation is illustrated in figure 3.17: The IM *is a* MC as it will behave as such to its eventual parent IM. Every IM *has a* set of MCs it manages and every MC *has a* single managing IM.

  The recommendation is somewhat ambiguous with regard to the cardinality of the MC *im* relation. Most passages suggest for an MC to have a single managing IM, implying the overall structure of a tree with a top-most IM to handle the most refined, most abstract representation of interaction events, but there is no unambiguous definition.

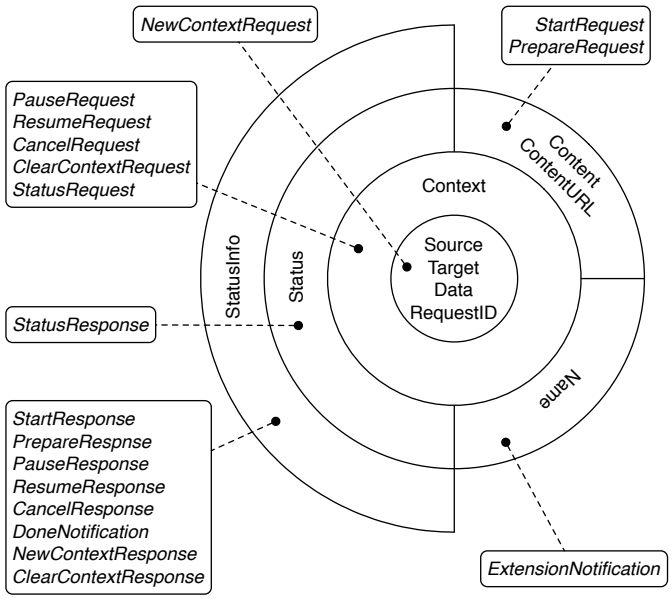


Figure 3.16.: Attributes of the life-cycle events in the W3C Multimodal Architecture and Interfaces recommendation (from [SWRMua13]).

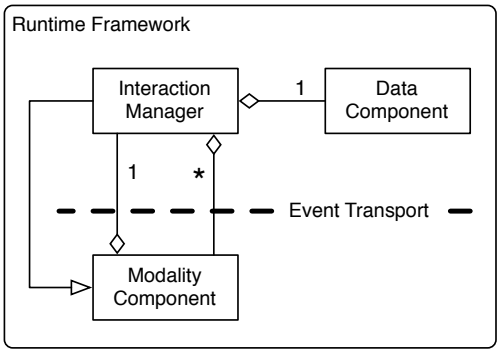


Figure 3.17.: Relationship of the components described in the W3C Multimodal Architecture and Interfaces recommendation.

- The **Data Component** stores *application-level data* and is associated with an IM as its client. A data component may be shared by many IMs but no MC may access it directly. The architecture recommends to distinguish among data components with private data, exclusively associated with a single IM and those containing public data, potentially shared by many IM instances.

  Instances of this component would subsume all the responsibilities of the three general components from the W3C MMI framework, with the possible exception of the *session component*. The session would contain all state relevant to the interaction session as such and, therefore, rather be a property of the *runtime framework* below.

- **Modality Components** in the W3C Multimodal Architecture and Interfaces recommendation are simply the instances controlled by the IMs. There is no attempt to differentiate them any further other than to postulate that they are *responsible for handling all interaction with the user* and need to implement the interface described for MCs.

  Where the W3C MMI framework attempted to classify them into three general classes for input and output each (see above), no such classification is found in the W3C MMI architecture as it is assumed to be *highly domain- and application-specific*. As such, their function within a MDS is solely defined by the events they accept, the payload of messages they emit and with whom they communicate.

- Finally, the **Runtime Framework** provides a platform for all *infrastructure services that are necessary for successful execution of a multimodal application.* With reliable and order-preserving event delivery being explicitly mentioned as a major responsibility.

  The equivalent in the W3C MMI framework would be the *host environment.*

The relevance of the W3C MMI framework and the conforming W3C MMI architecture for this thesis is the fact that both recommendations propose SCXML as a markup language for dialog control in the IM components to realize the responsibilities with regard to multimodal dialog management. And it is SCXML documents, here as multimodal dialog models, for which we will enable formal verification of temporal properties.

Criticism of the W3C Multimodal Architecture

There are two major points of criticism with regard to the suitability of the W3C MMI architecture for actual development of multimodal interfaces:

- **Only MCs can initiate an interaction session:**

  All life-cycle events but the `NewContextRequest` assume an existing context. However, the `NewContextRequest` event can only be initiated by an MC. This implies that it is not possible for an IM to initiate interaction.

  The general assumption might have been that every interaction session originates with a specific MC which registers the user's interaction intent initially. But this is not given, e.g. for multimodal personal assistants in a smart-home environment, where a long-running IM would also need to initiate pro-active interaction with a user. Furthermore, this limitation prevents an IM to instantiate an additional MC, e.g. to propose another means of interaction or simply display additional information on an ambient display. Indeed, all example interaction sessions in the W3C Multimodal Architecture and Interfaces recommendation have the interaction originating with a specific modality.

  Given the obvious desirability of an IM to initiate a context with a new MC, and with the recommendation explicitly stating that *"these events allow the Interaction Manager to invoke Modality Components and receive results from them"*, this really seems like an unfortunate oversight. The pragmatic solution is to allow for IMs to send `NewContextRequest` to MCs and initiate an interaction session.

- **The structure of the resulting MDS is unclear:**

  The architecture does imply a tree like structure with a top-most IM, several nested MC/IM pairs as the branches and simple MC components at the leaf. This, furthermore, implies distinct branches with components responsible for input and output respectively. Which, in turn, isolates the IMs and makes it hard to manage fine-grained interaction control, e.g. lip-synchronous avatars with speech synthesis.

  This problem is also mentioned in the description of an *interactive object* with the PAC-AMODEUS reference model, where it is deemed necessary for an interactive object to, potentially, have access to nested interactive objects.

  Here, the pragmatic solution is to allow an MC to have multiple managing IMs, or at least allow `Extension-Notifications` to be send liberally. This would entail a graph-like structure of components as opposed to the tree.

While these issues are a problem for a strict interpretation of the recommendation, they are not inherent in the conceptualization. Indeed, we will see later that SCXML as a platform to realize the responsibilities of the IM components does not impose such restrictions.

3.2 Multimodal Dialog Management Techniques

In this section, we will present and attempt to classify the different dialog management techniques to perform the responsibilities of the dialog manager component in the various reference models for multimodal dialog systems. It is difficult to find an orthogonal classification for the techniques as many approaches can be mapped onto each other; it is more a difference of the employed conceptualizations being more or less suited for a given kind of dialog. Two prior surveys of dialog management techniques are highly related, one performed by Green in 1986 [Gre86], and a more recent one by Bui in 2006 [Bui06]. While the earlier survey was very much concerned with the techniques' computational model, the latter focusses on more general approaches on how to conceive a dialog. Another early survey of dialog systems was performed by Hix in 1990 [Hix90]. Even if it only identifies four general generations of

systems with no consideration for the different dialog management techniques, it still provides an impression about their development and general requirements towards evermore natural interaction.

Recalling our preliminary formalization for a dialog management technique in equation 2.5, we defined the function performed by a dialog manager component as:

$$f_{DM} : Q \times \Sigma \to Q \quad := \quad \text{The dialog management technique}$$

This allows us to differentiate the various techniques, by considering the following questions: (i) what constitutes a dialog state Q, (ii) how are the input symbols Σ represented and (iii) what semantic features are available to determine the next state in f_{DM}. These questions result in five functional dimensions relevant to dialog management that are suitable to discern the techniques:

- **State Representation:** How does a given dialog management technique represent the state of a dialog in Q. This includes the complete state space as it is needed to perform f_{DM}. We will distinguish the following classes of state representations:

  1. *Discrete*: If f_{DM} is performed exclusively on a concept of state that is explicit and finite, we will consider it to be *discrete*. Every simple finite state transition system is in this class.
  2. *Uncountable*: If f_{DM} utilizes additional information to determine system output and the next state, i.e. the assignment of variables without any special considerations for their limitations, we will call the dialog states Q *uncountable*. This is to be read as *practically uncountable*: All techniques of any use will have to be interpreted in the constraints of a computer system with finite resources, thus we can always, theoretically, embed Q in $\mathbb{N}$ by enumerating all possible assignments in the available memory.
  3. *Enumerable*: This class is very similar to the *uncountable* state representation, however, special care is taken to retain finite enumerability of the variables considered to perform f_{DM}. There are no unlimited data types (e.g. dynamic arrays) and the basic types encourage enumerability, e.g. by dropping floating point numbers.
  4. *Density*: There is no explicit notion of a dialog being in a *single* state, but a probability density function as a vector of probabilities per state.

  The type of state representation for Q in a given technique has a direct consequence for its computational model as its place in the Chomsky hierarchy. However, the discriminative power of the computational model to classify techniques is diminished as most techniques are either able to embed Turing machines (Type-0) or can be expressed in a Deterministic Finite Automaton (DFA) (Type-3). As such, an identification of the computational model is less helpful than one might expect it to be: Theoretically, every dialog management technique can be expressed in SCXML without any data-model, but it is hardly practical to do so.

- **Structured Input:** Many dialog management techniques assume user input $\in \Sigma$ to be expressed as a single literal event. This is oftentimes insufficient if additional information needs to be conveyed. One of the early extensions in this regard was to allow a dialog management technique to operate on semantic frames of related information. This proofed to be a pragmatic prerequisite to enable more elaborate techniques, such as logical resolution below.

  In our classification, we will mark a technique as being able to support structured input, if it is capable to handle compound data types in its representation of user input and dialog states. Here, we might observe that every enumerable compound can be encoded as a single literal, but as argued with the computational model above, we will have to discern between "theoretically possible" and "practical".

- **Logical Resolution:** This dimension refers to the capability of a dialog management technique to infer knowledge from a set of facts and rules and is related to the availability of structured input as a practical prerequisite. Sometimes it is also called, *Grounding & Reasoning* in reference to established communication theories where "mutual knowledge, beliefs, and assumptions" are established as common ground during the dialog.

  There are many specific technologies that would enable logical resolution for a dialog management technique, from *SLD-Resolution* on Horn-Clauses as found e.g. in Prolog to more modern frameworks related to RDF and domain-ontologies. The important aspect is the capability of a system to infer new knowledge from existing facts about the world, those established with a human user during the dialog and respective rules.

- **Distributable:** In general, any system is distributable if we can find isolated subsystems with largely independent state. The state does not need to be completely independent, in fact, if this were the case there would be no

interdependence between the distributed systems at all. We can always synchronize the individual systems' state by sending events between them. But there is a trade-off as the number of events required to do so increases for states that are highly dependent on each other and it is difficult to assess the *distributability* of a system. In this classification we will call a dialog management system distributable if it offers any affordances to support distributed dialog models.

- **Chomsky Rank:** The Chomsky rank identifies a formalism's computational model on the Chomsky hierarchy (table 5.6) as being equivalent to one of the respective reference models. Formalisms with a lower rank (e.g. Turing machines) are always capable to simulate those with a higher rank (e.g. DFAs). As such, every dialog modeling technique with a low rank is theoretically capable to simulate any technique with a higher rank. While this is a very useful dimension to argue for the theoretical computational power of a formalism, it is less useful when applied to the concrete approaches for dialog management. E.g. in section 5.4.2, we proof SCXML (in fact all state-chart variants with *broadcast communication* and *instantaneous states*), even without any data-model, to be Turing complete and, consequentially, capable to simulate any other technique. Yet it is hardly practical to model, e.g. one of the probabilistic approaches in SCXML. Furthermore, most approaches are already Turing complete, diminishing the usefulness of this dimension to classify modern dialog management techniques.

Many of the individual techniques can be simulated in one another and there is hardly any related work evaluating or delineating the suitability of the individual approaches for different classes of dialogs. If we were to evaluate the "naturalness" of a resulting dialog as subjectively perceived by a human user, it can be observed (by the impression from the individual evaluations) that approaches allowing for more of the dimensions proposed above will, in general, perform better as they provide more flexibility and are not experienced as being as rigid as e.g. simple transition networks. On the other hand, many users seem to appreciate the unconditional predictability associated with the deterministic techniques [BD07]. A more formal, comparative evaluation is hard to perform due to the ultimate equivalence of most computational models and different scenarios / tasks employed.

Other classifications for dialogs, especially in the scope of SDS, also introduce the dimension of *initiative* with the specific classes of (i) *user-initiative* when a dialog is predominantly controlled by the user, with the system only reacting (e.g. in command & control dialogs), (ii) *system-initiative* when the system leads the complete interaction and guides a user through a conversation via a series of specific prompts and (iii) *mixed-initiative* for a hybrid of both. However, this classification is only useful for actual dialogs and not necessarily for dialog management techniques as it identifies no specific requirements for a formalization: We can always extend the class of events to other, non-user issued events even with reactive systems.

In the lecture notes of David Traum [Tra08], the different approaches are also classified into *structure-based* and *principle-based*. Where the first class contains all fundamental formalisms, e.g. the various automatons and the latter the more general techniques, e.g. *frame-*, *logic-*, *plan-* and *information-state-based* approaches.

3.2.1 Implicit / Programmatic

Interactive applications do not necessarily need to employ a dedicated dialog model in order to work. Yet, hardly anyone would attempt to read raw sensor data from input devices to realize an interactive system either and virtually all such applications are expressed in the confines of a toolkit or framework for interactive applications, wherein interaction logic is attached to event handlers (or call-backs). The problem with these toolkits is twofold:

- Their conceptualizations limit the design space for interactive applications, i.e. most early concepts of a pointing input device proved insufficient when new technology was introduced: (i) The additional scroll-wheel on a mouse was initially unavailable and many work-arounds, wherein the respective input event would be mapped onto the `PageUp / PageDown` or the cursor keys were required, (ii) the introduction of multitouch displays complicates the semantics of many event handlers, such as `onMouseOver` considerably. As such, a framework is less suited to explore the complete design space of Human-Computer Interaction (HCI) as it already preselects a subset via its very conceptualizations. A variation of this drawback was already identified by Myers in 1991 [Mye91] when he notes:

  > *"The call-backs closely tie the application code to a particular toolkit. Since each toolkit has its own protocol for how the call-backs are called, moving an application from one toolkit to another (e.g., from Motif to Open- Look) can require recoding hundreds of procedures."*

- Their implicit nature can lead to very opaque dialog behavior. With input handlers distributed all over the code-base and different semantics for input event propagation (e.g. capturing, bubbling and canceling) it is sometimes not obvious which event handler is being called in response to an input event. A similar concern is also raised, again, by Myers when arguing for more explicit dialog models:

"The call-backs make maintaining and changing the user interface very difficult. Changing even a small part of an interface often requires rewriting many procedures. Even if a graphical interface builder is used to change the widgets, the call-backs must be hand-edited afterwards if widgets are added, deleted, or modified."

With these short-comings of implicit dialog models identified, the following sections will introduce the various dialog management techniques employing explicit dialog models.

3.2.2 Automatons

This overall class of dialog management techniques contains approaches, wherein the focus is explicitly on discrete states and respective transitions. The interactive system is always in one given state and the set of available transitions from this state to others define the classes of user input the system is currently prepared to handle.

3.2.2.1 Transition Networks

The survey of Green [Gre86] identified a hierarchy of three classes of transition networks with the nice property of their computational models representing the upper three ranks in the Chomsky hierarchy. Even if such transition networks can no longer be considered state of the art with regard to dialog modeling, they provide a very apt basic conceptualization to relate the more modern approaches. Green introduces a helpful example to illustrate the limitations of the three different transition network classes, consecutively extended to motivate each subsequent class: Imagine a user drawing a series of rubber band lines on a graphical canvas. Clicking the button on a pointing device initiates drawing and clicking the button again will establish a line by connecting the initial position of the pointing device with the last position where the second click occurred.

	Basic	**Recursive**	**Annotated**
Chomsky Rank	3	2	1
Structured Input	No	No	Yes
Logical Resolution	No		
Distributed	No		
Dialog State	Discrete		

Table 3.2.: Features of transition networks for dialog management.

Basic Transition Networks

Green proposed a formalization for basic transition networks as follows:

$$
\begin{aligned}
Q &:= \text{a finite set of states} \\
\Sigma &:= \text{a finite set of symbols for user input} \\
P &:= \text{a finite set of system actions} \in Q \\
\delta : Q \times \Sigma \to Q &:= \text{the state transition function} \\
\gamma : Q \to P &:= \text{the labeling function to associate states with system actions} \\
q_0 \in Q &:= \text{the system's initial state} \\
f \subset Q &:= \text{the set of terminal states}
\end{aligned}
$$

This formalism is equivalent to a simple DFA as we can drop γ and P by encoding the system's action in the states Q. As such, the computational model as its rank on the Chomsky hierarchy is 3. The same argument still holds true, if we associate the system's action not with the entering of a state, but to a transition. This will reduce the number of states required for modeling a dialog, while still being expressible as a simple DFA.

Respective dialog managers are rather limited and their practical relevance is reduced to either very basic embedded systems with only discrete input and output or highly abstracted descriptions. To model the rubber band example via this transition network, we can employ a simple transition network as given in figure 3.18.

Recursive Transition Networks

If we abstract substructures in a basic transition network into compound states and allow transitions to enter such compounds recursively, we increase the computational power of the formalism. Recursively entering compound states implicitly establishes a stack structure which can be interpreted as the cellar of a push-down automaton, one of the reference models for type 2 in the Chomsky hierarchy. Thus, there are dialog models expressible with recursive transition networks, that were unavailable for basic transition networks. The example given by Green is to provide dialog support for polylines as connected sequences of rubber bands, wherein pressing of the `backspace` key will remove the last line (figure 3.19).

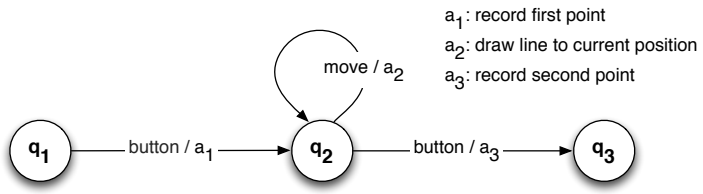


Figure 3.18.: The rubber band example with a simple transition network (adapted from [Gre86]).

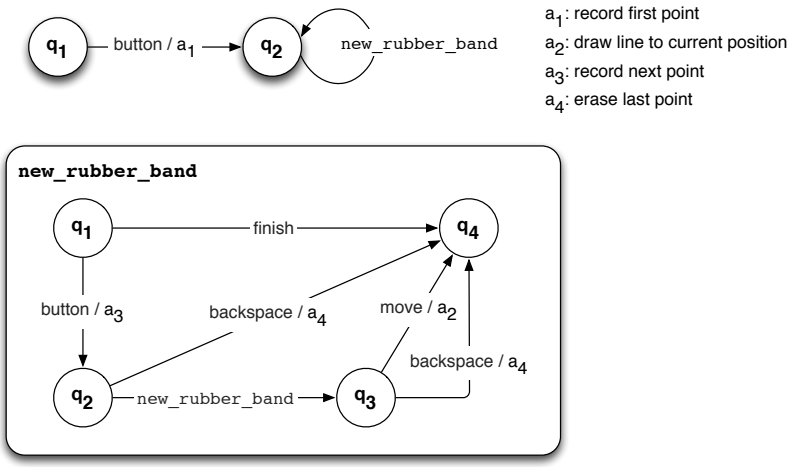


Figure 3.19.: The rubber band example with a recursive transition network (adapted from [Gre86]).

Such an interaction is inexpressible with basic transition networks if we require the number of rubber bands in a polyline to be arbitrary. This is an insightful variation as it only requires a very small adaptation to a DFA to increase its computational expressiveness.

Annotated Transition Networks

In annotated transition networks, a set of registers r_n for arbitrary values is available and transitions can employ functions $f(r_0, ..r_n) \rightarrow (b, r_0, ..r_n)$ to perform computations and guard transitions. Green extends the rubber band example to allow for a global `cancel` operation deleting the complete polyline established so far (figure 3.20). This behavior is inexpressible with recursive transition networks.

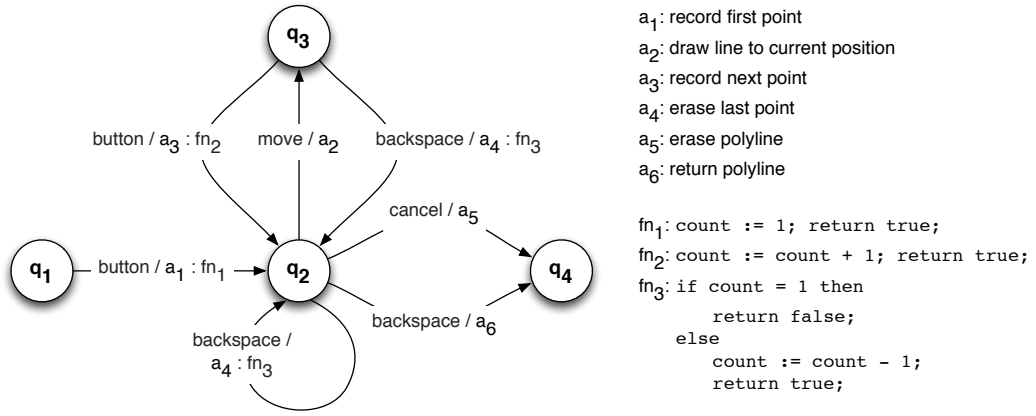


Figure 3.20.: The rubber band example with an annotated transition network (adapted from [Gre86]).

Formally, it is important to limit the number of registers $n < N$ and disallow unlimited data types or restrict f to the subset of primitive recursive functions (e.g. the class of LOOP-Programs [Sch97]), as the formalism would become Turing complete (Type-0 on the Chomsky hierarchy) otherwise.

There are a few short-comings of the transition network formalism, most of which can be remedied by respective extensions:

1. *Unexpected input*: If user input is supplied in a state for which no transition is defined, the system will not react. This can be solved by default transitions for error-reporting or wildcard events.

2. *Verbose descriptions*: Modeling a complete user interface as a set of discrete states tends to get huge and respective graphical representations suffer from "visual overload". This is especially the case with basic transition networks as a state in the transition network has to encode the complete state of the user interface, resulting in an exponential growth. The situation is somewhat relaxed if common substructures can be aggregated into compound states as with recursive transition networks. Annotated transition networks can reduce the overall number of states even more as those differing only in the assignment of a set of variables do not need to be modeled distinctively. However, the greatest relief comes from the representation of a transition network as a state-chart as we will see below.

3. *Rigid dialogs*: The dialogs described as transition networks tend to be inflexible and do not necessarily have any pretense to resemble interaction natural to a human user. On the other hand, one might say they are completely determined and unambiguous.

3.2.2.2 State-Charts

State-charts were originally proposed by Harel in 1987 [Har87] as a more "economical" variation of transition networks, i.e. as a solution to the following four deficiencies he identified in transition networks:

Chomsky Rank	0-3
Structured Input	(No)
Logical Resolution	No
Distributed	(Yes)
Dialog State	Discrete

Table 3.3.: Features of state-charts for dialog management depend on the actual formalization.

1. Transition networks are "flat", there is no notion of depth or hierarchy and as such, no means to refine a system description in a bottom-up or top-down approach.

2. They are uneconomical in their number of transitions as every transition has to be attached to every relevant state. This is especially dire with global behavior, wherein the same set of transitions is attached to *every* state.

3. They are uneconomical in their number of states as a linear growth in the system's description results in an exponential growth for the number of states.

4. They are inherently sequential as there is no syntax to describe concurrent states.

To solve these problems, Harel introduced state-charts as a visual formalism with hierarchically nested states for clustering and refinement of behavior, parallel states (orthogonal regions) for independence and concurrency as well as internal *broadcast communication* by allowing the automaton itself to raise events:

$$\text{state-charts} = \text{state-diagrams} + \text{depth} + \text{orthogonality} + \text{broadcast-communication}$$

In the state-chart formalism, a respective automaton is potentially in a set of states, called its *active configuration*. There are three classes of states: (i) *basic* or *atomic* states B without any child states, (ii) *compound*, *complex*, *super-* or *or-* states C for which exactly one of their children is active if C is active and (iii) *parallel*, *orthogonal* or *and-* states P for which all of their children are active if P is active. The non-basic states, i.e. compound and parallel states are collectively called *composite* states. All states are contained in an implicit, topmost complex root state. It is obvious that there are *illegal* or *inconsistent* configurations, e.g. when only a subset of a parallel state's children or multiple child state's of a complex state are active. Most elements of the graphical syntax are depicted in figure 3.21.

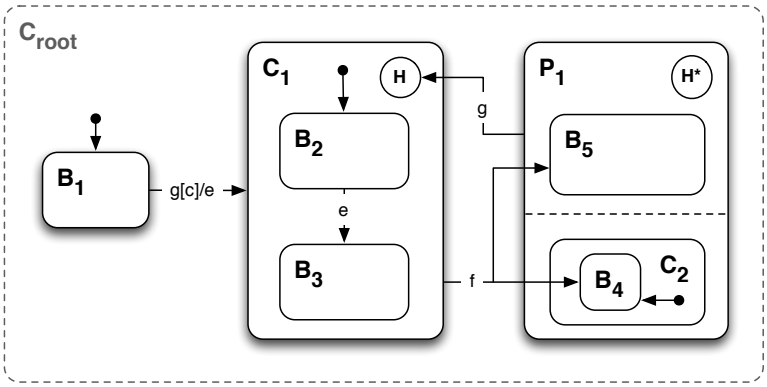


Figure 3.21.: Most of the visual syntactical elements of state-charts.

In addition to the three classes of states, the visual syntax allows for transitions depicted as arrows and labeled with a tuple of `name:event[condition]/action`, with all but the event being optional. A transition may originate from a set of states as its *source* set and lead into a set of *target* states. The set of expressions available for the conditions and the possible actions of transitions are defined in a state-chart's *action language*. Transitions originating in a filled circle denote default states for complex states. The circles with an `H` inside denote *history* states, which will restore the previous active configuration of complex or parallel states upon entry, wherein the optional asterisk differentiates deep $\texttt{H}^*$ from shallow `H` history states (it makes no sense for a parallel state to contain a shallow history, though).

While the visual notation of state-charts appears deceptively simple, a consistent formal semantic with an intuitive interpretation proofed elusive. To develop an intuition about a possible state-chart's formal semantics and variations, let us consider a potential behavior of the state-chart in figure 3.21 when exposed to the following sequence of input events: `(g,f,g)`.

1. When the state-chart is interpreted, it will begin by assuming its initial active configuration as the *default completion* of its root state. Here, the initial configuration contains $\{C_{root}, B_1\}$ as the complex state C_{root} contains a default transition for entering B_1.

2. In this initial configuration, the state-chart is exposed to the input event `g`. If we assume that `c` as a condition on the only active transition holds, the interpreter will transition from B_1 to C_1 and raise the event `e`. In C_1 the default transition into B_2 will cause an intermediate completion for the active configuration of $\{C_{root}, C_1, B_2\}$. Now, depending on the variant of the formal semantics, either `e` will cause the transition from B_1 into B_2 (e.g. SCXML) or is disregarded as not triggering any more transitions in the starting configuration (e.g. with the semantics Pnueli as described below).

3. Regardless of the above variation in semantics, if the next event `f` is received, all active states are considered to find transitions triggered by the event. In our example, C_1 contains a matching transition without any condition or action, which will cause the interpreter to exit C_1 and enter $\{B_4, B_5\}$. This implies to enter all their parent states as well, for a new active configuration of $\{C_{root}, P_1, B_5, C_2, B_4\}$. When exiting C_1, we will need to remember its immediate child state currently active (B_3) as C_1 contains a shallow history state `H`.

4. In the active configuration containing $\{C_{root}, P_1, B_5, C_2, B_4\}$, `g` is received again, causing the transition into the history state of C_1, which will cause a completion of C_1, not via its default transition, but via the last active configuration of C_1 as its immediately active sub-state B_3 we remembered earlier, for a final active configuration of $\{C_{root}, C_1, B_3\}$.

Assuming that a state-chart is in a given configuration, we can informally generalize its semantics as follows:

1. Wait for an input event or set thereof, identify the non-conflicting transitive closure of triggered transitions by iterating a series of *micro-steps*.

2. Assume a new configuration by performing the transitions in a *macro-step*, and goto `1`.

Semantics of Pnueli and Shalev

This behavior largely follows the formalization for a state-chart's semantics of Pnueli and Shalev [PS91]. Harel and Pnueli also did propose an earlier preliminary formalization [HPSS87] which was later found to be deficient,

mainly due to its lack of *global consistency* as transitions were potentially triggered without any external stimulus. In the formalization of Pnueli and Shalev, a state-chart is defined via five general sets of entities (largely adapted from [PS91]):

$$\begin{aligned} \Pi &:= \text{a set of primitive events} \\ S &:= \text{a set of states} \\ \mathcal{T} &:= \text{a set of transitions} \\ r \in S &:= \text{the } \textit{root} \text{ state} \\ V &:= \text{a set of variables} \end{aligned}$$

and three functions, *children*, *type* and *default* to describe structural relations:

$$\begin{aligned} \text{children}(s) &:= \text{the set of immediate substates contained in a compound state s} \\ \text{children}^{+}(s) &:= \bigcup_{i \geq 1} \text{children}^{i}(s), \text{the transitive closure of children}(s) \\ \text{children}^{*}(s) &:= \bigcup_{i \geq 0} \text{children}^{i}(s), \text{the reflexive-transitive closure of children}(s) \\ \text{children}^{*}(X \subseteq S) &:= \bigcup_{s \in X} \text{children}^{*}(s), \text{the reflexive-transitive closure extended to set of states} \end{aligned}$$

where

$$\begin{aligned} \text{children}^{0}(s) &:= \{s\} \\ \text{children}^{1}(s) &:= \text{children}(s) \\ \text{children}^{i+1}(s) &:= \bigcup_{s' \in \text{children}(s)} \text{children}^{i}(s'), i \geq 1 \end{aligned}$$

A state s with $\text{children}(s) = \emptyset$ is called *basic* and *composite* otherwise, with the set *Basic* containing all the former states. If $s_2 \in \text{children}(s_1)$ it is said that s_2 is a *child* of s_1 and s_1 is a *parent* of s_2. For two states s_1, s_2 with $s_2 \in \text{children}^{*}(s_1)$, s_1 is said to be an *ancestor* and s_2 a *descendant*, respectively and both to be *ancestrally related*. Note that s is always ancestrally related to itself. If $s_2 \in \text{children}^{+}(s_1)$, s_1 is said to be a *strict ancestor* and s_2 a *strict descendent*. The tree structure implies that there is always a root $r \in S$ with $\forall s \in S, r \notin \text{children}(s)$ which is called the state-chart's *root*.

$$\text{type}(s) := S \to \{and, or, undefined\}, \text{a partial function to assign one of the three types to a state}$$

With $\text{type}(r) = or$ and $\text{type}(s) = \text{undefined}$ iff $s \in Basic$.

$$\text{default}(s) := S \to S, \text{the default child state for } or \text{ states}$$

With the intention that entering s with $\text{type}(s) = or$ will also, automatically, enter $\text{default}(s)$. For a complete formal semantics, several additional sets and relations are required:

Orthogonal States and Sets of States

- The *Least Common Ancestor* (LCA) for a set of states $X \subseteq S$, denoted as $\text{LCA}(X)$ is the state x such that:

  $$X \subseteq \text{children}^{*}(x), \text{as the smallest such set: } \exists! x' : \left|\text{children}^{*}(x')\right| < \left|\text{children}^{*}(x)\right|$$

  This is the most deeply nested state that is ancestrally related to all $s \in X$, or the first common state on all the children's paths towards the root state r via the parent relation.

- Two states x and y are called *orthogonal* ($x \perp y$) is they are not ancestrally related and their $\text{LCA}(\{x, y\})$ is of type *and*. This basically identifies states that reside in different orthogonal regions.

- A set of states $X \subset S$ is called orthogonal if, for every $x, y \in X$, either $x = y$ or $x \perp y$ as an extension of the preceding concept to sets of states.
- A set of states $X \subset S$ is called *consistent* if, for every $x, y \in X$, either x and y are ancestrally related or $x \perp y$. This excludes sets of states wherein two non-ancestrally related children of a single *or* superstate are contained.
- A consistent set $X \subset S$ is called *maximally consistent* if, for every state $s \in S - X$, $X \cup \{s\}$ is not consistent. Such a maximal consistent set is called a *configuration* of the state-chart and represent its overall state at a given time.
- If X is a consistent set, its *default completion*, denoted by completion(X), is the configuration Y such that, if $X \cap \text{children}(s) = \emptyset, s \in X$ then default$(s) \in Y$. This essentially completes a consistent set of states into a maximal consistent set by adding default states.
- The *initial configuration* X_0 is the default completion of a state-chart's root r.

Transitions

- The source(t) of a transition $t \in \mathcal{T}$ is the set of states where a transition originates as the origins of the transition's arrow in Harel's visual formalism. It is required for source(t) to be an orthogonal set.
- Symmetrically, the target(t) of a transition $t \in \mathcal{T}$ describes the destination states of a transition and is required to be orthogonal as well.
- The arena(t) of a transition $t \in \mathcal{T}$ is an *or* state which contains *all the changes caused by a transition*. Usually this is the deepest *or* state that contains source$(t) \cup$ target(t), but if a transition leaves such an *or* state just to reenter it subsequently, the next higher *or* state constitutes its arena. There is always such a state as ultimately the topmost state r is an *or* state containing all other states.
- The trigger(t) of a transition $t \in \mathcal{T}$ consists of a set of literals $(l_1, ..l_n)$ with l_i being either a primitive event $e \in \Pi$ or a negation thereof.
- Given a set of events $E \subseteq \Pi$, a transition is *triggered* by E if $\exists e \in E : e \in \text{trigger}(t) \vee \nexists e \in E : \neg e \in \text{trigger}(t)$. The latter expression with the negated event literal was originally introduced to enable expressing priorities among transitions.
- Consequentially, triggered(E) identifies the set of transitions which are triggered by a set of primitive events $E \subseteq \Pi$.
- A transition may have an associated *action set*, denoted as actions(t), identifying a set of primitive events $g_1, ..g_n \in \Pi$ raised by the transition when it is taken. The combination of a transition's triggering events and its action set is denoted as $t : l_1, .., l_n / g_1, .., g_m$.
- A transition's action set can be extended to sets of transitions $T \subseteq \mathcal{T}$ by defining:

$$\text{generated}(T) \quad := \quad \bigcup_{t \in T} \text{actions}(t)$$

- Two transitions t_1 and t_2 are called *consistent* if arena$(t_1) \perp$ arena(t_2) and *in conflict* otherwise.
- A set of transitions $T \subseteq \mathcal{T}$ is called a *consistent set* if all constituting transitions are pairwise consistent.

With these sets and relations in place, Pnueli continues to describe the central *enabling function* $\text{En} : (T, C, I) \to T$ with C a configuration, $I \subseteq \Pi$, to ultimately identify the set of transitions $T \subseteq \mathcal{T}$ taken by a state-chart in configuration C in response to a set of primitive external events I as an *admissible step*:

$$\text{En}(T) \quad = \quad \text{relevant}(C) \cap \text{consistent}(T) \cap \text{triggered}(I \cup \text{generated}(T))$$

This function assumes, that some initial set $T \subseteq \mathcal{T}$ was already selected, with C and I fixed, and iteratively refines T until it converges. Here, relevant(C) is the set of transitions with source states in the current configuration. From this set of transitions, the consistent subset is generated by removing transitions with conflicting arenas. Finally, only transitions that are triggered either by the external input events or the generated events are considered. A single

iteration of the $\mathrm{En}(T)$ function is commonly called a *micro-step*, whereas the sequence of *micro-steps* until a new stable configuration is reached is called a *macro-step*. Pnueli gives two definitions to arrive at a set $T = En(T)$ containing the transitions that constitute an admissible (macro-) step of a state-chart in response to a set of external events: an operational and a declarative definition. We will only present Pneuli's operational definition (algorithm 1) as equivalence is shown in [PS91].

Input : $I \in \Pi$ a set of input symbols, $C \subseteq S$ a configuration
Output: A set of transitions T

```
1  T ← ∅;
2  while true do
3  |  if T = En(T) then
4  |  |  return T;
5  |  end
6  |  if T ⊂ En(T) then
7  |  |  T ← T ∪ non-deterministic t ∈ En(T) − T;
8  |  else
9  |  |  return failure;
10 |  end
11 end
```

Algorithm 1: Identifying the transitions of a state-chart in an admissible (macro-) step.

With this set of transitions T in an admissible step for a given set of inputs, we can define the follow-up configuration of a state-machine in configuration C as:

$$\mathrm{nextconfig}(C,T) \quad := \quad \mathrm{completion}((C - \bigcup_{t \in T} \mathrm{children}^*(\mathrm{arena}(t))) \cup \bigcup_{t \in T} \mathrm{target}(t))$$

The inner term will cause the removal of all states from C that are in an $\mathrm{arena}(t), t \in T$ of an enabled transition and subsequently add all states that are in a transitions $\mathrm{target}(t), t \in T$. Most likely, this will result in a partial configuration that is not maximally consistent and only its default completion constitutes the new configuration of the state-chart. There are a few special peculiarities with this formalization[2]:

- It is non-deterministic due to the random selection of $t \in \mathrm{En}(T) - T$ in algorithm 1. This is not a problem, if the state-chart itself is deterministic, i.e. no two conflicting transitions can ever be triggered at the same time when computing $\mathrm{En}(T)$.
- The transitive closure of enabled transitions disregards the order of events generated by transitions and represents them in a set.
- There is no distinction between external events and internal raised via a transition's action.

This formalization is subsequently extended to account for the more elaborate features described by Harel (e.g. variables and parts of the action language), but the general approach remains the same.

Possible Variations in State-Chart Semantics

The sheer amount of formalisms required for a complete specification of a state-chart's semantics is surprising, considering that state-charts started out as an extension to the simple formalism of state-machines. The formalization of Pnueli described above is in reference to the earlier formalization of Harel and Pnueli [HPSS87], which did not exhibit the behavior of *global consistency* insofar that transitions could be triggered spontaneously, which was deemed non-intuitive for system modelers [PS91]. Indeed, this is only one behavior in a series of variations for the formal specification of a state-chart's semantics. A more complete account of 19 possible variations as features or assumptions given or missing in a formal specification is listed by van der Beeck in [vdB94]:

1. The *Perfect Synchrony Hypothesis* assumes that the system is infinitely faster than its environment, i.e. transitions take zero time and all events are processed instantaneously.
2. *Self-Triggering and Causality*: Depending on the formal semantics, transitions may be triggered spontaneously, i.e. two transitions `{t1:a/b, t2:b/a}` residing in different orthogonal regions might trigger themselves as the perfect synchrony hypothesis postulates for them to occur at the same time. Here, the property of *causality* is invalidated, which requires every transition to be caused by a preceding transition or ultimately by an external event.

[2] The original formalization of Harel did include a syntax and semantics for an action language to be employed in condition and actions of transitions, which is adopted by Pnueli but dropped here for brevity.

3. *Negated Trigger Events* are required to be able to resolve non-determinism of state-charts and are a consequence of allowing set of events to trigger transitions. Consider the following state-chart:

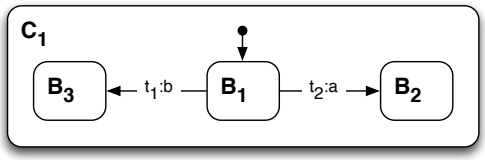


Figure 3.22.: Non-deterministic state-charts.

If the input set $I \subseteq \Pi$ contains a and b and $C : \{B_1\}$, there is no possibility to transform this state-chart for a deterministic behavior. If negated input events are allowed, one of the transitions can be conjugated with the negation of the respective other trigger, e.g. $t_1 : a \wedge \neg b$. This essentially allows to express priorities.

4. *Effect of a Transition Execution is Contradictory to its Cause*: This variation deals with the question, whether a transition $t : \neg e/e$ can ever be triggered as t depends on the absence of an event it generates itself.

5. *Inter-Level Transitions* are available if transitions are allowed to connect two states with a different parent state.

6. *State References* allow a condition to refer to the set of active states, i.e. via an `in(state)` predicate.

7. *Compositional Semantics, Self-Termination*: For a definition of formal semantics, it is desirable for a formalism to be *compositional*, i.e. the definition of a compound's semantics does only rely on its subcomponents. This is usually not possible if inter-level transitions, state-references or the history mechanism are allowed. Self-termination can be employed to compositional formal semantics of inter-level transitions by moving the respective transition to the topmost relevant state and formally terminating a containing compound state.

8. *Denotational Versus Operational Semantics* describes two different kinds of formalizations. In the former, meaning is assigned to individual program phrases and not only the complete program. This implies compositionality and has benefits when proving behavior of the semantics.

9. When *Instantaneous States* are allowed, the computation of the transitive closure of enabled transitions takes states added to the configuration via triggered transitions into account.

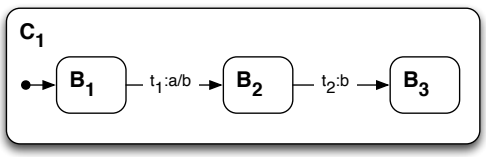


Figure 3.23.: State-chart with a potential instantaneous state $\mathtt{B}_2$.

These states may even potentially be entered and exited in the same macro-step. In the formalization of Pnueli above, $\mathtt{B}_3$ would not be entered in response to `a` in $\mathtt{B}_1$ as the configuration is fixed when computing $\mathrm{En}(T)$. This variation was already encountered in the initial example when we considered to enter $\mathtt{B}_3$ via the event generated from $g[c]/e$.

It is important to note that by allowing instantaneous states in a formal semantics for state-charts, the calculation of the transitive closure of enabled transitions in potentially infinite as new transitions might generate new events which in turn trigger new transitions. It is this aspect of the SCXML semantics, that allows us to embed a Turing machine in section 5.4.2.

10. *Durability of Events*: If a state-chart formalism allows for instantaneous states, the question whether assuming a new configuration containing such a state will consume the events.

11. *Parallel Execution of Transitions* is given, if all enabled transitions in a macro-step are performed at the same time. For an actual implementation which allows actions attached to transitions, this implies that the order of transition execution is actually only undefined. The variant is only meaningful, if the set of enabled transitions per macro-step is finite.

12. *Transition Refinement*: The usual approach to refine transitions is to replace a single transition by a sequence of transitions. If we disallow instantaneous states, we cannot refine a transition via this approach as only a single transition per *or*-state can ever be in the enabled set.

13. *Multiple Entered or Exited Instantaneous States.* One approach of preventing potentially endless sequences of micro-steps via ever new enabled transitions with instantaneous states is to disallow instantaneous states to be entered twice.

14. *Infinite Sequence of Transition Executions at an Instant of Time* is a modification of the previous variation, where no state is allowed to be entered twice during a macro-step.

15. *Determinism*: Non-determinism arises when an event in an active configuration triggers two non-orthogonal transitions. Enabling both might lead to an invalid configuration and most formal semantics will non-deterministically choose one over the other. While many formalizations allow for an explicit prioritization of transitions, non-determinism is not always obvious.

16. *Priorities for Transition Execution.* There are different approaches to provide implicit semantics to the priority of a transition and resolve some of the aforementioned non-determinism. We can define the scope of a transition as the nesting level of its source or target states or introduce negated event triggers to discern different set of input events.

17. *Preemptive vs. Non-Preemptive Interrupt.* Transitions originating in an *or*-state itself and can be thought of as an interrupt as they implicitly exit all states contained. If a set of input events were to trigger both, such an interrupt transitions and other transitions internal to the *or*-state, we can interpret them as preemptive if the internal transition is not triggered and non-preemptive if the internal transition is performed prior to the interrupt.

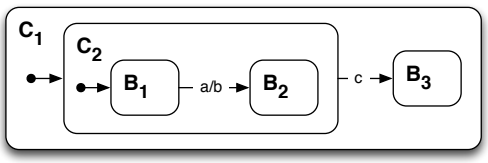


Figure 3.24.: State-chart with interrupt via `c` for an input event set of `{a,c}`.

18. *Distinguishing Internal from External Events.* A formalism may distinguish between internal events as those raised via the action of a transition and those delivered to the system in between macro-steps. If no such differentiation exists, internal events might trigger subsequent macro-steps.

19. *Time Specification, Timeout Events, Timed Transitions.* Formalization might provide affordances to model the progress of time, either via timed events or timed transitions.

Not all combinations of these 19 variations will result in consistent and intuitive semantics and van der Beeck provides a classification of 21 different state-chart formalizations proposed by different authors with regard to these variants in [vdB94].

With all the different formal semantics for state-charts, it is difficult to classify them as a dialog management technique with regard to the five dimensions introduced above. A discussion of the computational model as their rank on the Chomsky hierarchy follows the same considerations as with transition networks, as every transition network is already a state-chart without composite states. As such, we can define *recursive* and *annotated* state-charts correspondingly and arrive at the same Chomsky ranks as with the respective transition networks. Here, the initial description of Harel's state-charts would be equivalent to *annotated* transition networks, as a set of variables and operations to modify them are introduced [HPSS87]. We will see in section 5.4.2 that *intermediate states* and broadcast communication will potentially cause a formalization even to become Turing complete.

A discussion about their affordance of structured input and logical resolution depends on the action language they would offer, which differs among formalizations as well. We will, however, classify them as conditional distributability as parallel states explicitly model concurrency.

3.2.2.3 State-Chart XML

Essentially, SCXML is yet another formal semantics for state-charts. However, its status as a W3C standard and the W3C MMI WG proposal to use it for dialog management in their MDS grants it with a certain normative power from which it derives a distinguished relevance among all possible formalizations. It differs from most other formalizations in that even non-deterministic state-charts have a single, deterministic interpretation. That is, even though it is possible to express non-deterministic state-charts, their interpretation will always yield the exact same behavior when exposed to the same sequence of input events. Their action language (called *data-model* in SCXML terminology) is ultimately undefined as the standard is agnostic in this regard. However, ECMAScript is suggested and many interpreters will offer even more languages (e.g. Lua, Prolog or XPath). For now, we will postpone a more formal introduction of their semantics for chapter 4 and just describe them via their combination of van der Beeck's 19 variants from above.

	NULL	ECMAScript	Lua	Prolog	PROMELA
Chomsky Rank	0	0	0	0	0
Structured Input	No	Yes	Yes	Yes	Yes
Logical Resolution	No	No	No	Yes	No
Distributed	Yes	Yes	Yes	Yes	Yes
Dialog State	Discrete	Uncountable			Enumerable

Table 3.4.: Features of SCXML documents for dialog management depend on the data-model.

```
1  <scxml datamodel="ecmascript">
2    <state id="b1">
3      <transition event="g" cond="c" target="c1">
4        <raise event="e" />
5      </transition>
6    </state>
7    <state id="c1">
8      <history id="c1.h" />
9      <state id="b2">
10       <transition event="e" target="b3" />
11     </state>
12     <state id="b3" />
13     <transition event="f" target="b4 b5" />
14   </state>
15   <parallel id="p1">
16     <history id="p1.h" type="deep" />
17     <state id="b5" />
18     <state id="c2">
19       <state id="b4" />
20     </state>
21   </parallel>
22   <transition event="g" target="c1.h" />
23 </scxml>
```

State b_1 is the default completion for the root state per document-order. Transition on event `g` is conditionalized via the boolean expression `c` and leads to c_1.

Raising event `e` when transitioning from b_1 in line 4 will cause a micro-step into b_3 in SCXML.

Reentering via a deep history pseudo-state will restore a composite state's last configuration recursively.

Listing 3.1: Example state-chart from figure 3.21 expressed in SCXML.

They do follow the perfect synchrony hypothesis **(1)** in that all external input events are synchronized in an *external queue*. Only after a sequence of micro-steps lead to a new, stable configuration is another macro-step performed which will consume an external event. They do prevent self-triggering of transitions and support causality **(2)** in such that all transitions can be traced back to the action of a previous transition and ultimately an external event. However, the transition into the initial configuration as the state-charts default completion may already raise events and cause a sequence of micro-steps. They will not allow for negated trigger events **(3)**, though these can be expressed via a transition's condition if the employed data-model allows to reference the event name and offers respective operations on strings. As such, the problem of generating an event in a transition contradictory to its cause **(4)** is inapplicable. They do allow for inter-level transitions **(5)** and mandate the `in(state)` predicate for state references **(6)**. Consequentially, there are no compositional semantics **(7)** and only an operational description of their behavior **(8)**. They allow for instantaneous states **(9)** and the dequeued event is consumed in a micro-step **(10)**. Transitions are not executed in parallel **(11)**, but document-order from their textual XML representation and nesting depth is employed to impose a strict ordering. They do allow for transition refinement **(12)**, though more options than identified in [vdB94] are available. There is no limit to the number of micro-steps in a macro-step and all states can be entered and left an infinite number of times in a macro-step **(13,14)**. Determinism **(15)** was one of the major design goals to the point that an arbitrary but defined behavior was oftentimes chosen over undefined behavior. Priorities for transitions **(16)**

are inside-out and top-down, that is the most deeply nested transitions take priority with document-order to break ties. They do not support interrupting in the sense of van der Beeck **(17)**, as a more deeply nested transition would always preempt an outer transition. However, one can always introduce a new event to interrupt a complex state, in which case the question about the order of performing transitions becomes void as the inner transition would never be enabled in the first place. They do distinguish internal from external events **(18)**, both in their formal semantics, as well as in the expressions of the employed data-model. And finally, timed events **(19)** are available via a delayed `<send>` to the external queue.

There are a few more noteworthy differences when compared to e.g. the formalization of state-charts from Pnueli:

- Only a single event is ever considered per micro-step, no set of events as with other formalizations. However, the event under consideration may change during a macro-step as transitions can append events on an internal queue and a macro-step only ends when all events enqueued at the internal queue are processed.
- Expressions from the action language can be attached to transitions, as well as to the entering and exiting of states. As such, all these instances may enqueue additional events.
- The document-order from an SCXMLs textual representation carries semantics for the transition priorities and default entry states.
- There are event-less, spontaneous transitions.
- Transitions can be specialized / overridden by providing a conflicting transition in a more deeply nested states. This allows for the specification of general behavior in response to events in the upper states which can be overridden when the system is in a more specific (deeper nested) state.

With regard to their computational model, a similar consideration as with transition networks applies. The SCXML standard mandates an `<invoke>` element to recursively instantiate, among other types, nested SCXML interpreters. As such, every recursive transition network is expressible as an SCXML document. Depending on the data-model, we can also model annotated transition networks, raising their Chomsky rank to 1 as being able to recognize context sensitive languages. Furthermore, instantaneous states and broadcast communication can be abused to embed a Deterministic Queue Automaton (DQA) as an equivalent formalism to universal Turing machines, raising their rank to Turing complete. A respective proof is found in section 5.4.2. The other dimensions in our classification depend on the employed data-model. The SCXML standard only mandates the `NULL` data-model wherein the action language only consists of the `In(state)` predicate. In the `NULL` data-model, no structured input can be expressed as events are just identified by their name and carry no additional data. Ultimately, any suitable language can potentially be employed for data-model of SCXML documents as the standard only requires very few features. E.g. it must be able to evaluate boolean expressions for the transitions' conditions and support assignments. As such, many different languages were proposed:

- `ECMAScript` for a most expressive data-model with familiarity among web-developers. This data-model is especially popular with SCXML implementations that are themselves implemented in ECMAScript, e.g. in the context of an HTML browser.
- `Lua` for resource constrained devices. With a syntax and semantics similar to ECMAScript, but a more economical memory footprint and binary size, Lua is especially popular for embedded devices.
- `Prolog` to allow for grounding and reasoning as it is found in many of the more elaborate dialog management techniques (see below).
- `PROMELA` is a specialty of our `uSCXML`[3] interpreter employed for all simulations and automated transformations in this thesis. It allows to directly transform SCXML documents onto the input language of the SPIN model-checker and is discussed in detail in section 6.3.

3.2.2.4 Petri Nets

[3] `https://github.com/tklab-tud/uscxml`

Chomsky Rank	0-3
Structured Input	No
Logical Resolution	No
Distributed	(Yes)
Dialog State	Discrete

Table 3.5.: Features of Petri nets for dialog management depend on the actual formalization.

Petri Nets are state-transition systems introduced by Carl Adam Petri [Pet62] to model distributed systems. They are an application of elementary networks $EN = (P, T, F)$ with:

$$\begin{aligned} P &= \text{a set of places} \\ T &= \text{a set of transitions} \\ F \subset (P \times T) \cup (T \times P) &= \text{a set of arcs} \end{aligned}$$

such that

$$\begin{aligned} \forall t \in T\ \exists p, q \in P &: (p,t), (t,q) \in F \\ \forall t \in T\ \forall p, q \in P &: (p,t), (t,q) \in F \Rightarrow p \neq q \end{aligned}$$

Places constitute the states in an elementary net and are connected via arcs to and from transitions. The set of connections is defined by the flow-relation F. We can define a *marking* m of such an elementary network as:

$$m : P \to \mathbb{N}_0$$

This can be interpreting as assigning any number of *tokens* to a place in the elementary network. Furthermore, we can associate a cost to an arch with another labelling function V:

$$V : F \to \mathbb{N}$$

With the intuition that transitioning via an arch will require the given amount of tokens. With these two extensions, we can define a Petri net $PN = (P, T, F, V, m_0)$ with:

$$\begin{aligned} (P, T, F) &: \text{an elementary network} \\ V &: F \to \mathbb{N} \text{ a function to assign costs to arcs} \\ m_0 &: P \to \mathbb{N}_0 \text{ an initial marking of places} \end{aligned}$$

The intuitive semantics with regard to transitions is as follows: A transition $t \in T$ is enabled if all states in its input set ${}^\circ t$ have sufficient tokens such that $m(s) \geq V(s,t)$. In which case all tokens are moved through the transition into the output set t°. If we restrict the codomain of both m to $\{0, 1\}$ and that of V onto 1, the formalism is equivalent to a DFA and as such has a Chomsky rank of 3 (see theorem 12 in [RE98]). Many variations for the formalization of Petri nets exist and some will even elevate their computational model to Turing complete [Zai14].

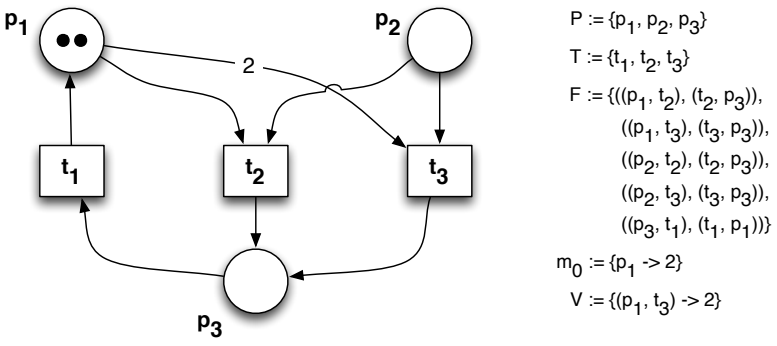


Figure 3.25.: Example of a Petri net.

Despite their rich formalization and well-understood characteristics, they are only rarely suggested as a dialog management technique. Harel identified their lack of a satisfactory approach for hierarchical decomposition as the main drawback when compared to state-charts [Har87] as *"there is only one level of concurrency, and the kind of high-level encompassing events that are applicable at once in many lower-level states (or places) are not naturally specifiable"*. One application of Petri nets for dialog management is given in [vB88], where *nested labelled* Petri nets are introduced and employed to recognize user input as context-free languages.

With regard to the dimensions in our classification, it is unclear how the formalism could support compound data as i.e. structured input or even logical resolution. We will grant distributability, however, as there are approaches to automatically segment Petri nets into largely independent transition systems [vGGSU12].

3.2.2.5 Frame-based

Chomsky Rank	?
Structured Input	Yes
Logical Resolution	(Yes)
Distributed	No
Dialog State	Uncountable

Table 3.6.: Features of usually associated with frame-based approaches.

In contrast to the dialog management techniques introduced above, the class of *frame-based* approaches (sometimes also *form-based*) does not necessarily define an actual formalism, but subsumes a number of techniques and extensions in which it is possible to operate on hierarchically compounded data-structures containing semantically related information, called *frames*. They are oftentimes motivated to provide additional flexibility for the rigid dialog structures imposed by finite-state automatons described above [Bui06] and as a possibility to reduce the number of states (similar to annotated transitions networks).

We can differentiate three classes of approaches depending on the operations on frames that are supported by a management technique:

1. **As the ability to represent frames**:
   A frame, as the representation of knowledge is essentially merely a compound data structure that groups facts related to a specific entity. E.g. a date might be constituted of a year, month, day, hour, minutes and even a timespan and knowledge about a specific date is encapsulated in such a frame. As such, this ability is equivalent to the dimension of *structured input* in our classification. This entails that, e.g. transitions can be conditionalized on specific values in a frame or more generally, that the frames representation is available in a formalism's equivalent to a state-chart's *action language*. The ability to represent frames is a practical prerequisite for any of the operations below.

2. **As the ability to instantiate frames**:
   As a means to instantiate frames, *frame-based* is often understood as the capability to gather related data as the filling of information slots or the instantiation of a template. That is, the dialog manager would require a specific entity (e.g. a date) and the management technique offers the affordances to guide a user to provide it. As an example, VoiceXML is often referred to as supporting frame-based dialog management in this regard. Here, frames are instantiated in *forms* and often embedded in a mixed-initiative[4] sub-dialog. A SDS would pose an open question to a user, such as "What are your travel plans?", with a set of information slots that needs to be filled. The *form interpretation algorithm* of VoiceXML would fill as many slots as possible from the initial user's reply and subsequently query explicitly for all unfilled information slots.

   Another notion in this context is related to the problem of multimodal fusion, wherein the system continuously attempts to recognize the user's interaction intent. E.g. in the scope of SDS this is the problem of natural language understanding. An established approach with regard to multimodal input is the *melting pot* metaphor of Nigay and Coutaz [NC93] wherein the system continuously attempts to map user input onto slots in a two-dimensional frame. Whenever all required slots in the frame were filled within a given time, the system will issue a multimodal user input event towards the dialog manager.

3. **The ability to reason with frames**:
   With frames being interpretable as a formal representation of facts, we can imagine a system to employ logical inference to derive new facts from a set of rules and a-priori knowledge about the world. Pertaining to the mixed-initiative VoiceXML example above, we can imagine that an interpreter might infer the value for some of the slots from existing knowledge and respective inference rules. E.g. if the user were to travel from Boston to Havana, the dialog manager might automatically infer that this is only possible by plane with an additional stop in Mexico City and would not need to query for these slots explicitly. This notion is explicitly included in our classification as the dimension of *logical resolution*.

When one refers to a dialog management technique as being *frame-based*, it only really implies the necessary ability of a formalism to represent and operate on input data as key/value pairs, being the most primitive form of frames. However, an extension to hierarchically compounded data-structures as trees or even graphs seems obvious and of course, these frames need to originate somewhere. If a formalism already provides the means to represent semantically related information, an application of logical inference seems only natural. But these extensions are not necessarily required for a dialog management technique to be called *frame-based*.

Being able to represent and instantiate frames is a rather obvious and very useful extension to the plain state transition automatons and respective examples are numerous. E.g. the notion of collecting and representing semantically related data is central in the original approach of HTML to collect user input in `<form>`s containing graphical and textual `<input>` elements of different types, with their values subsequently submitted to a server. This approach is

[4] The term *mixed-initiative* for these initial open prompts is disputed, but common nomenclature in VoiceXML

also mimicked with VoiceXML forms, where the user (implicitly) instantiates `<field>`s within a `<form>` which is, again, subsequently submitted to a server. In the context of SDS in general and VoiceXML in particular, the ability to instantiate frames is often entangled with the problem of natural language understanding [DQT+09, Nes09].

3.2.3 Rules

This overall class of dialog management techniques contains approaches, wherein a system's interactive behavior is expressed as sequences of `prerequisite` → `effect` rules. The dialog state is usually encoded as the valuation of a set of variables with the prerequisite as a boolean expression on these variables and the effect a new valuation with eventual system output. The language for the expressions in both the prerequisite and the effect of a rule is undefined but usually allows for logical inference. Here, the variable assignments in the dialog state and a-priori knowledge about the world, together with a set of formal rules are used to formally reason about the behavior of the system when exposed to input events.

A popular approach to allow for reasoning and inference in a dialog manager (or any application of logic programming) is to conceive facts, rules and queries as special forms of *Horn clauses*. In general, they represent a disjunction of literal expressions with at most one positive term in first-order logic:

$$u \vee \neg v_1 \vee \neg v_2 \vee \ldots \vee \neg v_n$$

which can be interpreted as the implication

$$v_1 \wedge v_2 \wedge \ldots \wedge v_n \rightarrow u$$

A Horn clause with exactly one positive term (as above) is called a *definite clause* and can express the rules for inference. A Horn clause consisting of a single positive term is called a *fact* and a set thereof can be derived from the variable assignments in the dialog state, along with a representation of user input. A query for the boolean expression in the prerequisite of a rule can be negated and expressed as a Horn clause with no positive terms, called a *goal clause*.

In order to assign a boolean value to a query, the query is resolved via the rules until it contains only facts. Here, a property of Horn clauses is used, wherein the resolvent of two Horn clauses is again a Horn clause. The selection of definite clauses for the resolution of a goal clause is performed via the *SLD-resolution* algorithm, which is usually a depth-first search of all possible resolutions with backtracking.

3.2.3.1 Information State Update

Chomsky Rank	0
Structured Input	Yes
Logical Resolution	Yes
Distributed	No
Dialog State	Uncountable

Table 3.7.: Features of ISU-based dialog management.

The Information State Update (ISU) model for dialog management is a conceptualization to model a dialog as a rule-based system. The basic idea is to represent all interactions as transformations on a central *information state* which encodes the dialog context (also named *conversational score* or *discourse context*) at a given point in the interaction. One of the motivating factors for the introduction of the ISU model was to provide a common framework in which various "dialog theories" could be *formalized, implemented, tested, compared and iteratively refined* [LT00]. Different theories would be formalized in suitable representations of the information state along with respective rules to account for its dynamic behavior.

An application of the ISU model as an interactive application is constituted of three, largely isolated layers: (i) a runtime toolkit to provide functionality common to all dialog theories (cf. figure 3.7 from the previous section), (ii) a formalization of a specific dialog theory and (iii) a domain dependent instantiation of such a dialog theory.

A dialog theory and a specific instantiation in the context of the ISU model is described via five general components (from [LT00, TL03]) pertaining to the upper two layers described above:

- A description of the **informational components** of the theory of dialog modeling, including aspects of common context as well as internal motivating factors (e.g., participants, common ground, linguistic and intentional structure, obligations and commitments, beliefs, intentions, user models, etc.).
- **Formal representations** of the above components (e.g. as lists, sets, typed feature structures, records, Discourse Representation Structures, propositions or modal operators within a logic, etc.).

- A set of **dialog moves** that will trigger the update of the information state. These will generally also be correlated with externally performed actions, such as particular natural language utterances. A complete theory of dialog behavior will also require rules for recognizing and realizing the performance of these moves, e.g. with traditional speech and natural language understanding and generation systems.

- A set of **update rules**, that govern the updating of the information state, given various conditions of the current information state and performed dialog moves, including (in the case of participating in a dialog rather than just monitoring one) a set of selection rules, that license choosing a particular dialog move to perform given conditions of the current information state.

- An **update strategy** for deciding which rule(s) to select at a given point, from the set of applicable ones. This strategy can range from something as simple as "pick the first rule that applies" to more sophisticated arbitration mechanisms, based on game theory, utility theory or statistical methods.

A central part of the common runtime framework is the Dialog Move Engine (DME) (figure 3.26) to execute a dialog theory by iteratively selecting rules whose prerequisite fulfilled to *update the information state based on the observance of moves and select moves to be performed* [TL03].

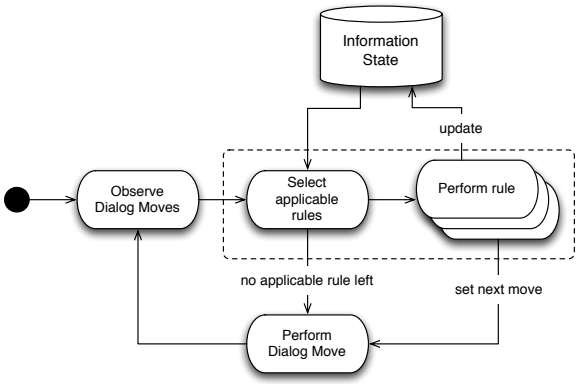


Figure 3.26.: Dialog Move Engine (refined from [TL03]).

Example

In the following, we will concretize the approach by following the instructive example from [TL03]. Given an information space formalized as:

$$\begin{bmatrix} \texttt{PRIVATE}: & \begin{bmatrix} \texttt{BEL}: & \texttt{SET(PROP)} \\ \texttt{AGENDA}: & \texttt{STACK(ACTION)} \end{bmatrix} \\ \texttt{SHARED}: & \begin{bmatrix} \texttt{BEL}: & \texttt{SET(PROP)} \\ \texttt{QUD}: & \texttt{STACK(QUESTION)} \\ \texttt{LM}: & \texttt{MOVE} \end{bmatrix} \end{bmatrix}$$

And a set of update rules as follows:

1. `integrateSysAsk`

$$\texttt{PRE}: \begin{cases} \texttt{val(SHARED.LM, ask(usr, Q))} \\ \texttt{fst(PRIVATE.AGENDA, raise(Q))} \end{cases} \qquad \texttt{EFF}: \begin{cases} \texttt{push(SHARED.QUD, Q)} \\ \texttt{pop(PRIVATE.AGENDA)} \end{cases}$$

"If the last move was a question to the user and it was on the top of our agenda, remove it and remember it as the current question under discussion."

2. `selectAsk`

$$\texttt{PRE}: \quad \texttt{fst(PRIVATE.AGENDA, raise(Q))} \qquad \texttt{EFF}: \quad \texttt{set(NEXT\_MOVE, ask(Q))}$$

"If we want to ask a question as per our agenda, do so."

3. `integrateUserAnswer`

$$\text{PRE}: \begin{cases} \texttt{val}(\texttt{SHARED.LM}, \texttt{answer}(\texttt{usr}, \texttt{A})) \\ \texttt{fst}(\texttt{SHARED.QUD}, \texttt{Q}) \\ \texttt{DOMAIN} :: \texttt{relevant}(\texttt{A}, \texttt{Q}) \\ \texttt{DOMAIN} :: \texttt{reduce}(\texttt{Q}, \texttt{A}, \texttt{P}) \end{cases} \qquad \text{EFF}: \quad \texttt{add}(\texttt{SHARED\_BEL}, \texttt{P})$$

"If the user answered the current question under discussion, add the answer to the shared beliefs."

4. `downdateQUD`

$$\text{PRE}: \begin{cases} \texttt{fst}(\texttt{SHARED.QUD}, \texttt{Q}) \\ \texttt{in}(\texttt{SHARED.BEL}, \texttt{P}) \\ \texttt{DOMAIN} :: \texttt{resolves}(\texttt{P}, \texttt{Q}) \end{cases} \qquad \text{EFF}: \quad \texttt{pop}(\texttt{SHARED.QUD})$$

"If the question under discussion can be inferred from our shared beliefs, remove it."

We will assume an initial valuation of the information state as empty except for a series of items on the agenda, corresponding to the data that needs to be instantiated (i.e. as a frame) for a successful interaction:

```
PRIVATE.AGENDA = <raise(?x.dest_city(x)), raise(?x.depart_city(x)), ...>
```

The DME would identify the prerequisite of update rule 2 (`selectAsk`) as the first to be fulfilled. Executing its effect will raise the question `"Where do you wan to go?"` as the next dialog move and implicitly set `SHARED.LM` as the last dialog move performed to `ask(?x.dest_city(x))`. This will enable the prerequisite of rule 1 (`integrateSysAsk`), causing the DME to pop the action from the agenda and push it as the current question under discussion. At this point, the information state would contain

$$\begin{bmatrix} \text{PRIVATE}: & \begin{bmatrix} \text{BEL}: & \{\} \\ \text{AGENDA}: & < \texttt{raise}(?\texttt{x.depart\_city}(\texttt{x})), \ldots > \end{bmatrix} \\ \text{SHARED}: & \begin{bmatrix} \text{BEL}: & \{\} \\ \text{QUD}: & <?\texttt{x.dest\_city}(\texttt{x}) > \\ \text{LM}: & \texttt{ask}(?\texttt{x.dest\_city}(\texttt{x})) \end{bmatrix} \end{bmatrix}$$

with the system waiting for the user to respond to the question. If the user were to respond e.g. with `Malvern`, the DME would identify rule 3 (`integrateUsrAnswer`) as applicable, as the supplied input answers the question under discussion and update the information state to contain the given destination as a shared belief. This will cause rule 4 (`downdateQUD`) to become applicable and pop the current question under discussion for a final information state of:

$$\begin{bmatrix} \text{PRIVATE}: & \begin{bmatrix} \text{BEL}: & \{\} \\ \text{AGENDA}: & < \texttt{raise}(?\texttt{x.depart\_city}(\texttt{x})), \ldots > \end{bmatrix} \\ \text{SHARED}: & \begin{bmatrix} \text{BEL}: & \{\texttt{dest\_city}(\texttt{malvern})\} \\ \text{QUD}: & <> \\ \text{LM}: & \texttt{answer}(\texttt{malvern}) \end{bmatrix} \end{bmatrix}$$

Several more formalizations of information states and respective dialog theories are given in the work of Traum and Larsson. An essential aspect of this approach is the application of logical inference to minimize the number of dialog moves by inferring facts from the shared belief established between the user and the system as the dialog progresses.

For the initial presentation (and many subsequent refinements) the TrindiKit toolkit as a Prolog program was employed to provide functionality common to all dialog theories in the lowest layer. This allows us to identify the Chomsky rank of the approach to be 0 as it is possible to express the dynamic behavior of a universal Turing machine in Prolog if we assume the availability of an unlimited band or queue.

We will not grant the approach the property of being distributed as logical inference will need access to all the facts established during a dialog. It is conceivable to isolate the facts into different domains and distribute a respective dialog manager, then again this is the essence of the *agent-based* approaches we will introduce below.

3.2.3.2 Agent-based

Chomsky Rank	0
Structured Input	Yes
Logical Resolution	Yes
Distributed	(Yes)
Dialog State	Uncountable

Table 3.8.: Features of agent-based dialog management.

With agent-based approaches to dialog management, the overall interaction is perceived as *a collaborative process between intelligent agents* [Bui06]. Herein, it is not necessarily defined how individual agents perform the responsibilities assigned to them, or even how to distribute responsibilities among agents. As such, it is not so much a formalism for dialog management, but more an approach to distribute the task onto a set of entities that collectively attempt to further the dialog. An agent may employ any of the more concrete dialog management techniques.

A popular conceptualization is to express the mental state of an agent as consisting of Beliefs, Desires and Intentions (BDI) [BIP88]. Herein, an agent has a set of *beliefs* as all the facts it assumes to be true about the state of the world. This set of facts includes all additional facts that can be inferred via the rules available to an agent. The *desires* of an agent formalize the various things an agent would like to become truth eventually. E.g. an agent may have the desire to instantiate a frame via the most concise back and forth of interaction with a user. These desires and the beliefs about the state of the world result in *intentions* the agent would like to perform in order to, ultimately, change the state of the world according to its desires. An extension of the BDI conceptualization, specific to the context of dialog management with multiple cooperating agents, is the introduction of explicit social attitudes as *mutual beliefs*, *shared plans* and *obligations* by Traum [Tra96]. These concepts make certain social conventions explicit, which allows to model interactive behavior more aligned with the expectations of a human user.

Usually, these software agents are managed via some form of central facilitator, a software component which evaluates the individual agents for their applicability in a given situation and, in the case of dialog management, assigns the responsibility to further the dialog state towards some goal. We have already seen this architecture in the context of the QuickSet and Jaspis systems when we introduced the reference models for MDS in section 3.

There are various applications of agent-based dialog management for multimodal interfaces and the technique is still popular. The bulk of contemporary research about agent-based dialog management is concerned with actual applications and specific approaches to organize and valuate the agent's mental state, e.g. [NW05, WHNK05, Nes10].

With regard to our classification, many of the same considerations as with ISU technique for dialog modeling apply. Though, we will grant conditional distributability as there are architectures that employ distributed, collaborative software agents to realize the overall dialog with a user.

3.2.3.3 Plan-based

Chomsky Rank	0
Structured Input	Yes
Logical Resolution	Yes
Distributed	(No)
Dialog State	Uncountable

Table 3.9.: Features of plan-based dialog management.

There are different conceptions about what constitutes a *plan-based* approach to dialog management. Common to most is the notion that a plan is the means to arrive at a goal (e.g. to completely fill a frame) via a series of subgoals (e.g. fill individual information slots). In order to achieve the final goal, a dialog manager will have to decide which sequence of subgoals as a plan is most suited given the current dialog state and eventual additional information.

The problem gets more challenging if we allow for the inference of new information via logical resolution. In this case, one series of subgoals might initially appear longer than another but becomes dramatically shorter as most of the subsequent subgoals can be derived from facts established in prior steps or a-priori knowledge.

To illustrate the latter point, imagine an interactive application which intends to identify a random animal a user selected via a series of questions. Even though, such systems are a classical example for classification and regression trees, we could imagine them to be specified via a set of facts and rules, e.g. "frogs croak" or "birds fly". For a plan-based dialog, the task would be to find the minimal set of definite clauses that will resolve onto a unique solution given the facts established a-priori and as the answers to previous questions. In this regard, a plan-based dialog manager would attempt to find the shortest sequence of interactions that are necessary to arrive at a goal.

In another notion of plan-based dialog management, the goal is initially unknown and it is the task of the dialog manager to recognize the user's goal / intention by observing the given input. This is known as *plan recognition* and exemplified via the example given by Cohen [Coh97]: Imagine a customer asking a butcher *"Where are the steaks you advertised?"*. Here, the customer is not actually interested in the location of those steaks, but implicitly expresses the intention to purchase some of those. Therefore, a plan-based dialog management technique, if it would be able to uncover this hidden goal, would return a response along the lines of *"How many do you want?"*. Different approaches and variations were described in scientific literature. E.g. in [AP80], the authors describe a general framework for collaborating agents, which attempt to detect other agents' goals to identify and help to overcome *obstacles* as "goals in the plan that the other agent cannot achieve (easily) without assistance".

An important concept that, ever again, appears with dialog management in general and plan recognition in particular is to identify and instantiate *speech acts* [Aus62, Sea69, Sea76] as representations of the different functions of

a (spoken) utterance. These speech acts are differentiated into three kind of functions reflecting different senses or dimensions of the "use of a sentence": (i) the *locutionary act* of saying something with a certain sense and reference (meaning), (ii) the *illocutionary act* of establishing a certain force in saying something (e.g. stating, warning or promising) and (iii) the *perlocutionary act* of actually achieving some result with an utterance. Speech acts are popular as a prerequisite for natural language understanding and much research has been conducted in terms of taxonomies and identification (e.g. [CMP90]). They were subsequently abstracted into general *dialogue acts* by Bunt [Bun96] to conceptualize interactive behavior as such, which even culminated in the ISO standard 24617-2[5].

It is difficult to align this technique in the context of our classification as it is, by itself, not a dialog management technique, but rather a supporting approach to help, e.g., with agent-based dialog management. As such, we will grant it with the same classification, with the exception of distributability as it is not immediately obvious how the recognition and execution of a plan could be distributed.

3.2.4 Probabilistic Automatons

The class of probabilistic automatons encompasses all approaches to dialog management wherein the dialog is not conceived to be in a specific state, but in a set of probable states as a probability distribution function or, generally, where probabilistic processes are considered to drive the dialog. Another defining characteristic is the fact that virtually all dialog models expressed via these formalisms are not hand-crafted, but can be learned from a corpus of annotated interactions. They are very much suited to provide dialogs that appear natural to a human user and, in fact, constitute the bulk of contemporary research for dialog management.

3.2.4.1 Markov Decision Processes

A (discrete) Markov Decision Process (MDP) formalizes a stochastic system in a discrete time domain and allows to reason about optimal strategies for its control [FS12]. MDPs are a popular approach for "planning under uncertainty". At every time, the system is in one of several states with a set of available actions. The uncertainty is due to the fact that taking an action in a state will, according to a probability density function, lead to a random successor state and grant an associated reward. If the reward is disregarded and only a single action is available per state MDPs will degrade to simple Markov chains. There are different possible formalizations for such a system, mostly due to the different approaches of granting rewards. In the variation of associating a reward with a non-deterministic transition, a MDP can be formalized as a tuple $MDP = (S, A, T, R)$:

Chomsky Rank	?
Structured Input	(No)
Logical Resolution	No
Distributed	No
Dialog State	Discrete

Table 3.10.: Features of dialog management with MDPs.

$$
\begin{aligned}
S &:= \text{a set of discrete states} \\
A &:= \text{a set of possible actions} \\
T_a(s, s') &:= \text{the probability of action } a \in A \text{ to cause a transition from } s \text{ to } s' \\
R_a(s, s') &:= \text{the reward associated with taking a given transition}
\end{aligned}
$$

The most interesting difference to the formalism of classical finite state automatons is the ability to model an uncertainty as to whether performing an action in a state will actually yield the intended result, i.e. entering a desired subsequent state and the association of rewards with transitions (or states). This allows us to find a solution to the problem illustrated in the example MDP from figure 3.27.

Suppose the automaton is in state s_1. To increase its immediate reward, it might be tempted to take the transition towards s_2 by performing a_2 for the reward of 100. However, there is a large probability, that this action will not actually lead to s_2, but to s_3 and incur a cost of 100 instead. As such the main benefit of the formalism is to provide the means to find an optimal *policy* $\pi(S) \rightarrow A$ as a function that maximizes the total *reward* (or *utility*) for choosing an action in a given state:

$$
\pi(s) := \arg\max_a \left(\sum_{s'} T_a(s, s') \; R_a(s, s') \; \gamma V(s') \right)
$$

[5] `http://www.iso.org/iso/catalogue_detail.htm?csnumber=51967` (accessed September 28th, 2015)

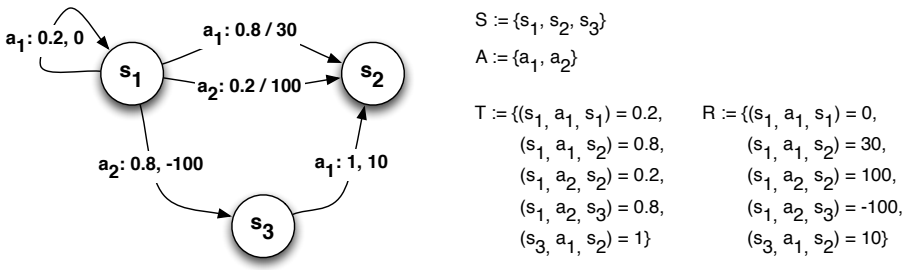


Figure 3.27.: Example for a Markov decision process.

Here, $\gamma \in [0, 1]$ is a discount factor to balance immediate against delayed rewards and $V(s) : S \to \mathbb{R}$ is a recursively defined value function. Intuitively, we can think of $V(s)$ as the value of being in state s, with its value being propagated backwards (with a discount factor γ) from the rewards associated with subsequent transitions.

$$V_{i+1}(s) \quad := \quad \max_a \left(\sum_{s'} T_a(s, s') \ \left(R_a(s, s') + \gamma V_i(s') \right) \right)$$

Initially, $V_i(s)$ is assumed to be zero with only the reward of available transitions contributing to the overall value of a state. In subsequent iterations, V_{i+1} is continuously refined until it converges [Bel57], at which point, the policy π can be established. Such a policy allows to identify, at each state, which action is to be taken to maximize the total reward. After $V(s)$ converges, $\pi(s)$ essentially codifies "hill-climbing" towards ever higher-valued states. There are some variations on the formalism, e.g. where the reward is defined for a being in a state or performing an action in a state and not the actual transitions, but the general approach remains the same.

A popular example is that of planning a path on a grid that contains areas which are dangerous to enter. Imagine a robot navigating such a grid: entering the destination area is associated with a high reward and entering a dangerous area with a negative reward (cost). For every move between areas, there is a chance for the robot to actually end up in an adjacent area, which might be dangerous. The policy π would describe, for each state as an area, where to go next in order to avoid the dangerous and ultimately reach the destination area. Herein, an additional small cost would be incurred for every move to select shorter over longer paths.

Employing MDPs to establish an optimal dialog structure with regard to some cost function was originally proposed by Levin et al. [LPE98]. The idea is to conceive pairs of system output and user input as actions in an MDP that influence the state of the world with regard to the dialog. A probabilistic model, derived from a corpus of previously recorded and annotated interactions, is available to valuate the probability transitions to subsequent states. An application of a MDP to derive a dialog structures in the context of a SDS is given in [You00]. Herein, the responsibility of the dialog manager is conceived to *"change the system from some initial uninformed state to a sufficiently informed state"*. The actions to be performed are, foremost, questions to the user and the reward function incurs a small penalty for each action. Entering a sufficiently informed state is awarded with a huge reward, for an overall policy that *"satisfies the user's requirements while minimizing transaction time"*. To arrive at the probabilities for $T_a(s, s')$, Young defines a joint probability taking into account the task-model, the dialog control strategy, a user's response and the performance of a speech-understanding system. This allows to infer policies that describe dialogs differing in the amount of explicit, implicit and absent confirmations during a spoken interaction, depending on the probabilistic models for each component in the joint probability.

It is important to note that a MDP with a fixed policy can be interpreted as a simple (yet optimized) finite state machine again. As such, the same considerations with regard to our classifications apply.

3.2.4.2 Partially Observable Markov Decision Processes

A Partially Observable Markov Decision Process (POMDP) is an extension of the MDP formalism for a stochastic system in a discrete time domain, where the system can not even determine the state of the world resulting unequivocally. Instead, each action of the system will only cause a potentially ambiguous observation for the system to perceive. As such, the state is no longer discrete, but given as another probability distribution of likely states b in which the world might be, called a *belief state* [KLC98]:

Chomsky Rank	?
Structured Input	(No)
Logical Resolution	No
Distributed	No
Dialog State	Density

Table 3.11.: Features of dialog management with POMDPs.

$$b(s) := \text{Probability of being in state } s, \text{ with } 0 \leq b(s) \leq 1, \sum_{s \in S} b(s) = 1$$

The fundamental problem approachable with the formalism, to identify a policy π to identify the most rewarding sequence of actions per state remains essentially the same, but needs to be adapted for belief states. A POMDP can be formalized by extending the MDP formalization from above as $POMDP = (S, A, T, R, \Omega, O)$:

$$\begin{aligned} (S, A, T, R) &:= \text{a MDP} \\ \Omega &:= \text{a set of observations perceptible from the world} \\ O_a(o, s') &:= \text{the probability of perceiving } o \in \Omega \text{ given a transition to } s' \text{ by performing action } a \end{aligned}$$

Whenever an action $a \in A$ is performed with the system in a given belief state b, an indirect observation $o \in \Omega$ about the state of the world is perceived in turn. These three entities can be used to calculate the subsequent belief state b' with its constituting coefficients $b'(s')$ as beliefs in a given state s' as follows:

$$\begin{aligned} b'(s') &:= P(s' \mid o, a, b) \\ &:= \frac{O_a(s', o) \sum_{s \in S} T_a(s, s')\, b(s)}{P(o \mid a, b)} \end{aligned}$$

In order to find an optimal policy and decide upon an action $a \in A$ for a current the belief state, the transition function needs to be defined, not on discrete states, but probability distributions of states and the reward function adapted accordingly (see [KLC98] for a complete formalization). One major problem with POMDPs is their intractable nature with regard to the value function as it is needed to calculate an optimal policy to decide upon actions in belief states [Lit09]. In fact, it has been shown that an exact solution for an optimal policy with an infinite horizon is impossible to calculate as it contains the halting problem [MHC99]. In order to, nevertheless, operationalize POMDPs, different heuristics and simplifications were introduced to approximate the value function and the resulting policy.

An early description of using POMDPs for dialog management is given in the work of Roy et al. [RPT00]. Wherein a spoken dialog system for an interaction with a nursing robot is modeled as a hand-crafted POMDP:

- The **states** model the various phases an interaction with a user may traverse, e.g. a pending request for the television program or being tasked to go to the kitchen, for a total of 13 states.
- The **actions** are distinguished into two classes:1. 10 available tasks the robot can perform and 2. 10 actions to *maintain* the dialog via clarification or confirmation.
- The **observations** represent 16 different spoken keywords.
- The **state transition function** and the **reward function** encourage the robot to satisfy the user's request and penalize everything else. Here, selecting confirmation and clarification actions is less penalized than not fulfilling the user's request and, wrongfully, moving into another room is more heavily penalized.

Despite this limited domain and finite horizon, the authors were unable to establish an optimal policy for the complete POMDP at the time and resorted to a smaller domain with a more limited action set and state space (only time and weather information).

Many more variations and adaptations to the POMDP approach for dialog modeling were proposed over the years. For a more current survey and review of available techniques refer to [YGTW13].

3.2.4.3 Hidden Markov Models

Chomsky Rank	?
Structured Input	(No)
Logical Resolution	No
Distributed	No
Dialog State	Density

Table 3.12.: Features of dialog management with HMMs.

The formalism of a Hidden Markov Model (HMM) describes another probabilistic automaton with a purpose different from MDPs and its variations. An HMM allows to model a remote system as a stochastic process and reason about its probable state at a given point in time by observing its output. As such, it is a means to explain its dynamic behavior. There are three specific problems [Rab89] the HMM formalism attempts to solve:1. What is the probability of a remote system to emit a given sequence of observations, 2. what sequence of assumed states does explain a given observation best and 3. how to derive a HMM given a corpus of observation sequences. We can formalize a $HMM = (S, V, A, B, \pi)$ as:

$$
\begin{aligned}
S &:= \{s_0, ..., s_N\}, \text{ the set of discrete states for the remote system} \\
V &:= \{v_0, ..., v_M\}, \text{ the set of distinct observations we can perceive} \\
A \in \mathbb{R}^{N \times N} &:= \text{The state transition probability matrix} \\
B \in \mathbb{R}^{N \times M} &:= \text{The emission probability matrix} \\
\pi \in \mathbb{R}^{N} &:= \text{The probability for the initial states}
\end{aligned}
$$

With a_{ij} denoting the probability to change from state s_i to state s_j and $b_i(j)$ the probability to emit observation v_j in state s_i. The major application of the formalism is a solution to the second problem listed above: An assumption of a likely sequence of states, a remote system assumed in order to generate a perceived sequence of observations. The probabilities for the best sequences of states can be calculated by the *Viterbi algorithm* [Vit67]. It allows to identify the most probable sequence of states ending in any given state that explains the observations. By merely selecting the end state with the highest probability, back-tracking can be employed to arrive at the best overall sequence of states that explains the observation.

In the work of Boyer et al. [BHP+09], an approach is described to infer sequences of *dialog modes* by observing a sequence of dialog acts as interactive events. They introduce a system for an interactive tutoring environment in the computer science domain where a labeled corpus with dialog act sequences, such as *question*, *statement*, *grounding* or qualified forms of *feedback* is used as the basis to train two different HMMs. The first HMM employs the dialog acts themselves as observations, wherein the second operates on adjacency pairs of dialog acts, not unlike diphones with speech recognition. For both variations, they train HMMs and suggest HMMs with 5 and 4 hidden states respectively. These states correspond to *dialog modes* that identify general phases of the tutoring interaction, such as *tutor lecture*, *grounding* and *question/answer*. It is unclear, how this approach can be employed to realize all the responsibilities of a dialog manager, and there are hardly any other publications attempting HMM-based dialog management. However, it does provide a promising additional knowledge source for the more established dialog management techniques.

3.2.5 Summary

With Petri nets occupying only a niche in the dialog management techniques, state-charts constitute the most expressive, best understood deterministic approach to describe dialog models. Their major advantage over flat transition networks is their more compact representation and suitability for graphical authoring environments while retaining the same expressiveness as their flat counterparts. We will provide an impression about just how much more compact state-charts can be in section 5.3.8 when we introduce an upper bound for the number of distinct *global states* in a state-chart.

However, the plethora of possibilities for a formal semantics of their behavior diminish their usefulness and applicability. Here, SCXML provides a solution by standardizing an (actually arbitrary) semantics as a W3C recommendation. The abstraction of the action language into a *data-model* ensures considerable flexibility and allows a platform to offer a wide range of approaches for dialog management. In fact, the Prolog data-model extends the applicability of SCXML for all rule-based approach as we will show in section 4.5.

With regard to the probabilistic automatons, there is no obvious extension to state-charts in general or SCXML in particular to incorporate their approach to dialog management. While MDPs might be used to decide upon the most beneficial transitions, representing the dialog state as a probability distribution seems just too foreign to state-charts with their deterministic state configurations.

Name	Structured Input	Logical Resolution	Distributed	Dialog State	Chomsky Rank	Remarks
Automatons						
Transition Networks						
Simple	No	No	No	Discrete	3	
Recursive	No	No	Yes	Discrete	2	
Annotated	(Yes)	No	No	Discrete	1	
State Charts	(Yes)	No	(Yes)	Discrete	0-3	Depends on formalization [vdB94]
SCXML with data-model:					0	
NULL	No	No	Yes	Discrete	0	
ECMAScript	Yes	No	Yes	Uncountable	0	
Lua	Yes	No	Yes	Uncountable	0	
PROMELA	Yes	No	Yes	Enumerable	0-3	Rank 3 with additional restrictions
Prolog	Yes	Yes	Yes	Uncountable	0	
Petri Nets	No	No	(Yes)	Discrete	0-3	Depends on formalization
Frame-based	Yes	(Yes)	No	Uncountable	?	Not an actual formalism
Rules						
Information State Update	Yes	Yes	No	Uncountable	0	
Agent-based	Yes	Yes	(Yes)	Uncountable	0	
Plan-based	Yes	Yes	No	Uncountable	?	Rather a supporting technique
Probabilistic						
Markov Decision Processes	(No)	No	No	Discrete	?	
Partially Observable	(No)	No	No	Density	?	
Hidden Markov Models	(No)	No	No	Discrete	?	

Table 3.13.: Features of dialog management techniques.

In this chapter, we will more formally introduce State Chart eXtensible Markup Language (SCXML) as an XML dialect and a standard recommended by the World Wide Web Consortium (W3C) for the formal semantics of state-charts in general and multimodal dialog management in the scope of the W3C Multimodal Interaction Working Group (W3C MMI WG) in particular.

We will begin by describing its syntax as specified in the W3C recommendation and give an impression about its expressiveness and applicability. This is followed by a description of its semantics, i.e. its interpretation of a given state-chart as a sequence of micro- and macro-steps, similar to the concepts found in the original formalizations of Harel and Pnueli [PS91] introduced in the previous chapter. With the semantics in mind, we can present a set of validity criteria that can be used to identify potential issues with a given state-chart description. This is very useful to avoid common pitfalls and largely a consequence of feedback from deployments of our interpreter at industrial partners.

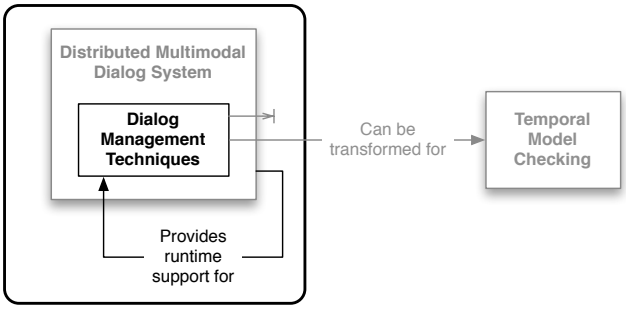


Figure 4.1.: This chapter tackles the second part of the proposition "*Multimodal interaction in pervasive environments can be expressed in a dialog model on a suitable platform.*" from the main argument of this thesis (cf. figure 1.1).

Subsequently, we will compare SCXML as a dialog management technique to the other approaches introduced in chapter 3 and present some adaptations within the confines of the standard, i.e. to extend its applicability towards all *rule-based* approaches. With regard to our ultimate goal of temporal verification of a state-chart, we have to keep in mind, that not all of these proposed extensions are necessarily transformable onto the input language of the SPIN model-checker as we will see in chapter 6. The chapter concludes with some apparent deficiencies of SCXML and an attempt to describe some respective solutions.

4.1 The State-Chart XML Language

Work for a standardization of state-charts as a XML dialect with a defined formal semantic began as early as 2004 in response to the requirement for a common interpretation of the visual formalism by Harel [Har87]. One of the major design goals of the standard was a complete and deterministic description of the execution semantics of state-charts. This aspect was deemed inviolable in response to the many problems with the conflicting semantics [vdB94] found, e.g., in state-charts from Unified Modeling Language (UML) activity diagrams. In fact, arbitrary but defined semantics were sometimes chosen over ambiguous definitions and undefined behavior. As such, every conforming interpreter with a given SCXML document will transition through the very same set of configurations and perform the same actions in the same order when exposed to the same sequence of events.

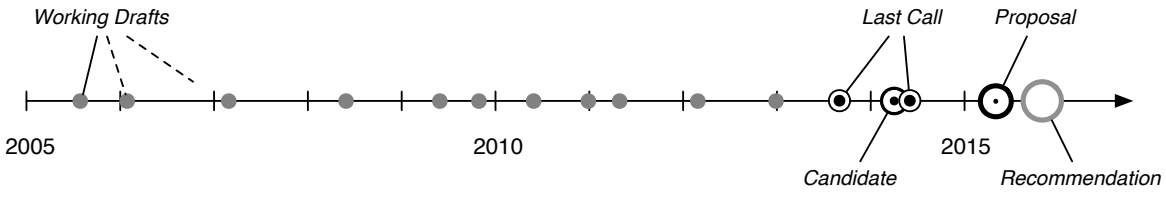


Figure 4.2.: Timeline for the W3C publications regarding the SCXML standard.

During the writing of this thesis, the W3C SCXML standard progressed through a series of statuses, from *working draft* status through *proposed recommendation* and finally a proper *recommendation* in September 2015 (see figure 4.2). The complete graph established by the *valid child of* relation of XML elements defined in SCXML as its syntax is given in (figure 4.3).

At its core, an SCXML document encodes a Harel state-chart as introduced in the original description of the visual formalism [Har87] with one important difference: all transitions and states are inherently ordered. With SCXML

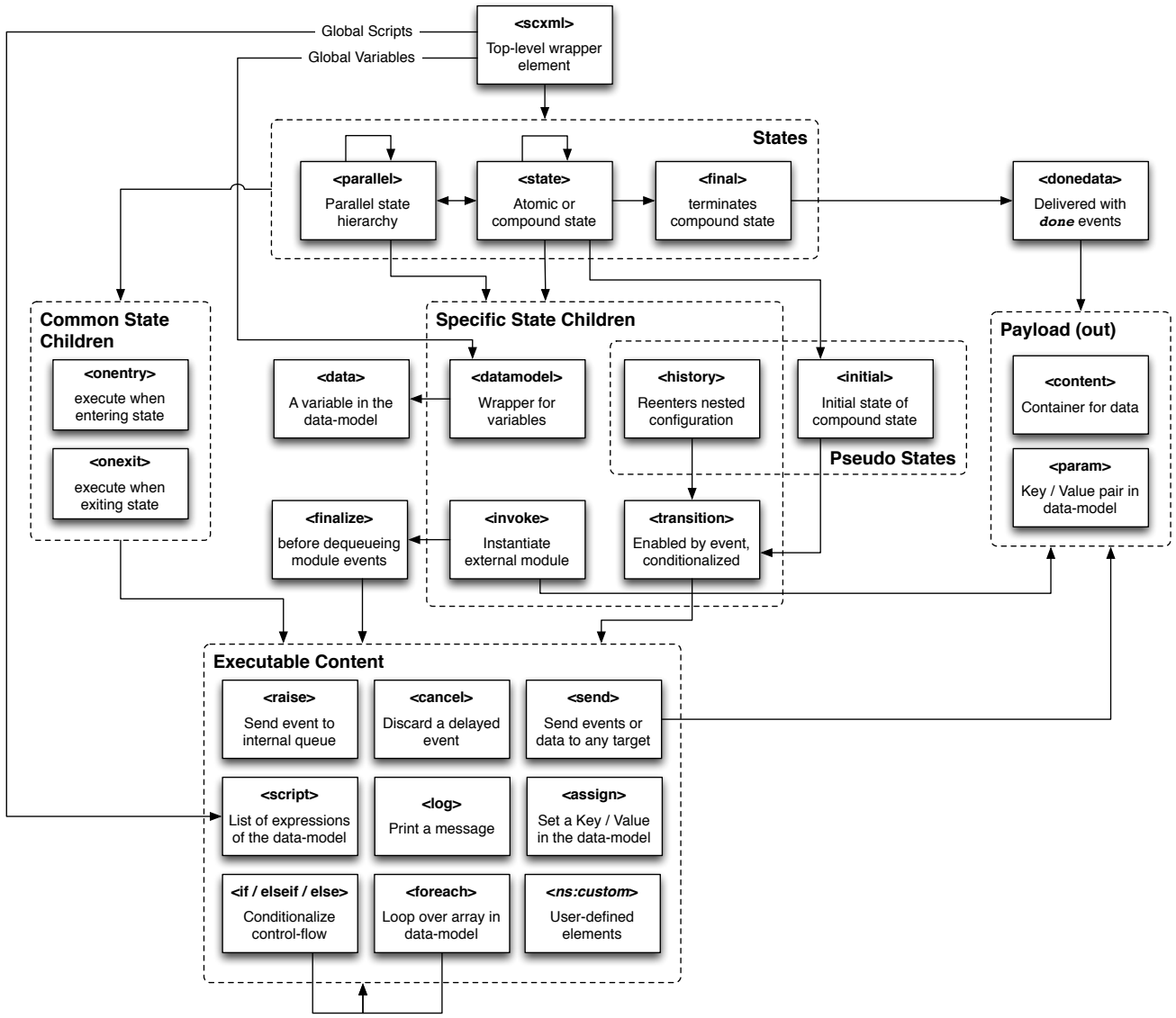


Figure 4.3.: The complete *child-of* relation for elements in SCXML.

being a XML dialect and thereby a textual representation, the order in which transitions are written in the document carries semantics that are inexpressible in Harel's original visual formalism. This order is used extensively to guarantee the determinism of the execution semantics.

Apart from the ability to express Harel state-charts, another important consideration of the standard is the *action language* it defines, as the operations available as a *side-effect* when transitioning through active state configurations in response to events. As we will see below, hardly any of the XML elements constituting the action language of SCXML are useful without an interpreter providing an embedded scripting language (called *data-model* in the SCXML standard). The data-model defines a set of syntactical and semantical requirements for a formal language and virtually any traditional programming language can be employed as long as there is an interpreter with a respective implementation. The standard itself describes a normative data-model for `ECMAScript` and used to contain a `XPath` data-model, which was dropped due to missing implementation reports. Many other programming languages were proposed and are supported in the various SCXML interpreter platforms.

To transition between configurations, an interpreter depends on events. These are either raised internally by the state-chart itself, delivered from external, *invoked* components or via *I/O processors*. External events are only processed when all internal events are exhausted and, as such, there are two respective queues for incoming events, one internal queue and one external queue.

In the following sections, we will present these aspects in more detail. We start by the general approach to express Harel state-charts with the language features available in SCXML. Afterwards, we will have to introduce the data-model as virtually all subsequent aspects, most notable the action language, rely heavily on its functionality to do

any meaningful work. This is followed by a concluding presentation of the remaining SCXML language elements for external communication and a brief presentation of events raised automatically by a compliant interpreter.

4.1.1 Modeling Harel State-Charts

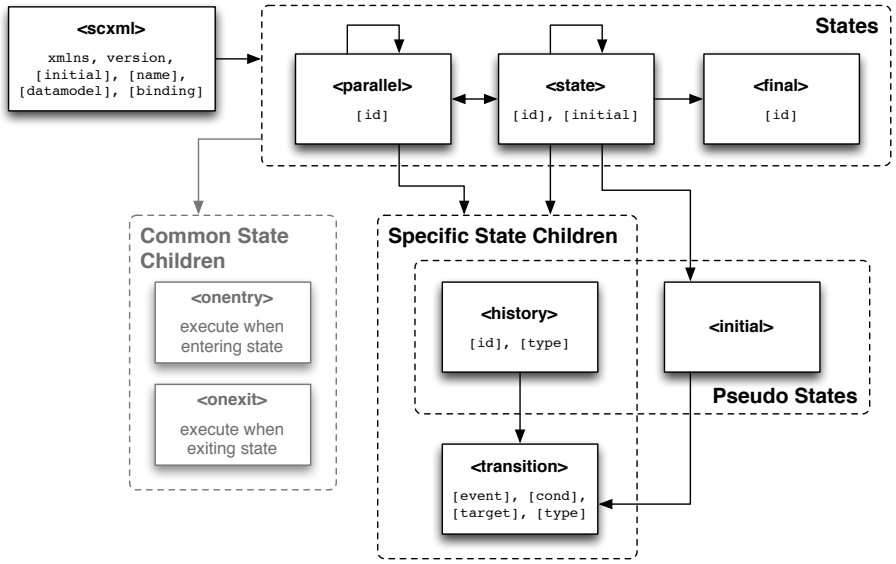


Figure 4.4.: SCXML elements pertaining to the visual formalism of state-charts with attributes listed.

The XML elements available to model the visual formalism of Harel state-charts are depicted in figure 4.4. Here, and in the remainder of this thesis, we will use the nomenclature of the SCXML standard and only briefly reference the respective terms from the formalization of Pnueli were applicable:

Top-level element:
Every SCXML document is contained in a topmost `<scxml>` wrapper element (its root state) as a compound state (*or*-state). This element requires to specify the state-chart's `version` and the XML namespace, which is required to be `http://www.w3.org/2005/07/scxml`. The other attributes are primarily related to the data-model and discussed in section 4.1.2 below.

Set of states:
Every `<state>`, `<parallel>` and `<final>` element constitutes a (proper) state, their union the set of states $S :=$ `<state>` $\cup$ `<parallel>` $\cup$ `<final>`. In addition, the set of `<history>` and `<initial>` elements constitute the pseudo-states. All proper and pseudo states can define transitions and have an optional identifier given in its element's `id` attribute. Pseudo-states only influence a states *completion* (see below) and can never be contained in a state-chart's configuration.

Type of states:
If a `<state>` element s contains no other states ($children^+(s) = \emptyset$), it is an *atomic* (or basic) state and a compound (*or*-state) otherwise. Every `<parallel>` state is an *and*-state.

Configuration:
The set of states active at a given point in time is referred to as the *active configuration* S_a of the state-chart (a maximally consistent set). The subset of atomic states in the active configuration constitutes the *basic configuration*. It is interesting to note, for the formal semantics later, that every transition is ultimately enabled by a single basic state from the active configuration and a basic state can enable one transition at most.

Default Completion:
The default completion for a compound state c is its first child state $s \in S \wedge s \in children^1(c)$ given in document order unless a dedicated `initial` attribute or mutually exclusive `<initial>` element is specified. The former contains a list of states that constitute a consistent configuration of child states, the latter a `<transition>` element with a target state $s \in children^1(c)$.

History States:
Entering a state s via an eventual history pseudo state (`<history>` $\in children^1(s)$) changes the default completion. If the type of `<history>` is `shallow`, the set of child states $\in children^1(s)$ that were active when s was last left is entered again. For `deep <history>` elements, the complete set of child states $\in children^+(s)$ is entered again.

Transitions:
In SCXML, transitions are specified via `<transition>` elements and can be contained in both, states and pseudo-states. Unlike with the formalization of Harel and Pnueli, it is not possible to describe a transition t originating in more than one state $|source(t)| > 1$. This is not a problem, though, as the same element can just be copied into all respective source states with equivalent semantics. The set of transitions T as direct children of active states $t \in children^1(s), s \in S_a$ constitutes the set of potentially enabled transitions in response to an event. Every transition as the child of a proper state can have (i) an `event` attribute containing an *event descriptor* to match a set of event names (see next item), (ii) a `cond` attribute as a boolean expression in the data-model's language, (iii) a space separated list of state identifiers in the `target` attribute and (iv) a `type` attribute of `internal` or `external` (default) with consequences for its *domain* (*arena* with Pnueli).

Events:
For most formalization of state-charts, events are represented by simple literals [vdB94] and transitions would be enabled by matching these literals or their negation (to provide a simple means to express transition priorities). In SCXML events have a name as a dot-separated sequence of string literals $\texttt{l}_1\texttt{.l}_2\texttt{...l}_\texttt{n}$. This allows to organize events into a tree of events, each branch identified by an ever deeper constituting string literal (e.g. `doorPanel.button3.pressed` or `doorPanel.button3.released`). The `event` attribute of a `<transition>` contains a space separated sequence of event descriptors that allow to refer to individual leafs or complete branches in this tree. If we regard the constituting literals as symbols and their concatenation as a word ($\texttt{l}_1\texttt{l}_2\texttt{..l}_\texttt{n}$), we can say that a transition *matches* an event if one of its event descriptors is a prefix or exact match of the event's name as such a word. E.g. the event descriptor `foo` will match all event names starting with `foo.` and the event `foo` itself. It will, however, not match anything starting with `foobar`, as `foo` and `foobar` are different symbols.

In the original work from Harel, events could be rather generic, e.g. "when in state `update` for 2 minutes", "when time `T` is reached" or just about any condition expressible via the action language, however, not all of those were formalized and none of those are available in SCXML.

Final States:
The original description of state-charts from Harel did not include final states and, as such, no means for a state-chart ever to terminate. In SCXML, a `<final>` state is allowed for the topmost `<scxml>` element and every compound state. Entering a final state that is contained in the topmost `<scxml>` element will cause the complete interpretation to terminate. Whereas entering the final state of a compound state c will issue an *internal event* `done.state.[c.ID]` and remain there until the containing state hierarchy is left. Additionally, entering a final state for every child state of a parallel state p will issue `done.state.[p.ID]`.

We have already seen a visual state-chart expressed in SCXML in the previous chapter, when the example with most of Harel's visual syntactical elements from figure 3.21 was encoded in the SCXML document in listing 3.1. The similarities in syntax and nomenclature will have to suffice for a convincing argument that, indeed, every state-chart conforming to Harel's visual formalism can be expressed in SCXML. Though, not vice versa, as there is no notion to express the document-order of elements from SCXML in Harel's visual formalism.

4.1.2 The Data-Model

Before we continue the introduction of actual SCXML language elements, we have to introduce the concept of the *data-model*. The data-model is the (unfortunate) term for an optional, scripting language accessible via many SCXML elements and attributes (see table 4.1). The process of *data-modeling* is described, e.g., in the context of UML whose activity diagrams bear some resemblance with SCXML. However, for all intents and purposes, the SCXML data-model is not only a model of data, but usually a Turing complete *scripting language* and as such this very term would seem more apt. We will refer the data-model as *data-model* when we specifically talk about its context in SCXML and prefer *(embedded) scripting language* otherwise.

In its essence, the data-model is an isolated runtime for the syntax and semantics of a formal (programming) language selected via the root element's `datamodel` attribute. The SCXML standard does describe a normative

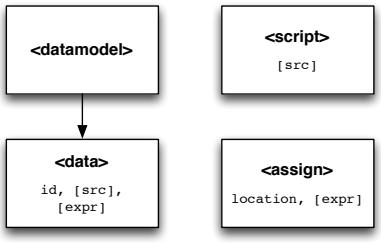


Figure 4.5.: SCXML elements exclusively related to the data-model.

`ecmascript` data-model and used to contain a normative description about an `xpath` data-model, which was dropped due to insufficient implementation reports. While many more languages are employed by the different SCXML implementation, ECMAScript seems to enjoy the most widespread use.

The connection of the data-model's runtime to the interpretation of an SCXML document is realized via (i) the attributes and elements of table 4.1, (ii) the set of XML elements depicted in figure 4.5, (iii) a few system variables to make dynamic runtime information available to the state-chart and here, most notably (iv) a representation of the current event in a data-model variable called `_event`. The SCXML elements from figure 4.5 are exclusively related to the data-model and provide the principal means to change the state of a respective runtime from within a state-chart:

- The `<datamodel>` element is just a wrapper for `<data>` elements with no attributes or other child elements defined.
- A `<data>` element declares and optionally defines the value for a *location* in the data-model's runtime. The term location is intentionally vague as not all possible data-models do necessarily offer the concept of variables as such. A `<datamodel>` element containing `<data>` elements can be a child of any proper state. If the topmost `<scxml>` has its `binding` attribute set to `late`, the respective location will only be declared / defined if the respective state is initially entered, otherwise all `<data>` elements are evaluated at the start of interpretation.
- `<assign>` elements are the equivalent of `<data>` elements for locations that are already declared and can appear anywhere as *executable content*. Their precise semantics is, as with `<data>`, dependent on the actual data-model and additional attributes are allowed to account for specific semantics of an assignment. For instance, in our Prolog data-model, `<assign>` will establish a fact as a predicate given in `location`. Here, we support an additional attribute for `<assign>` to specify whether already established facts are retracted first or the new fact is appended or prepended.
- The `<script>` element either references a URI in its `src` attribute, where a sequence of statements in the language of the data-model can be retrieved, or allows to provide such statements inline as a text child element.

For the *system variables*, the standard mandates the following variables to be available in the data-model:

The `_event` variable:
The standard mandates that the current event, to which a state-chart responds, has to be available in a compound data-model variable called `_event`. The structure of this compound variable is as follows:

- `_event.name` is a string containing the name of the event as a dot-separated sequence of literals ($l_1 l_2 .. l_n$).
- `_event.type` is one of `{platform, internal, external}` and describes the general source of the event. Events of type `platform` are received from the SCXML interpreter itself and usually denote some kind of error during interpretation (e.g. a timeout when fetching a remote resource or invalid syntax with data-model expressions). Internal events always originate from a `<raise>` element or a `<send>` element with a `target` attribute set to `_internal`. All other events are of `external` type and are usually delivered into the interpreter via `I/O processors`, an embedding application or external, *invoked* components.
- `_event.sendid` is not mandatory for all events and its availability depends on the *I/O processor* that received the event. For two communicating SCXML documents, it is required to contain the `id` of the others documents `<send>` element. For messages received via other means the value is undefined. If it is given, it ought to contain a unique identifier for the received message.
- `_event.origin` is an URI, which, when set as the `target` attribute of a `<send>` element addresses the entity from which the message was received.

Element	Attribute	Type	Remarks
`if`	`cond`	`boolean`	Conditionalize executable content in child elements.
`elseif`	`cond`	`boolean`	
`transition`	`cond`	`boolean`	Guard a transition with a boolean expression.
`log`	`expr`	`string`	Evaluates to a string for logging.
`send`	`eventexpr`	`string`	The event's name to send.
`send`	`targetexpr`	`string`	The target to send the event to (`type` specific).
`send`	`typeexpr`	`string`	The type as the `I/O processor` to use (e.g. `basichttp`).
`send`	`delayexpr`	`string`	A time specification to delay the sending.
`send`	`idlocation`	`location`	A variable to set to an identifier for eventual canceling.
`send`	`namelist`	`location array`	A list of variables with their values to be encoded in the payload of a message (`type` specific).
`invoke`	`typeexpr`	`string`	The kind of entity to invoke (e.g. `xhtml`).
`invoke`	`srcexpr`	`string`	Instantiate entity with a document (e.g. a xhtml page).
`invoke`	`idlocation`	`location`	Set to an identifier to address this invoker via `<send>`.
`invoke`	`namelist`	`location array`	Variable values to pass to the invoked component.
`cancel`	`sendidexpr`	`location`	Abort delayed send with identifier from variable.
`foreach`	`index`	`location`	A variable to contain the iteration index.
`foreach`	`item`	`location`	A variable to contain the current item.
`foreach`	`array`	`arbitrary array`	An iterable collection in the data-model
`data`	`id`	`location`	The variable (or compound member, array index) to set.
`data`	`expr`	`arbitrary`	A data-model specific expression with arbitrary value.
`data`	*text child nodes*	`arbitrary`	A data-model specific value (e.g. JSON or XML).
`assign`	`location`	`location`	Same as the `id` attribute of `<data>`.
`assign`	`expr`	`arbitrary`	Same as with `<data>`.
`assign`	*text child nodes*	`arbitrary`	Same as with `<data>`.
`content`	`expr`	`arbitrary`	Same as with `<data>`.
`content`	*text child nodes*	`arbitrary`	Not subject to evaluation per standard, but vastly more useful and a common extension.
`param`	`expr`	`arbitrary`	A data-model specific expression with arbitrary value.
`param`	`location`	`arbitrary`	A variable in the data-model of arbitrary type.
`script`	*text child nodes*	`statements`	Sequence of statements from the data-model's syntax.
`finalize`	*text child nodes*	`statements`	Sequence of statements from the data-model's syntax.

Table 4.1.: Data-model dependent elements and attributes in SCXML by element.

- `_event.origintype` is the equivalent of the `type` attribute with a `<send>` element and contains the name of I/O processor via which the message was received.
- `_event.invokeid` contains the value of the `id` attribute of an `<invoke>`, when the message was received from a component invoked via the state-chart itself (see section 4.1.4 below).
- `_event.data` contains any payload associated with the event's message. The standard is somewhat evasive as to how this data is to be presented and its valid types. The SCXML IRP tests for the `ecmascript` data-model do clarify the issue to some extend, e.g. any data formatted in JSON has to be mapped as a respective compound and in one test XML elements are contained.

All fields of the `_event` compound but `name` and `type` depend on the specifics of the event's message arrival. With the representation of `_event.data` even specific to the employed data-model.

In our implementation, the data field is represented by a generic, data-model agnostic `Data` class that can contain arbitrary values. It features five different, mutually exclusive fields to represent data: (i) an `atom` field as a single scalar value in either verbatim or interpreted form, (ii) a `compound` to contain nested `Data` objects for specific keys, (iii) an `array` field for `Data` objects addressable via a numeric index, (iv) a `xml` field with a representation of a XML structure accessible via the interface defined in the W3C DOM Level 2 Core specifications and (v) `binary` data as an arbitrary sequence of bytes. When representing instances of the `Data` class in the data-model, it depends on the specific implementation to make these fields available in the data-model's runtime.

Other system variables:

- The `_sessionid` variable contains a string as a unique identifier for a running SCXML interpretation. Its major application within the standard is the ability to address any running interpreter via a `target` attribute of `#_scxml_[_sessionid]` in the `<send>` element.
- The `_name` variable contains the value of the `name` attribute of the topmost `<scxml>` element.
- The `_ioprocessors` variable is a set of key value pairs. It contains for every name of an I/O processor its location as a protocol specific address for external systems.
- The compound `_x` variable is a root for any platform-specific extensions.

In general, all data-model variables starting with an underscore are reserved for future usage of SCXML. Again, it is not defined how these variables are actually represented in the data-model as an embedded scripting language. While there is, most often, a straight-forward approach to map them directly onto respective variable names, there are instances, e.g. with the Prolog or an eventual Lisp data-model, where this is less obvious.

The SCXML standard tries hard to avoid any assumptions about the specifics of the syntax and semantics of any (programming) language employed as the data-model. A normative ECMAScript data-model is described but, in general, the encapsulation of the data-model and its responsibilities is rather felicitous. This is evidenced by the plethora of data-models available in the various SCXML interpreter platforms. Our implementation alone provides support (with different levels of maturity) for ECMAScript/JavaScriptCore, ECMAScript/V8, ECMAScript/Rhino, Lua, Prolog/SWI, XPath/Arabica and PROMELA, all via a common interface for data-models. It is this interface that provides some more insights with regard to the functionality required from the data-model.

Type	Name	Arguments	Remarks
`bool`	`isLocation`	`string expression`	True if `expression` is a location. E.g. with ECMAScript, this is required to be a valid left-hand side expression and is true is `expression++` is syntactically valid.
`bool`	`isDeclared`	`string location`	True if the `location` is already declared, e.g. via a `<data>` element already evaluated.
`void`	`setEvent`	`Event event`	Represent given event as `_event` in the data-model. Structure of the `Event` class is as described above with its `data` field a `Data` object.
`uint`	`getLength`	`string expression`	Try to interpret `expression` as an iterable entity and return its length to iterate via `<foreach>`.

`void`	`setForeach`	`string item,`
		`string array,`
		`string index,`
		`uint32_t iteration`
			Assume `array` to be an iterable entity, set the location in `item` to its entry at index `iteration` and the location at `index` to the current iteration. Required to support the `<foreach>` element.

`void`	`assign`	`string location,` `Data data`	Assume `location` to be already declared and set its value to a representation of the given `Data` object. This is called e.g. for `<assign>` elements.
`void`	`init`	`string location,` `Data data`	Declare `location` to be a valid variable and optionally set its value to a representation of the given `Data` object. This is called e.g. for `<init>` elements and usually realized by declaring `location` and calling `assign`.
`void`	`eval`	`string expression`	Evaluate the `expression`, e.g. as given in a `<script>` element.
`string`	`evalAsString`	`string expression`	Evaluate the `expression` and convert result into a string. E.g. for the `expr` attribute of the `<log>` element.
`bool`	`evalAsBool`	`string expression`	Evaluate the `expression` and convert result into a boolean. E.g. for the `cond` attribute of a `<transition>` element.
`Data`	`getStringAsData`	`string content`	Evaluate the given string and represent its value as a Data object. E.g. to create a JSON representation for external communication.

Table 4.2.: The API of the uSCXML interpreter for data-models.

While the requirements for the functionality of a given data-model, exemplified in the API in table 4.2, seem rather generic and straight-forward to provide for any runtime of a programming language, there is a slight bias towards languages with an imperative programming paradigm. This is especially evident in the `<foreach>` element and its related API calls `getLength` and `setForeach`. While iterating a collection of items and assigning them to some ephemeral variable is natural to do in imperative programming languages, it is an unorthodox approach e.g. with functional or logic programming.

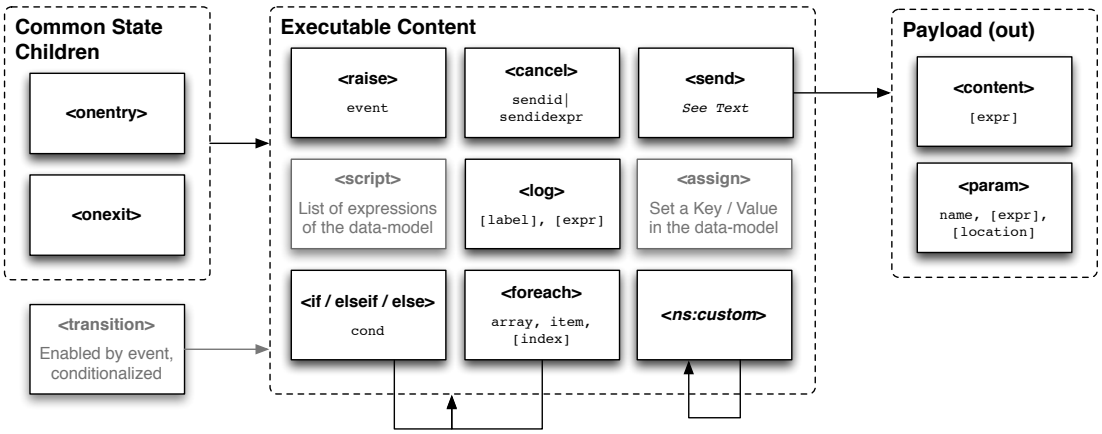


Figure 4.6.: SCXML elements constituting its action language.

A state-chart's *action language* defines a set of operations available during the interpretation of the state-chart, e.g. upon entering and exiting of states or when taking transitions in response to events. Some operations are directly related to its interpretation and have an influence on its future transitions and configuration, while other operations (called *activities* by Harel) provide the means to connect a state-chart to an actual problem domain.

In SCXML all aspects of the action language pertaining to the first aspect are subsumed into a set of elements called *executable content* depicted in figure 4.6 (activities would be modeled via `<invoke>` for external components, see below). Elements from this set are valid children of `<onentry>`, `<onexit>` and for `<transition>` elements and, as such, interpreted whenever states are entered / exited or transitions taken. The distinction between actions and activities is not clearly defined in SCXML. Theoretically, every action with externally observable behavior can be abused to control activities and we will revisit this point explicitly when we talk about *application specific instantiation* in section 4.2.

Element	Requires Data-model	Externally Observable	Function
`log`	No	Yes	Write an expression or string literal with an optional label into a log file.
`send`	No	Yes	Send a message to a given target via a given `I/O processor`.
`raise`	No	No	Enqueue a given simple event literal on the internal queue.
`cancel`	No	No	Abort a delayed `<send>`.
`script`	Yes	No	Evaluate a sequence of statements in the embedded scripting language.
`assign`	Yes	No	Assign a variable (*location*) of the embedded scripting language a given expression, XML structure, string literal or compound data.
`if / elseif / else`	Yes	No	Conditionalize contained executable content via a boolean expression in the embedded scripting language.
`foreach`	Yes	No	Iterate contained executable content for every item in an array from the embedded scripting language.
`ns:custom`	?	?	Execute user-supplied or platform-specific functionality.

Table 4.3.: Overview of executable content in SCXML.

A listing of the elements constituting the executable content of SCXML is given in table 4.3. It is noteworthy, that hardly any of these elements are of any use without a data-model and all will have greatly extended expressiveness in its presence.

log

The `<log>` element allows to generate a log or debug message. It can be employed without an embedded scripting language by providing string literals for its `expr` attribute and the optional `label`. However, in the presence of a data-model, the `expr` can be any expression that evaluates to a value. In our implementation, this is not limited to string values, but any data type is accepted and will be logged as its JSON representation.

send

The <send> element is a rather versatile language feature to emit messages to just about any endpoint, as long as a corresponding I/O processor is defined. Here, a message is usually an event with optional payload of arbitrary type. The actual means of delivery and the structure of the message to be sent depend on the given *type*, which selects an I/O processor to handle the encoding and delivery. While all attributes and child elements of a <send> element are potentially available for an I/O processor to encode into a message, they can be classified into those with a semantic common to all I/O processors (1-3) and those attributes with a semantic solely defined by the I/O processor (4-8):

1. The id attribute can be used to specify the identifier of a message. Alternatively, the idlocation attribute may contain a location in the data-model where the platform is to write an auto-generated identifier.
2. The delay or mutually exclusive delayexpr can be used to specify an optional time designation to delay the delivery of a message. If a message is delayed and has an identifier its delivery can be cancelled via <cancel> if it was not yet delivered.
3. The type attribute, or the mutually exclusive typeexpr attribute selects the specific I/O processor responsible to deliver the message by its name. Within the standard, only the scxml I/O processor to communicate among SCXML runtimes per their _sessionid and the basichttp processors are defined. A normative section about a dom[1] I/O processor to communicate among different XML Document Object Model (DOM) environments was moved into separate working group notes due to missing implementation reports. Our implementation also features a http I/O processor which allows to send messages via long-polling HTTP requests, e.g. initiated from Hypertext Markup Language (HTML) browsers via a XMLHTTPRequest object to communicate with their ECMAScript runtime. If the type attribute is omitted, the default scxml I/O processor is employed.
4. The target attribute or its dynamic variation targetexpr is used to provide a type-specific URI identifying the remote endpoint(s) as the receiver. For the default scxml I/O processor, a few special targets are available: (i) #_internal will address the state-chart's own internal event queue (ignoring an eventual delay specification), (ii) #_scxml_[_sessionid] addresses another state-chart per session identifier, (iii) #_parent delivers an event to an eventual state-chart that invoked the issuing one and (iv) #_[_invokeid] allows to address an *invoked* (see below) system .
5. The event's name is specified either as a string literal in the event attribute or as an (mutually exclusive) expression in eventexpr, evaluating to a string value on the data-model.
6. The namelist attribute contains a set of locations on the data-model to include in the message. They may evaluate to arbitrary values and it is the responsibility of the I/O processor to encode them appropriately.
7. The <param> children of <send> can be used to specify one key/value pairs. This child element is mutually exclusive with the namelist attribute and extends its expressiveness by the means to specify *names* for arbitrary values from the data-model.
8. Finally the <content> child element may be used to specify arbitrary content, either inline as a set of child elements or per URI in its src attribute.

While the actual semantics on how these attributes and elements are encoded in a message to be emitted depend on the I/O processor, we can, nevertheless, assign some general semantics an application developer might expect.

The responsibility of an I/O processor addressed as the type of a <send> element does not only include the appropriate encoding of these attributes and child elements, but also the decoding of respective messages received from external systems. This decoding entails a representation of the messages received as events delivered into the state-chart (see structure of the _event system variable above). As such, there are some members of an event, with a natural mapping from attributes and child elements of a <send> invocation. The _event.name will contain the name of the event specified via the event or eventexpr attribute and the _event.sendid will contain the values of id or idlocation respectively. The event's type member will be the name of the I/O processor chosen via its type and the origin will be an URI encoding the address of the emitting endpoint.

However, for the attributes and child elements specifying the actual payload, i.e. namelist, <param> and <content>, a representation in the _event structure is more ambiguous and somewhat underspecified in the standard. The general expectation is to represent these values in the data member field of an event. As per

[1] http://www.w3.org/TR/scxml-dom-iop/ (accessed 26th October, 2015)

standard, it is allowed to specify either a `<content>` child element or the `namelist` attribute and the `<param>` child with a `<send>` element.

As such, in the former case, `_event.data` would be a representation of whatever is contained in the `<content>` child element. If this would be a set of XML elements, we can no longer represent it as a single XML node in `_event.data` but need a `NodeList` or a wrapper element - the SCXML standard does not address this. In the latter case, `_event.data` would be a compound with the names of given parameters and the tokens from the `namelist` as the keys. Here, it is mandated that eventual duplicate keys are to be encoded in the message to be send, but no behavior to resolve duplicate keys in the `_event.data` compound is specified.

raise

The `<raise>` element is only available to deliver a single event literal to the internal queue. No additional data can be specified. As such, its functionality is a subset of `<send>` as both allow to address the internal queue, but `<send>` will allow arbitrary data attached to an event in addition.

cancel

As hinted at above, the `<cancel>` element can be used to abort a delayed message per identifier. Its sole attribute is `sendid` or its dynamic pendant `sendidexpr` to specify the identifier of a `<send>` invocation that is still pending due to its delay. Also, all delayed messages send by an invoked, nested state-chart will be cancelled if the state-chart is terminated before the messages are send.

script

The `<script>` element was already described when the data-model was introduced above as it is of no use without one. It will allow to specify a sequence of statements from the data-model's language, much like the `<script>` element from e.g. HTML and is the principal language feature to modify the state of the data-model's runtime interpreter.

assign

The functionality of the `<assign>` element is a subset of the `<script>` element. In essence, it is just another notation for assignment statements from the data-model's language. TheSCXML standard explicitly allows to add any additional attributes to specify the details of an assignment. This is relevant, e.g. with our Prolog data-model, where assignments are facts expressed via a predicate and an additional attribute will allow to append or prepend the new fact or retract all other members of the respective predicate prior.

if / elseif / else

These three elements have the same semantics as in just about any imperative programming language. In SCXML, they will conditionalize a set of executable content as its child elements via a boolean expression on the data-model in their `cond` attribute. It is noteworthy that `<elseif>` and `<else>` are always direct children of an `<if>` element and serve as *delimiters* of other executable content at the same nesting depth. They provide the means to employ data-model specific conditions to select one set of executable content over the other. This is important as it would not be possible, e.g. to conditionally `<raise>` internal events as there is usually no representation of the respective functionality in the data-model.

foreach

The `<foreach>` element allows to iterate a set of executable content for every item in a iterable collection from the data-model. The iterable collection is specified as a variable or an expression in its `array` attribute and for each iteration, the data-model location given in `item` is set to the current item with the optional `index` set to the current iteration number. As with the `if / elseif / else` elements above, it allows to refer to values contained in the data-model to provide the values for executable content, most notably the `<send>` element with all its `*expr` attributes.

ns:custom

Finally, the SCXML platform can offer the means for an application developer to register any custom executable content in any XML namespace. Our implementation does use this possibility to offer:

- A `<postpone>` element which will stop processing and redeliver an event when a given condition in its `cond` attribute is given.
- A `<fetch>` element to asynchronously retrieve the contents of a given `src` URL and deliver a specified `callback` event with respective data once the resource is available.
- An optional `<file>` element to perform the usual operation on local files.
- A `<respond>` element to return data for requests from remote entities.

Of all those executable content elements only `<log>`, `<raise>`, `<send>`, `<cancel>`, and possibly `<ns:custom>` are of any use without a data-model, and all but `<raise>` will become immensely more useful once a data-model is specified. Furthermore, if we disregard the `<log>` element, the only means per standard to actually *do* things with executable content is to send messages to other entities. This may already be sufficient, but we will see in section 4.2 how an SCXML state-chart can be extended / instantiated within the standard to control an actual application.

4.1.4 External Components

While the `<send>` element, with an appropriate I/O processor will already allow to send messages to arbitrary end-points. The `<invoke>` element is available to even instantiate external entities and control their lifecycle: Whenever a state with an `<invoke>` child element is in the active configuration at the end of a macro-step, the platform will instantiate / invoke the specified entity. If the respective state is no longer in the active configuration at the end of a macro-step, the invocation is cancelled. This is the principal extension point for systems more elaborate than a simple I/O processor as it will allow to instantiate a session with a remote entity and control it via subsequent events.

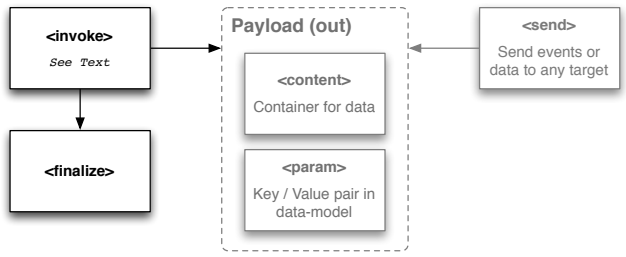


Figure 4.7.: Instantiating and communicating with external systems.

The following attributes and child elements are available for the `<invoke>` element:

- The entity to invoke is specified via the `type` attribute or its dynamic variant `typeexpr`. The SCXML standard mandates for a compliant interpreter to support the `http://www.w3.org/TR/scxml/` type as a nested state-chart and our implementation features many more (see section 7.2).

- The `src` and `srcexpr` attribute allows to specify an URI with a document to pass to the invoked system. E.g. for an invoker of type `xhtml`, this would be a URL to an Extensible HyperText Markup Language (XHTML) document. This attribute is mutually exclusive with a `<content>` child element to specify data with the same responsibility inline.

- As with the `<send>` element above, the `id` and `idlocation` attributes will allow to manually or automatically assign an identifier to an invoked component. This is allows a runtime to communicate with the invoked component by sending messages with a target of `#_[invokeid]`.

- Also similar to the `<send>` element is the specification of the additional payload to include in an invocation per `<param>` child element or `namelist` attribute. Unlike the `<send>` element however, the `<content>` child element is reserved to pass a document to the invoked component (see `src` above) and the `namelist` and `<param>` attribute are mutually exclusive. Though, it is perfectly valid to specify `<content>` and either `namelist` or `<param>`. The semantics of passing such additional data depend on the type of entity invoked. For nested SCXML interpreters, given values are to be copied at the respective location in the new data-model.

- Finally, the `autoforward` attribute is available as a convenience feature to forward all external events received in the invoking state-chart into the invoked component. For invoked state-charts (type `scxml`), the events are to be copied with all their fields and delivered to the invoked state-chart's external queue.

As a companion child element to `<invoke>`, the `<finalize>` element contains an executable content processed whenever the invoked component returns an event into the invoking state-chart (bound to the `_event` system variable). For the `scxml` invoker, this is the case, whenever the invoked state-chart addresses the `#_parent` as the target of a `<send>` element. It allows the invoking state-chart to update its data-model and react to events specifically originating from an invoked component.

4.1.5 Automatic Events

Typically, the majority of events to be processed by the state-chart are delivered from I/O processors, invoked components or `<send>`/`<raise>` elements. There are, however, some events a compliant interpreter is to deliver automatically in some situations.

Done Events

Whenever the interpreter enters a `<final>` state of a compound state c, it will emit an event named `done.state.[c.ID]`. Here, `<final>` is a valid child of any compound state and may optionally contain a `<donedata>` element with additional payload for the done event specified via a set of `<param>` or a `<content>` child element (see figure 4.8). If the respective compound state is, in turn, contained in a `<parallel>` state p and the last of p's child states to enter a final state, the interpreter will subsequently also emit `done.state.[p.ID]` to signal the completion of the complete parallel state hierarchy. It is not possible to attach any payload to done events raised for the completion of a `<parallel>` state. This will allow to join all concurrent, compound states contained by waiting for the completion of each before exiting the parallel state.

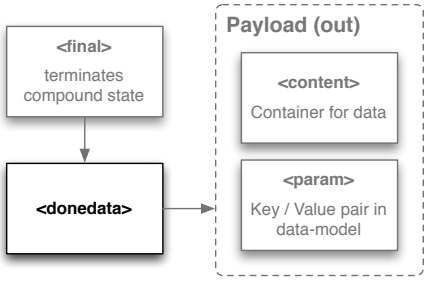


Figure 4.8.: Final states with `<donedata>`.

Another instance, when the platform will raise a done event is to indicate that an invoked component is finished. This event has a name of `done.invoke.[invoke.ID]` with no additional payload and will only be generated if the invoked component terminates on its own before being cancelled.

Error Events

Whenever the interpreter encounters an error at runtime it will emit an `error.communication`, `error.execution` or `error.platform` event. The situations in which a compliant interpreter is to emit these are listed in table 4.4. They will play an important role later when we describe our approach to formally verify a state-chart via temporal logic in section 6.5 as we explicitly have to exclude their occurrence.

Error Event	**Context**	**Reason**
`error.communication`	`<send>`	I/O processor generally unable to dispatch message. Inaccessible SCXML session with `_scxml_[sessionid]` target. `scxml` I/O processor cannot decode received message. Invalid or missing `target` with `basichttp` I/O processor.
`error.execution`	data-model statements	Invalid syntax or semantic error (e.g. scalar in `<foreach>` array). Attempt to modify a system variable.
	event reception	Inexpressible event data.
	`<send>`	`target` attribute is not supported by selected I/O processor. Unknown I/O processor selected via `type` attribute.
`error.platform`	any	Reserved for platform and application specific errors.

Table 4.4.: Error events with the context and reasons why they are emitted.

4.2 Application Specific Instantiation

If we are to use the interpretation of SCXML documents to control an application, we need to assign semantics to its observable behavior. The only language features of SCXML documents that will necessarily lead to externally

observable behavior are `<send>` requests with a respective I/O processor (e.g. `basichttp`). However, opening an HTTP server socket and waiting for events encoded in requests is hardly a convenient approach to control an application.

There are five principal means to connect the interpretation of an SCXML state-chart to the control of an application. The first, to provide callbacks, is something most implementations already offer despite not being mentioned in the standard. Within the standard, there are four more extension points, all of which will be discussed below.

- **Interpreter Callbacks**
  There are some obvious candidates of behavior common to all compliant SCXML interpreters that could be made observable by providing the means for an embedding application to register callback functions. We defined an open set of such callbacks in [RSWS14], where it was used to implement an on-line debugger for SCXML documents. It is reprinted in table 4.5.

Callback	Arguments	Context
`beforeProcessingEvent`	`Event event`	An event was just popped from either the internal or the external queue.
`beforeMicroStep`		The optimal enabled set of transitions was identified and we are about to perform a micro-step.
`beforeExitingState`	`Element state` `bool moreComing`	The set of transitions for the micro-step will cause the given state to be exited and its `<onexit>` handlers will be called.
`beforeExecutingContent`	`Element element`	The interpreter is about to process the given element as executable content.
`afterExecutingContent`	`Element element`	Finished processing the executable content.
`afterExitingState`	`Element state` `bool moreComing`	The given state was just exited.
`beforeUninvoking`	`Element invoke` `string invokeid`	A state we left contained an `<invoke>` element which will have to be cancelled.
`afterUninvoking`	`Element invoke` `string invokeid`	An `<invoke>` was cancelled as its containing state is no longer in the active configuration.
`beforeTakingTransition`	`Element transition` `bool moreComing`	States were exited and the interpreter is about to process the executable content contained in the given `<transition>` element.
`afterTakingTransition`	`Element transition` `bool moreComing`	Interpreter finished processing executable content of `<transition>`.
`beforeEnteringState`	`Element state` `bool moreComing`	Interpreter is about to add the given state to the active configuration and process its `<onentry>` handlers.
`afterEnteringState`	`Element state` `bool moreComing`	State is now contained in the active configuration and the executable content in its `<onentry>` handlers were processed.
`beforeInvoking`	`Element invokeElem` `string invokeid`	Interpreter entered a state with an `<invoke>` element and is about to instantiate the respective invoker.
`afterInvoking`	`Element invokeElem` `string invokeid`	Invocation of given `<invoke>` element is complete.
`afterMicroStep`		A micro-step was completed. Continue to check if more spontaneous transitions are enabled or if there are events in the internal queue remaining.
`onStableConfiguration`		Interpreter reached a stable configuration and is waiting for external events. s
`beforeCompletion`		A top-level final state was entered and the interpretation is about to end.
`afterCompletion`		Processing of the complete SCXML document is finished, tidy up and terminate.

Table 4.5.: Open set of SCXML interpreter callbacks, coarsely sorted by their order of execution in response to an event.

With these callbacks, already a wide range of behavior of a state-chart under interpretation can be assigned semantics for an actual application. There are some ambitions within the SCXML community to agree upon a set of these callbacks within a general Interface Description Language (IDL) for SCXML interpreters, but no W3C publication with a standardized approach exists as of yet.

Within the actual SCXML standard, there are four general extension points for interpreters and all can principally be used to control an application or connect to business logic.

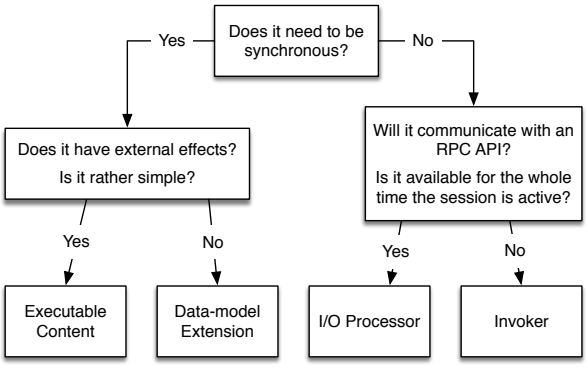


Figure 4.9.: Decision tree to decide upon an extension point for an SCXML interpreter platform.

- **New I/O Processor Types**
  When an interpreter encounters the `<send>` element as part of executable content, it will dispatch its attributes and child elements to an I/O processor specified by the `type` attribute to be encoded into a message. Here, a platform can define additional types and an application developer can be enabled to register respective I/O processor instances with the interpreter. The life-cycle of these instances is not defined, however, it is assumed that they are available whenever a respective `<send>` element is processed, implying that they are either instantiated ad-hoc and terminated upon completion or surpass the life-cycle of the containing interpreter.

- **New Invoker Types**
  As with `<send>` above, new types can also be introduced for entities to be `<invoke>`d. In contrast to I/O processors though, their life-cycle is determined by the interpreter's active configuration: As long as the state containing an `<invoke>` element is in the active configuration, the invoked component will persist. If the respective state is exited, the invoked component is terminated.

  A state-chart can `<send>` messages to such an invoked component via the `scxml` I/O processor (default when omitting the `type` attribute) by addressing `#_<invokeid>` via the `target` attribute. This ability to instantiate and communicate with any entity for which a respective type id defined makes this extension point more versatile than additional I/O processors. In our implementation, we defined additional invoker types even for entities which would originally be considered I/O processors, e.g. subscriptions to topics in a publish/subscribe are realized via invokers as it allows us to finely control their scope (see section 7.2).

- **Custom Executable Content**
  The standard allows to introduce arbitrary new XML elements in their respective namespaces as executable content. These are processed just as native executable content is processed and are subject to the same rules that apply, e.g. they can be conditionalized via `<if>` or iterated with `<foreach>`. Providing a new element will, usually, cause the interpreters control flow to be diverted into user-supplied functionality whenever the respective element is *opened* and *closed*. The semantics about how to process the children of a user supplied element are undefined. In our implementation we allow such an element to explicitly ask the interpreter to process its children as nested executable content or to ignore it.

- **Data-Model Extensions**
  Extending the data-model by introducing e.g. application specific functionality and data to statements in the data-model's language is another possibility. However, one has to be careful: While introducing functions to directly control the application via the data-model might be unproblematic, exposing information from the application in the data-model might lead to unintuitive behavior as this information becomes part of the interpreter's extended state and validates the assumption that events are the source of all state changes.

During the open panel discussion of the SCXML workshop in 2014 [RSWL+14], the possible extension points above were discussed and a simple decision tree was developed to decide, which kind of extension is most suited for which requirements (figure 4.9).

In our experience, the extension via specific invokers provides the most versatile approach as the invoked component can be instantiated and terminated upon request with communication via `<send>` in between. The versatility of the approach is evidenced by the plethora of invokers available in our implementation (see section 7.2).

The semantics of SCXML state-charts are given in three different forms: 1. as an informal set of normative, functional requirements in the actual text of the specification, 2. as an algorithm for SCXML interpretation in pseudo-code and 3. as a set of 233 non-conclusive tests of the functional requirements. From a puristic point of view, none of those constitute a formal specification in a strict, mathematical sense. The text of the specification does follow RFC2119 [Bra97] with regard to the meaning of certain *requirement level indicators* such as `must`, `must not`, `shall` or `may` but the actual requirements are not formalized beyond precise prose form. The algorithm is described in pseudo-code for which no formal semantics are given either. Though, the pseudo-code does establish sets and relations and it might be argued that it, therefore, constitutes an algebraic formalization. Finally, the 233 tests defined as part of the Implementation Report Plan (IRP) for SCXML are only a non-conclusive enumeration of valid behavior and not exhaustive. However, all three sources combined provide a rather solid basis to decide upon the behavior of a compliant SCXML interpreter.

The SCXML semantics are generally aligned with the original semantics by Harel and Pnueli but differ considerably in some specific areas. The principal mode of operation, assuming an interpreter is in a stable configuration, is as follows:

1. The interpreter will remain idle until an external event can be dequeued which enables transitions.
2. For the *optimal* subset of transitions, perform a *micro-step*: (i) leave all states from its *exit set* in document order, (ii) perform the contained transitions and (iii) enter the states from its *entry set*.
3. Step 2 is repeated with eventual *eventless* (spontaneous) transitions until they are exhausted.
4. If these activities caused any internal events, steps 2 and 3 are repeated with transitions enabled by internal events until no more internal events are enqueued.
5. Goto step 1 and wait or process the next external event enqueued.

As such, there are three interleaved processing cycles, each performing a micro-step for a set of transitions (see figure 4.10): 1. The spontaneous loop will perform a micro-step for eventless transitions, 2. the internal loop for transitions caused by internal events and 3. the external loop for transitions caused by external events.

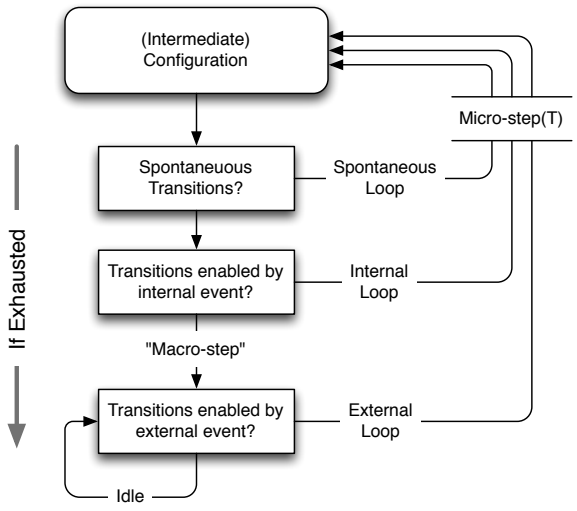


Figure 4.10.: The three interleaved processing cycles for transitions.

Whenever the external loop processed one iteration, the interpreter is said to have processed a *macro-step* and to be in a *stable* configuration. To reach its first stable configuration at start-up, we can think of the interpreter as initially having the empty configuration and processing a micro-step for a single transition with an empty exit set, no executable content and the interpreter's topmost `<scxml>` element in the transitions entry set. This will cause the interpreter to enter the default *completion* of the root state and, thereby, possibly enable spontaneous transitions or raise internal events, which will have to be exhausted before the actual, initial stable configuration is assumed. There

are some more tasks to be performed prior to this initial micro-step but for the formalization of the semantics for transitioning through configurations, the notion holds and we can always assume to start from a stable configuration.

With this intuition, we can introduce the semantics more formally. The most important operation to transition through the state-chart's configuration is a *micro-step over an optimal transition set*, each of which will cause the processing of executable content and put the interpreter in a follow-up configuration. To generalize the behavior of the three loops, it is helpful to introduce the ϵ event which has no name, will never match any event descriptors and only enables spontaneous transitions without an `event` attribute. Now, every event will implicitly establish a subset hierarchy of transitions in a configuration:

- **Active Transitions**
  All `<transition>` elements contained as direct children of states in the active configurations are said to be *active*. They form the superset of all other transition sets below.

  Definition 7 (Active Transition): *A transition t in state s is active if s is in the state-chart's active configuration.*

$\supseteq$ **Matched Transitions**
The subset of active transitions, with at least one event descriptor in their `event` attribute matching the current event's name are said to be *matching*. If the current event is the ϵ event, all active transitions with no `event` attribute (spontaneous transitions) are matched.

Definition 8 (Matched Transition): *A transition t is matches by event e if*

1. *t is active and*
2. *the event descriptor of t matches the name of e.*

A transition t is matched by the ϵ event if

1. *t is active and*
2. *t lacks an `event` attribute.*

Definition 9 (Matching Event Descriptors): *A transition's event descriptor d matches an event's name e if*

1. *d is equals e and*
2. *d with an appended dot is a prefix of e.*

It is allowed for an event descriptor to have a `.*` suffix for compatibility with Call Control eXtensible Markup Language (CCXML), this suffix is to be removed to establish d.

It is legal for a transition to specify multiple, space separated event descriptors in their `event` attribute. In this, case an event matches a transition if one of the transition's event descriptors matches the event's name.

$\supseteq$ **Enabled Transitions**
The set of matching transitions is further reduced by requiring an eventual `cond` attribute to evaluate to `true` on the data-model.

Definition 10 (Enabled Transition): *A transition t is enabled by an event e in atomic state s if*

1. *t is matching and*
2. *t lacks a 'cond' attribute or its 'cond' attribute evaluates to "true".*

$\supseteq$ **Optimally Enabled Transitions**

Definition 11 (Optimally Enabled Transition): *A transition t is optimally enabled by event e in atomic state s if*

1. *t is enabled by e in s and*
2. *no transition that precedes t in document order in t's source state is enabled by e in s and*
3. *no transition is enabled by e in s in any descendant of t's source state.*

For a transition to be optimally enabled, there can be no earlier enabled transition in the same state, neither can a transition in a descendant state be enabled. The first criterion provides an ordering for enabled transitions within the same state. The second criterion allows to *specialize* a state-chart's behavior in response to events by overriding behavior in a more deeply nested sub-state of a composite state.

⊇ **Optimal Transition Set**
Generally, it is not possible for all optimally enabled transitions to be taken in the same micro-step as they might lead to an invalid subsequent configuration. Therefore, the optimal transition set is established as the largest set of non-conflicting, optimally enabled transitions.

> **Definition 12 (Optimal Transition Set):** *The optimal transition set enabled by event e in state configuration C is the largest set of transitions such that*
>
> 1. *each transition in the set is optimally enabled by e in an atomic state in C and*
> 2. *no transition conflicts with another transition in the set and*
> 3. *there is no optimally enabled transition outside the set that has a higher priority than some member of the set.*

> **Definition 13 (Transition Priority):** *Let t_1 be optimally enabled in atomic state s_1, and t_2 optimally enabled in atomic state s_2, where s_1 and s_2 are both active. We say that t_1 has a higher priority than t_2 if*
>
> 1. *t_1's source state is a descendant of t_2's source state, or*
> 2. *s_1 precedes s_2 in document order.*

An enabled transition t is *optimally enabled* if there is no enabled transition in the descendant states of t's source with an overlapping exit set. The requirement of non-overlapping exit sets for all transitions in the optimally enabled transition set will ensure that the follow-up configuration of a macro-step is legal (from the SCXML specification):

> *"Loosely speaking, transitions are compatible when each one is contained within a single `<state>` child of the `<parallel>` element. Transitions that aren't contained within a single child force the state machine to leave the `<parallel>` ancestor (even if they reenter it later). Such transitions conflict with each other, and with transitions that remain within a single `<state>` child, in that they may have targets that cannot be simultaneously active. The test that transitions have non-intersecting exit sets captures this requirement. (If the intersection is null, the source and targets of the two transitions are contained in separate `<state>` descendants of `<parallel>`. If intersection is non-null, then at least one of the transitions is exiting the `<parallel>`).".*

We will more formally define a transition's *exit set* below.

Pragmatically, a compliant interpreter would not establish all these sets explicitly. It is sufficient to establish the set of transitions enabled by the atomic states in the state-chart's configuration by iterating their transitions in document order, traversing via the states' parents toward the root and selecting the first enabled transition. This will already ensure criteria 1 and 3 from definition 11. It might still be the case, however, that a parallel state was traversed on the path to the root with an enabled transition in another child state. This would invalidate criteria 2 of definition 11 and needs to be explicitly checked for. If we iterated the atomic states in document order, they will already be sorted with regard to their priority and we only need to remove those transitions that conflict with earlier transitions in the set to form the optimal transition set for a macro-step.

As such, all three loops above merely establish the optimal transition set for different events and process them in a micro-step:

1. The spontaneous loop exhausts optimally enabled transitions caused by the ϵ event. For each such set of spontaneous transitions, a micro-step is performed and the process repeated for the follow-up configuration. When no more such transitions exist, the next iteration of the internal loop is started.

2. The internal loop will dequeue an internal event and perform a micro-step for transitions optimally enabled by its name. After which it will execute the spontaneous loops again. If there are no more internal events to be dequeued, the interpreter is said to have performed a macro-step and the external loop is started.

3. The external loop acts exactly as the internal loop but attempts to dequeue an event from the external queue and performs a micro-step on its optimally enabled transitions. Afterwards, it will also call the spontaneous loop which, in turn, will call the internal loop until all spontaneous events and eventual internal events are exhausted. If there is no event to be dequeued, it will block until an event is delivered.

Within a micro-step, the interpreter will:

1. Exit all states from the union of exit sets (defined below) of transitions in the optimal transition set in document order. This will call the states' `<onexit>` handlers and subsequently remove the respective state from the state-chart's configuration.

2. Process all executable content of transitions in the optimal transition set in document order.

3. Enter all state from the union of entry sets (again, defined below) of transitions in the optimal transition set in document order. This will add the respective state to the state-chart's configuration and subsequently call the states' `<onentry>` handlers.

Note that a state's `<onentry>` handlers are processed when the state is already in the current configuration, whereas a state's `<onexit>` handlers are processed prior to removing the respective state from the configuration.

A sequence of such micro-steps up until the next external event is dequeued is called a macro-step. Only after a macro-step is performed are `<invoke>` elements processed by instantiating invokers contained in states new to the configuration and canceling invokers from removed states. As such, an interpreter may exit and enter states containing invokers within a series of micro-steps, but respective changes to the invokers are only regarded after a macro-step was performed. A more detailed flow-chart of the interpreter's overall processing cycle is given in figure 4.11.

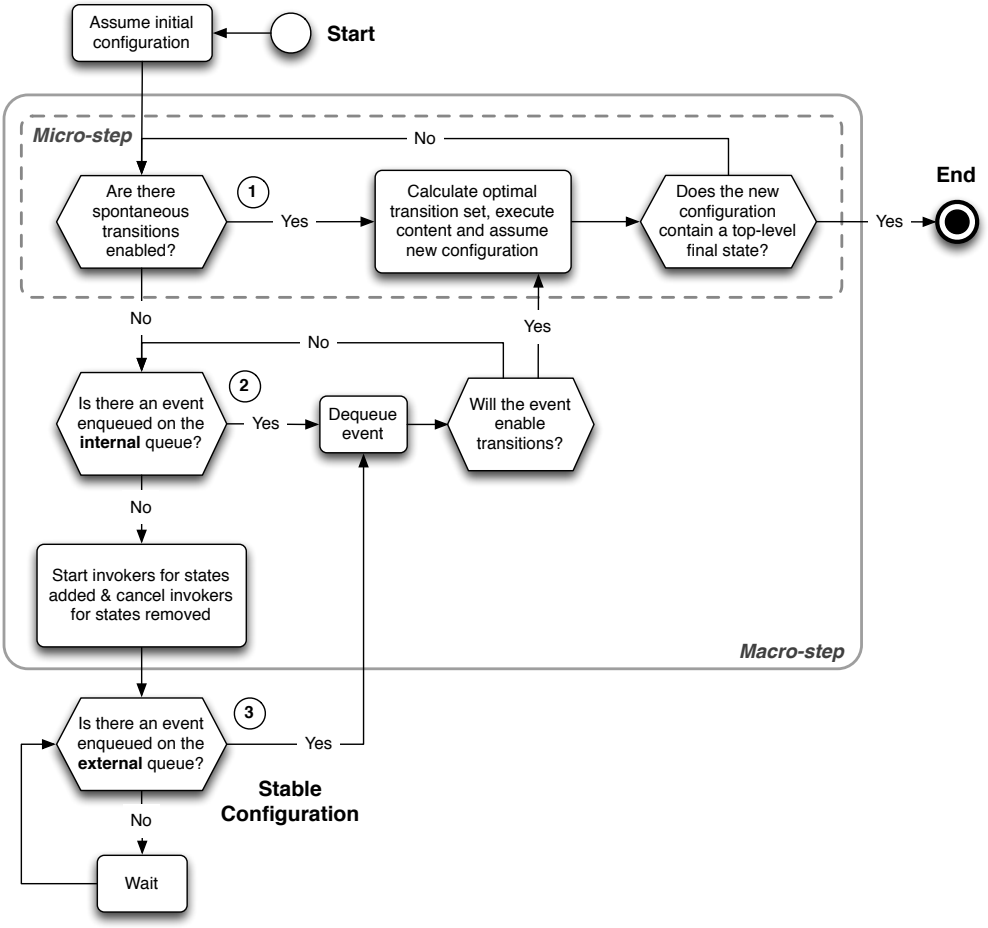


Figure 4.11.: Event processing and transition selection in an SCXML document.

4.3.1 The Completion of a State

Whenever a state s is entered, it will cause an interpreter to recursively enter other states as the *completion* of s:

- If s is a `<history>` pseudo-state, all states active when the containing state was last left are entered as well, with the history's type of `shallow` or `deep` denoting the nesting depth of s's completion. If the containing state was never left before, an eventual `<transition>` element in s is regarded or, if no such transition is given, the containing state's initial state is entered.

- If s is a compound state, its initial state is entered. A compound state's initial state is either (i) specified in an `initial` attribute of s,(ii) specified via a `<transition>` element in an `<initial>` child element of s (not unlike with history states above) or (iii) the first proper child state of s in document order.

- If s is a parallel state, the completion of all its child states is entered.

- Entering a state s will also enter all its ancestor states with the same considerations.

4.3.2 Domain of a Transition

A transition's *domain* is the most deeply nested state that contains all of the transition's changes to the configuration. As such, it is not unlike the *arena* of a transition in the formalization of Pnueli [PS91].

The domain depends on the transition's `type` and `target` attribute: If the transition has an empty or missing `target` attribute, its domain is not defined. If the type of a transition is `internal` and all of the transition's target states are descendants of the transition's source state and the transition's source state is compound, then the domain of the transition is its source state. If the type of a transition is `external` (the default), its domain is the *least common [proper] compound ancestor* (LCCA) of the transition's source state and its target states. I.e. the most deeply nested compound state (`<state>` or `<scxml>`) that contains all the transition's target states and the parent (as the first proper ancestor) of the transition's source state.

4.3.3 The Exit Set of a Transition

When a transition is in the optimal transition set and subsequently taken during a micro-step, the interpreter will start the step by exiting the set of states in the transition's *exit set*. The union of all optimal transitions' exit sets defines the set of all states exited within a micro-step.

The exit set of a transition without a `target` attribute is empty. Otherwise, let S_{exit} be the set of all proper descendant states of the transition's domain. Now, a transition's exit set is the intersection of S_{exit} and the states in the state-chart's active configuration; it contains all the states that will be exited by a transition.

4.3.4 The Entry Set of a Transition

After the states contained in the exit sets of transitions in the optimal transition set are left and the transitions' eventual executable content is processed during a micro-step, a new configuration is assumed by entering the states in the transitions' *entry sets*. The union of all optimal transitions' entry sets defines the set of all states entered within a micro-step.

The entry set of a transition without a `target` attribute is empty. Otherwise, let S_{entry} be the completion of all states given in the `target` attribute of a transition t. The subset of states in S_{entry} that are not already active or are in an *exit set* of another transition from the optimal transition set is referred to as the transition's *entry set*. In other words, this set contains all states that will be entered (not already active) or reentered (in another exit set) by t.

4.3.5 Examples

In this section we will discuss some examples of SCXML documents, especially with regard to the various sets associated with transitions introduced above. In all examples, the exit- and entry sets of a transition are given as S_{exit} and S_{entry} respectively, because their actual exit- and entry sets would depend on the current configuration. Furthermore, a transition's exit set is given only as the topmost states exited by the transition. Per definition above, all their children are also included the exit set but omitted for brevity.

Internal and Targetless Transitions

```
1  <scxml>
2    <state id="s1">
3      <transition target="s1.s2"
4                  exitset="s1"
5                  entryset="s1.s2, s1"
6                  domain="/scxml[1]" />
7      <transition target="s1.s2"
8                  type="internal"
9                  entryset="s1.s2"
10                 domain="s1"
11                 exitset="s1.s1, s1.s2" />
12     <transition exitset=""
13                 entryset=""
14                 domain="#UNDEF"/>
15     <state id="s1.s1"/>
16     <state id="s1.s2"/>
17   </state>
18 </scxml>
```

Normal, *external* transition. Its domain is the `<scxml>` element as the LCCA of the transition's source state `s1` and its target state `s1.s2`.

Internal transition in a compound state with all its target states as descendants of its source state and, therefore, its source state as its domain.

Targetless transition with undefined domain.

Listing 4.1: Difference in exit-, entry set and domain for internal and external transitions.

In listing 4.1 an SCXML state-chart with a compound state `s1` and two nested states `s1.s1` and `s1.s2` is given. Three transitions originating in the compound state are annotated with regard to their domain, exit sets and entry sets.

The first transition (line 3) is a normal transition with a target state contained within the compound. Its domain is the topmost SCXML element as the most deeply nested compound state as a proper ancestor of the transitions source state `s1` and its target state `s1.s1` (the LCCA). When this transition is in the optimal transition set during a micro-step, it will cause an interpreter to first exit `s1` and all its child states before both, `s1` and `s1.s2` are entered. This is in contrast to the internal transition in the same source state and with the same target state (line 7). Its domain is its source state `s1` and its exit set are all proper child states of `s1`. It will not exit the compound state `s1` as only the proper children of the domain are exited. Finally, the third transition is targetless with an undefined domain and no states to enter or exit. None of these transitions can occur concurrently within an optimal transition set as they share a common source state.

Conflicting Transitions

```
1  <scxml>
2    <parallel id="p">
3      <state id="foo">
4        <transition event="e1"
5                    target="foo.2"
6                    priority="2"
7                    conflicts="1, 0"
8                    exitset="p"
9                    domain="/scxml[1]" />
10       <transition event="e1"
11                   target="foo.2"
12                   type="internal"
13                   priority="1"
14                   conflicts="2"
15                   exitset="foo.1, foo.2"
16                   domain="foo" />
17       <state id="foo.1"/>
18       <state id="foo.2"/>
19     </state>
20
21     <state id="bar">
22       <transition event="e2"
23                   target="bar.2"
24                   type="internal"
25                   priority="0"
26                   conflicts="2"
27                   exitset="bar.1, bar.2"
28                   domain="bar" />
29       <state id="bar.1"/>
30       <state id="bar.2"/>
31     </state>
32   </parallel>
33 </scxml>
```

Normal, *external* transition. Its domain is the `<scxml>` element as the LCCA of the transition's source state `foo.2`. This transition conflicts with any of the other transitions as it will leave the complete parallel state.

Transition internal to state `foo`. Its domain is its source state as it is a compound state and all the transitions target states are proper descendants. It may form an optimal transition set with the other internal transition in state `bar` as their exit sets do not overlap.

Transition internal to state `bar`. This transition may occur with the internal transition from above in an optimal transition set.

Listing 4.2: Conflicting and non-conflicting transitions.

Listing 4.2 shows a parallel state with two compound states contained. Having a parallel state will, potentially, cause the optimal transition set to contain multiple transitions as multiple atomic states may optimally enable different, non-conflicting transitions. There are three transitions in total, one external transition in the first compound state (line 4) and two internal transition, one in each compound (line 10 and 22). The external transition is not unlike the external transition from listing 4.1, with its domain again being the topmost root state. Here, the containing `<parallel>` state is not eligible to be the LCCA (and therefore the domain of the external transition) for not being a *compound* state. The other two transitions are internal transitions with their domains contained within their respective compound source states. With the exit set as all proper children of the domain, these two transitions are not in conflict as their exit sets are completely contained within their compound source states.

4.4 Static Analysis and Syntactical Validation

Apart from a complete and deterministic description of the execution semantics of state-charts, another important design goal of the SCXML specification was to enable static analyzability of respective state-chart's. E.g. the absence of dynamic targets for transitions or dynamic event descriptors allow to identify unreachable states. There are many syntactic and even semantic validity criteria that can be analyzed on a state-chart even without subjecting it to interpretation. In our implementation, an instantiated interpreter object will return a list of issues when its `validate` method is called. A complete list of the validity criteria that are checked for is given in table 4.6. Some of these are mandated by the SCXML standard itself, others were supplied by industrial partners and users to alert about common pitfalls. Some of these criteria are also checked via the official XML schema definition accompanying the standard.

Criteria	Remark
Syntactic Validity	
Document contains at least one `<scxml>` element.	Multiple root states may exist within an SCXML document when invocable state-charts are specified with in-line `<content>`.
Every element from the SCXML standard is a child of a valid parent.	Validating a documents structure via the *valid-parent-of* relation allows for more flexibility with experimental elements while ensuring that no standard element is misplaced.
All the elements' required attributes exist are non-empty.	
Constraints with regard to multiplicity, mutual exclusion and alternatives for attributes and elements are honored.	Some attributes of an element are mutually exclusive or may not occur in conjunction with a specific child element (e.g. `initial` attribute and an `<initial>` child for a compound state). In other cases, a given child element may occur only once or exactly once (e.g. a `<transition>` in a `<initial>` element).
No two states have the same `id` attribute.	It is actually not required for a state to have an identifier, but if one is given, it has to be unique.
Semantic Validity	
All states are reachable	We can actually only define a superset of reachable states (see section 5.3.8 for algorithm).
All state references exist as states	
Composite states with a `<history>` element have multiple legal configurations.	Introducing a `<history>` pseudo-state for a state hierarchy that can only form a single legal configuration is useless.
State reference constraints are honored	E.g. the target states of an `initial` attribute or its `<transition>` child element have to refer to sub-states contained in the parent state.
All states in a target state specification can be part of a legal configuration	If an element or attribute specifies a set of target states, e.g. via the `target` attribute of a transition or a specification of initial states, the respective set must have a legal completion as their eventual configuration.
All type of invokers, I/O processors, custom executable content and the data-model is known to the platform.	For I/O processors and invokers, this can only be validated if the type is given as a literal in the `type` attribute of `<send>` and `<invoke>` respectively.
All expressions and statements of the data-model have a valid syntax.	This can only be checked with data-models whose implementation allows to validate a given string syntactically.

Table 4.6.: Validity criteria checked for when calling `validate` on an interpreter instance in our `uSCXML` implementation.

In addition to the issues regarding these validity criteria, another common problem with SCXML state-charts are endless spontaneous transitions within a micro-step: If a given (intermediate) configuration enables a set of spontaneous transitions which lead into the very same configuration, the interpreter will be stuck in an endless

loop. Such a loop can, in general, not be recognized via static analysis as the constituting transitions can still be conditionalized with data-model expressions. As such, our implementation provides a callback to signal an application at runtime that the interpreter might currently iterate a sequence of configuration endlessly within a micro-step. This is done by storing every intermediate configuration and comparing every new configuration to those that were already assumed.

4.5 Extensions for Rule-based Dialog Management

In SCXML, transitions are guarded by boolean expressions of the data-model. Usually, these are simple expressions such as comparisons of variables in the data-model with some event related value. However, when we introduce a suitable data-model, the condition can, effectively, represent the prerequisite condition in a rule-based dialog management technique. To this effect, we implemented the Prolog data-model where boolean expressions can be logical queries evaluating to `true` if there is a solution for the given query. This allows us to actually embed the complete TrindiKit system [LT00] to perform Information State Update (ISU)-based dialog management with SCXML. The general idea was already suggested by Kronlid and Lager [KL07], but their approach remained sketchy. Our approach concretizes the semantics of a suitable Prolog data-model and actually provides an implementation via the SWI Prolog interpreter[2].

4.5.1 The Prolog Data-Model

Pertaining to the specific responsibilities of a data-model introduced via the respective API in our implementation (table 4.2), the following list will outline the most important considerations to map the Prolog semantics onto an SCXML data-model:

- **Representing the `_event`** (setEvent):

  Whenever an event is dequeued, it is required to be available as a compound data structure named `_event` in the data-model. In our implementation of the Prolog data-model, the event is represented as a series of facts in the predicate `event`. Unfortunately, Prolog will not allow to establish facts beginning with an underscore; all facts have to begin with a lowercase alphabetic character. As such, we assert all the event's compound fields in the predicate `event/1`:

```
1 event(name('foo')).
2 event(type('external')).
3 event(sendid('s1.bar')).
4 event(origin('http://host/path/basichttp')).
5 event(origintype('http://www.w3.org/TR/scxml/#BasicHTTPEventProcessor')).
6 event(data(...)).
7 event(param(...)).
```

Listing 4.3: Example facts for event/1 (from [RSWRA13]).

  This allows accessing the event's individual fields by simple queries such as `event(name(X))`, which will resolve `X` to the event's name `foo`. Now, in Prolog, it is perfectly legal to have many facts with the same name, that is, `event(name(X))` may resolve to many solutions, which is clearly undesired as the previous names are no longer given. Therefore, whenever a new event is about to be represented in the data-model, all old facts about `event/1` are retracted and reasserted with regard to the new event.

- **Initializing Data and Assignments** (init, assign):

  Just as with the ECMAScript data-model, we allow string literals, JSON and XML as the text-child of either a `<data>` or an `<assign>` element. In each case, the given data-model location is asserted as a predicate and available to logical queries. In the case of `<data>`, the given structure is simply asserted as a compound predicate (listing 4.4) similar to `_event` above, though for XML and JSON, the SWI/Prolog modules `sgml` and `http/json` are used (listing 4.5 and 4.6 respectively). The same is true for `<assign>`, but here we allow for an additional attribute `type` with valid values of (i) `append` to assert the structure's facts at the end of eventual existing facts with the same name, (ii) `prepend` to add the facts at the end of Prolog's knowledge base and (iii) `retract` to remove any existing facts with the same name prior to establishing the new facts.

[2] `http://www.swi-prolog.org`

```
<data id="father">
  bob, jim.
  bob, john.
</data>
```

```
father(bob, jim).
father(bob, john).
```

Listing 4.4: Asserting simple facts from a <data> element.

```
<data id="childs">
  <child name="john" father="bob" />
  <child name="jim" father="bob" />
</data>
```

```
childs([
 element(child,
   [father=bob, name=john], []),
 element(child,
   [father=bob, name=jim], [])]).
```

Listing 4.5: Asserting facts from a <data> element with XML content.

```
<data id="household">
  {
    name: "The Bobsons",
    members: ['bob', 'martha', 'jim', 'john']
  }
</data>
```

```
household({
  name:'The Bobsons',
  members:[bob, martha, jim, john]}).
```

Listing 4.6: Asserting facts from a <data> element with JSON content.

Furthermore, if the `location` attribute with `<assign>` or the `id` attribute of `<data>` are omitted, complete Prolog knowledge bases as declaration of facts can be loaded from an URL via the `src` attribute or specified as an inline text child element. This allows considerable flexibility to work with data as facts and is very much aligned with the original semantics of both, the `<data>` and the `<assign>` element.

- **Iterating with `<foreach>`** (getLength, setForeach):

  Iterating a list, via an index and an ephemeral item is a fundamental concept with imperative languages, but somewhat awkward with functional or logic programming languages. The semantics we choose for `<foreach>` is to iterate the solutions to a logical query given in the `array` attribute (listing 4.7)

```
<foreach array="father(bob, X)"
         item="child"
         index="index">
  <log label="child" expr="child(X)" />
  <log label="index" expr="index(X)" />
</foreach>
```

```
child: jim
index: 0
child: john
index: 1
child: jack
index: 2
```

Listing 4.7: Foreach expression (left) and respective output (right).

- **Evaluating Expressions** (eval*):

  There are three situations wherein the data-model is tasked to evaluate expressions specified in an SCXML document:

  – Statements specified via a `<script>` element are either evaluated in *query* mode or as external Prolog files, depending on the value of the `<script>` element's `type` attribute specific to the Prolog data-model in our implementation. In query-mode, the statements are interpreted as if they were provided at an interactive Prolog console, otherwise as if they were loaded from a file. The semantics differ slightly as the assumption is that external files will establish facts and rules prior to a user posing logical queries. As such, e.g. introducing new facts in query mode necessitates to `assert` them, along with a few other differences in semantics. These two modes are inherent to the Prolog language and the `type` attribute allows to select the mode desired.

  – For some SCXML language features, it is necessary to evaluate an expression as a string literal, e.g. the `expr` attribute of a `<log>` element. Here, we allow to specify a logical query with a single free variable that is required to resolve to an atomic term and attempt to interpret its last solution as a string literal.

  – Evaluating a boolean expression, e.g. for the `cond` attribute of a `<transition>` element, requires the expression to be a logical query and it will evaluate to `true` if there is at least one solution.

- **Export `Data` Objects** (getStringAsData):

  Whenever data more elaborate than a string literal needs to be exported from the data-model, e.g. in order to

constitute the content of a message, the data-model is asked to return a `Data` object (see section 4.1.2). Here, our implementation allows for the specification of Prolog terms or logical queries and either represents their literal compound form or the compound form of their solutions a `Data` object.

These semantics allow for considerable flexibility and applicability of rule-based dialog management when employing Prolog as a data-model in SCXML documents. By relying on an established Prolog implementation (SWI Prolog), we offer the full wealth of Prolog via the `<script>` element. Indeed, we were able to load all of TrindiKit along with the GoDIS application [LE02] into SCXML, making all their facts and rules available for dialog management.

4.6 Applicability for Dialog Management in Pervasive Environments

There are two aspects when arguing for the applicability of SCXML to model dialogs in pervasive environments: (i) its applicability as an Interaction Manager (IM) in the context of the W3C Multimodal Dialog System (MDS) and (ii) its general applicability, unrestrained from any conventions and specifications but those of SCXML itself. While the implied claim of the W3C MDS is that the former is sufficient, this is not actually a limitation we have to uphold for our subsequent approach to formal verification with temporal logic. Arguing for SCXML as a suitable markup language to handle the responsibilities of an IM in the W3C MDS is more to establish relevance than sufficient expressiveness and, indeed, many concrete user interfaces may violate a puristic interpretation of the W3C MDS standards in favor of pragmatic solutions.

That is, even if we failed to sufficiently show applicability of the W3C MDS, we still can argue for the applicability of SCXML as such to uphold our claim for the central argument of this thesis. The following sections will elaborate on both aspects.

4.6.1 Instantiation in the W3C Multimodal Dialog System

The SCXML standard has no notion about any interaction specific responsibilities, nor does it make any reference to either the W3C Multimodal Interaction Framework (W3C MMI framework) or the W3C Multimodal Architecture and Interfaces (W3C MMI architecture) recommendations. However, there is a continuity with regard to the specification's authors and SCXML is, indeed, suggested as a suitable markup language to describe IMs in the latter recommendation.

As such, the question about how to actually realize the responsibilities of an IM with SCXML arises. We already introduced the general responsibilities of an IM as a component in both these recommendations and its relation to the Modality Components (MC) when we introduced the reference model in section 3.1.3. The complete set of requirements for the IM component is given in table 4.7, as all statements related to the IM component from both recommendations, organized by the general issue they address.

Statement	Document
Coordination of MC	
1. *The IM is the logical component that coordinates data and manages execution flow from various input and output MC interface objects.*	Framework
2. *The IM [..] manages [..] changes and coordinates input and output across component interface objects.*	Framework
3. *In multimodal applications, multiple user interface components are controlled and coordinated individually by the IM.*	Framework
4. *The IM [..] coordinates the different modalities. It is the Controller in the MVC paradigm.*	Architecture
5. *The IM [invokes] MCs and receives results from them.*	Architecture
State Management	
6. *The IM maintains the interaction state and context of the application.*	Framework
7. *[The IM] responds to inputs from component interface objects and changes in the system and environment.*	Framework
Composability	
8. *Different aspects of interaction management may be handled at different levels of the hierarchy.*	Framework
9. *Hierarchical interaction management also enables the delegation of complex input tasks to lower levels of the hierarchy.*	Framework
10. *MCs may contain their own IMs to handle their internal events.*	Architecture
Neighboring Systems	
11. *The multimodal interaction framework must allow the IM to determine what information is available, as this will be system dependent.*	Framework
12. *The IM is a client of the Data Component and is able to access and update it as part of its control flow logic.*	Architecture
Constraints	
13. *All life-cycle events that the MCs generate must be delivered to the IM.*	Architecture
14. *All life-cycle events that are delivered to MCs must be sent by the IM.*	Architecture
15. *If the IM does not contain an explicit handler for an event, it must respect any default behavior that has been established for the event. If there is no default behavior, the IM must ignore the event.*	Architecture
16. *All control data logically sent between MCs must flow through the IM.*	Architecture

Table 4.7.: Responsibilities of an Interaction Manager from statements in the "W3C Multimodal Interaction Framework" and "Multimodal Architecture and Interfaces" recommendations.

Some of these statement express, or at least imply, functional requirements for an IM component and, as such, a compliant SCXML interpreter in the role of an IM. In the case of nested IM/MC pairs (statement 10), SCXML can also assume the role of an MC, though the respective responsibilities can be reduced to "will react to control events" in the form of the life-cycle events from the architecture and application-specific messages. In the following, we will attempt to identify the functional requirements and describe how they can be realized in SCXML.

1. The general responsibilities with regard to coordinating MCs can be interpreted as the requirements to

   - Control their life-cycle (statements 3, 5):
     The obvious language feature available in SCXML to instantiate and terminate external entities is `<invoke>` and, in fact, the choice of the word *invoke* in statement 5 and the continuity of authors is a strong indication that this is, indeed, the intended mechanism.

     The life-cycle of an MC would be controlled by representing it as a type of invoker (e.g. `xhtml`, `voicexml`) and instantiate and cancel instances with the `<invoke>` element by entering and existing its parent state respectively.

     Not all life-cycle events from the W3C architecture have an obvious semantic equivalence to the `<invoke>` element, e.g. there is no notion to pause or resume an invoked component and it is up to the SCXML interpreter platform to ignore or employ them beneficially. E.g. by allowing for respective events to be sent explicitly or preparing and pausing an MC as soon as the invocation is *likely*.

   - Process the events received (statements 1 - 5):
     It is the responsibility of a respective invoker implementation in SCXML to deliver the events emitted by such an external component (as `ExtensionNotification` of `DoneNotification` life-cycle events) to the interpreter's external queue after being processed in an eventual, instance specific `<finalize>` block.

As such, events and attached data from an invoked MC is available via the usual event processing and `_event.origin` can be consulted to differentiate events by their source.

- Provide the events required (statements 1 - 4):
  The `<invoke>` element allow to specify an `id` or `idlocation` attribute (see above). The given, or generated identifier can, subsequently, be used in the `target` attribute of a `<send>` element to deliver arbitrary events and data to the invoked MC.

2. Maintain the state of the interaction (statements 6, 7):
   This responsibility is apparently native to SCXML, though, with *state* we will explicitly refer only to the interaction state, not the complete application state as this would imply to model all of the business logic of an actual application with the concepts of the W3C MDS, which is rather undesirable. But even with a notion of *state* restricted to interaction, we still have an issue as an SCXML interpreter's state is not solely defined by its active configuration, but the state of the employed data-model and the assignment of its variables as well.

   Here, a very practical issue arises: most embedded runtimes for e.g. ECMAScript will not support a serialization of their state and, as such, provide no means to *move* an interpreter in a distributed system. While this is not necessarily required per standard, it is unfortunate.

3. Allow for the assembly of IM hierarchies (statements 8 - 10):
   Hierarchies of nested IM/MC pairs can be established by simply invoking nested SCXML interpreters (`<invoke>` with type `scxml` as defined in the SCXML specification). The invoking interpreter can be addressed via the `#_parent` target in a `<send>` element, specific invoked child interpreters via their invocation identifier.

   While this will allow to assemble trees with a topmost, proper IM, branches as nested IM/MC pairs and leafs with proper MCs, there is also the possibility to address *any* other SCXML runtime if its session identifier (`_sessionid` data-model variable) is known by using the `#_scxml_[_sessionid]` target with the `<send>` element. Thus, even if the W3C architecture does imply a tree-like structure, this is no limitation inherent to SCXML.

4. Provide the means to coordinate with additional, external components (statements 11 - 12):
   This really is a question about which extension mechanism is suited to represent such external systems in the SCXML runtime (compare above extension points). When we are to represent the *System & Environment* component or any external system whose life-cycle inherently exceeds or complements the life-cycle of an SCXML session, we might as well represent these as a special I/O processor and avoid the problems with eventual cancellation. If the life-cycle needs to be controlled more closely, e.g. if the concept of a *connection session* is beneficial, we would employ a special invoker type.

   In any case, the representation of an external component depends on the nature of the component and the SCXML standard allows for a wide range of possibilities to represent them. With the missing reference of SCXML to both, the architecture and the framework, this, and most other specifics are actually undefined.

5. Adhere to the constraints imposed (statements 13-16), specifically:

   - Deliver events in the role of an MC to the invoking IM, which is possible with the `#_parent` target for `<send>` or, more generally, by addressing any other SCXML runtime via its session identifier.
   - Make sure that eventual events, in the role of an MC were sent from our IM, which is possible by checking for the `_event.origin` field or by restricting the allowed origins already in the platform.
   - Provide event handlers, respect the default event handler or ignore an event. Establishing a default event handler in SCXML is achieved by simply specifying a `<transition>` in the topmost SCXML state or at least any state higher up in the nesting order. Those `<transition>` elements in a more deeply nested state will, when enabled, override the transitions higher up in the nesting order. If no `<transition>` is enabled by an event, the event will automatically be ignored as not triggering any transitions.

     Now, in the case of an IM hierarchy, the situation is somewhat more complicated. For output events descending from IMs towards proper MCs, the `<invoke>` element features an `autoforward` attribute to automatically dispatch the event to all respective invoked systems. In the case of input events ascending from an MC towards the top-most IM, every intermediate IM will have to specify a `<transition>` as a default handler to send any unhandled event to its `#_parent` (listing 4.8).

```
<scxml datamodel="ecmascript">
  <transition event="*">
    <send eventexpr="_event.name" target="#_parent">
      <content expr="_event.data" />
    </send>
  </transition>
</scxml>
```

Listing 4.8: Forwarding an event to eventual handlers in the parent IM.

To conclude, it is perfectly possible to employ SCXML for the responsibilities of an IM and nested IM / MC pairs in the proposed W3C MDS. There are a few areas, e.g. with respect to the actual SCXML language features, where unfortunate ambiguities remain. Here, more experiences are needed in order to arrive at completely standardized multimodal dialog control and we will detail a selection of approaches in section 7.2.

4.6.2 General Applicability

While section 4.6.1 already described the role of SCXML within the W3C MDS, and we already argued for the applicability of this very MDS for interaction in pervasive environments in chapter 3, this section will briefly relate SCXML as such to the requirements specific to distributed multimodal dialog management, regardless of the W3C MDS.

Here, the emphasis on "synchronizing state via events" found in SCXML is very suited for distributed architectures. Every component can be run wherever is opportune and the overall behavior is synchronized by passing messages. In our earlier ambitions to adapt Voice eXtensible Markup Language (VoiceXML) for pervasive environments [SWRM15], we argued for four general communication schemes that guarantee the applicability of a (distributed) dialog manager in a pervasive environment (figure 4.12).

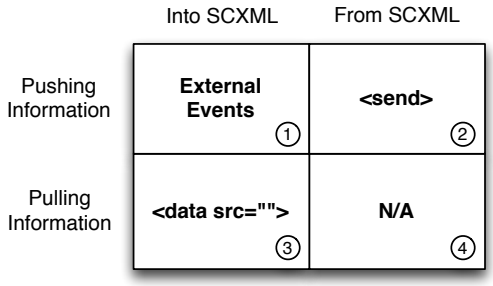


Figure 4.12.: SCXML language elements for the different communication schemes (adapted from [SWRMua13]).

1. **Pushing information from a remote system into SCXML**:
   The principal means for a remote system to push information into a running SCXML interpreter is for it to encode the information in a message and have it delivered to the external event queue. This can either be achieved by addressing a suitable I/O processor of an interpreter, such as the one of type `basichttp` or, for invoked components, via the mechanism offered by the respective invoker implementation.

   Pushing information encoded in a message via an I/O processor is more flexible as the remote system does not, necessarily, need to be invoked by the SCXML interpreter. However, it is undefined how to discover the endpoint addresses for the I/O processors of running SCXML instances. E.g. our implementation will allow to specify a TCP port and establish a HTTP server with an interpreter's session identifier as the base path for all I/O processors based on the HTTP protocol. Still, the session identifier would need to be known. Here, our implementation offers a pragmatic alternative as an aliased path based on the state-chart's `name` attribute with an integer suffix enumerating the running instances with the given name.

   Another alternative, specific to our SCXML implementation is a publish / subscribe invoker, which allows for a topic-based message delivery.

2. **Pushing information from SCXML into a remote system**:
   The obvious SCXML language feature to push information into a remote system is the `<send>` element with a respective I/O processor type. If the concept of a session is necessary, a custom invoker type can be employed. This is straight-forward to do within the language and does not leave much to be desired.

3. **Pulling information from a remote system into SCXML**:
   There are a few language features available to request information from external systems, e.g. the `src` attribute with `<data>` or the `<script>` element. But they will request the specified information as soon as the state-chart is initialized or, with a late `<data>` binding, when a containing state is entered for the first time. There is no language feature to request external information, e.g. as part of executable content.

   While it is always possible to "send a request for data" and have the remote system push information into the SCXML interpreter, this approach is somewhat awkward as it requires the remote system to understand the request. A simple assignment of a variable with the contents of e.g. an HTTP reply as can be done when initializing `<data>` is no longer available afterwards.

   To this effect, our implementation extends the SCXML specification by two additional language features:

   - A new `src` attribute for the `<assign>` element. This extension behaves much like the initialization of variables with the `<data>` element but is available as part of executable content. Assignment of the specified variable is synchronous and blocking.
   - A new element `<fetch>` as custom executable content. This element will enable the interpreter to, asynchronously, retrieve the contents of an URL specified in its `src` or `srcexpr` attribute. The contents will be made available it as the data of an external event named by the value given in the `callback` attribute and delivered as soon as the reply arrived.

   With these two extensions, it is much more comfortable and straight-forward to request additional information from remote systems at runtime.

4. **Pulling information from SCXML into a remote system**:
   Sometimes, a remote system might require information from the state-chart, e.g. the current assignment of a variable in the data-model or a response as a function of the state-chart's configuration.

   The solution given in the SCXML specification is for both endpoints, the interpreter and the external system, to maintain an open socket and to use a suitable I/O processor which will set the `_event.origin` field appropriately to be a valid target for a subsequent `<send>` element. This is problematic, however, as many remote systems will allow to request information, e.g. via HTTP, but expect the information to be contained in the actual reply and not via another connection (e.g. XHTML or VoiceXML interpreters). This is, not possible with SCXML as specified; the response of a request towards the `basichttp` I/O processor is simply a status code of `200` if an event was encoded correctly and `500` otherwise.

   Our implementation introduces a new `http` (as opposed to the specified `basichttp`) I/O processor, which will represent a complete HTTP request as an event with all its headers and content contained as compound data fields. This I/O processor is accompanied by a new executable content element `<respond>`, which accepts the event's origin as a `target` attribute and allows to actually reply to a request.

With these extensions, we achieve considerable flexibility when communicating with external systems: All four quadrants of the push/pull, from/into diagram in figure 4.12 can be addressed by convenient language features.

4.7 Deficiencies of State Chart XML and Work-Arounds

With the maturation of the standard and improved tool support for SCXML in recent years came an increased interest in applying its state-charts for a variety of applications, both from industry and academia. Multimodal infotainment systems in an automotive context, ambient assisted living and home automation, task modeling for end-user support; SCXML is seemingly being applied in ever more domains. This also entails that respective evaluations about the applicability of SCXML reveal ever more short-comings to meet the domains' various requirements. In this section, we will present some of its deficiencies and propose work-arounds, either within the standard or very close to it.

4.7.1 Dynamic Multiplicity

The apparent inability of SCXML state-charts to dynamically instantiate multiple entities of any given class surfaced many times. Imagine an SCXML state-chart handling telephone calls in a support center, each call would transition through a set of states, such as `waiting`, `in-progress`, `escalated`, `resolved`. In order to accommodate n calls, a state-chart would need each state n times contained in some parallel state, with some *active/inactive* state hierarchies within, which is clearly undesirable. Even worse, these states could not have identical names as the standard will not allow a state identifier to occur more than once. Invokers are inapplicable as well as they are bound much the

same way by a single state. Thus, a single call and its state transitions can be expressed very eloquently, but multiple dynamic instances are inexpressible.

There have been different suggestions to solve this problem. One of the first ideas was to generally expose the SCXML DOM via a W3C Core Level 2 API [Jun14] in the data-model as is done with the `document` object in dynamic HTML. However, with state-charts there is an additional problem of semantics when a system is adapted during its execution. Junger suggested to to defer all actual adaptations until a micro-step is complete but no convincing discussion about the remaining ramifications is given. Another problem with this approach is the sheer amount of work from an implementation perspective: One of the strong points of SCXML is the light-weight interface required to provide new data-models. If one were required to expose a complete DOM in the data-model, such an endeavor would become way more ambitious. Furthermore, it still leaves many other problems, e.g. unique state identifiers and the manual life-cycle management.

Forbrig [FDK15] suggested to simply introduce a new attribute `new_instance` to the `<transition>` element and have the platform handle multiplicity by repetitively entering a state via such a transition. However, this approach still leaves many open questions with regard to its semantics.

The solution we came up with in our implementation is to provide *persistent invokers*. Our `<invoke>` elements feature a new attribute `persist` which, if given, will cause the invoked component not to be terminated when the invoking state is left at the end of a macro-step. This is accompanied by an explicit `<uninvoke>` element which allows to terminate an invoker referenced by its identifier. Specifying an `idlocation` with an `<invoke>` element will already cause the platform to automatically generate an identifier, which allows repeated instantiation with ever new identifiers for subsequent communications via `<send>` and eventual termination. All other semantics remain exactly the same: the invoker can terminate on its own or be terminated via `<uninvoke>`. This allows to instantiate dynamic multiplicities by repetitively entering a corresponding state with only very little changes to the SCXML semantics specific to invocations.

4.7.2 Invoke and Send on Entry

Another inconvenience arises from the fact that `<invoke>`d components are only instantiated after a macro-step is completed. Thus, it is not possible to: 1. enter a state, 2. invoke a component and 3. directly communicate some messages via its `#_[invokeid]` as the target of a `<send>` as the respective invoker is not yet instantiated.

Listing 4.10 illustrates the problem. When the interpreter enters the given state, a nested SCXML interpreter is to be invoked (listing 4.9) and some initial message is to be send. The only event-handler that is applicable at all is `<onentry>`, but the invoker will only be instantiated once the interpreter is stable and entering a state implies that this is not yet given.

One proposed solution is to encode any initial information in the invoke request, e.g. via a `<param>` child element, but this leads to unfortunate code duplication as many of the invokers event handlers now also have to regard the initial invoke request.

```
1  <scxml>
2    <state>
3      <transition event="finish" target="done" />
4    </state>
5    <final id="done"/>
6  </scxml>
```

} Just wait for event `finish` and terminate.

Listing 4.9: SCXML state-chart in `invoke-problem.inc.scxml` invoked in listings below.

```
1    <state id="send.onentry">
2      <invoke type="scxml"
3              id="send.onentry.invoker"
4              src="invoke-problem.inc.scxml" />
5      <onentry>
6        <send target="#_send.onentry.invoker"
7              event="finish"/>
8      </onentry>
9
10     <transition event="done.invoke.send.onentry.invoker"
11                 target="send.onentry.pass" />
12     <transition event="error.communication"
13                 target="send.onentry.fail" />
14   </state>
15
```

} Component is only invoked immediately before the state-chart assumes a stable configuration.

} Sending an event to the invoked component in this `<onentry>` block will cause `error.communication` as it does not exist yet.

} Receiving the event `finish` would cause the invoked state-chart to terminate, raising `done.invoke.[invoke.ID]`.

Listing 4.10: Defective SCXML excerpt to illustrate the "send on entry" problem with invokers.

Another solution, is to enqueue some unique event to the interpreter's external queue when entering the state, which will only be consumed once the interpreter's configuration is stable. Thus, during its processing, the invoker is already instantiated (as we passed a stable configuration before dequeuing the external event) and we can send our message from the executable content processed when a respective transition is taken (listing 4.11). This is definitely a viable approach but using such an intermediate event seems somewhat intricate just to pass a message to an invoked component.

```
1     <invoke type="scxml"
2             id="external.event.invoker"
3             src="invoke-problem.inc.scxml" />
4     <onentry>
5       <send event="send.to.invoker"/>
6     </onentry>
7
8     <transition type="internal"
9                 event="send.to.invoker">
10      <send target="#_external.event.invoker"
11            event="finish"/>
12    </transition>
13    ...
14  </state>
```

When entering enqueue an otherwise unused external event with some unique name.

External events are only dequeued after a stable configuration was reached, thus the invoker is available now.

Listing 4.11: Solving the "send on entry" problem by enqueuing and reacting to an external event.

Yet another possibility is to delay the `<send>` just by a tiny amount (listing 4.12). As invocations in SCXML are synchronous the invoker will be available once the respective event is delivered. However, having to wait whatever small amount of time is still unsatisfactory as a general work-around.

```
1   <state id="delayed.event">
2     <invoke type="scxml"
3             id="delayed.event.invoker"
4             src="invoke-problem.inc.scxml" />
5     <onentry>
6       <send target="#_delayed.event.invoker"
7             event="finish"
8             delay="1ms"/>
9     </onentry>
10    ...
11  </state>
```

We specify the message to send to the invoker in `<onentry>` but only dispatch it with a delay. This will effectively send the message 1ms after the interpreter assumed a stable configuration. Invoker is guaranteed to exist as processing `<invoke>` is synchronous.

Listing 4.12: Solving the "send on entry" problem with a delayed event.

Finally, to avoid any ambiguity and provide a reliable idiom to handle these situations, our implementation of the `<invoke>` element will allow to specify an additional attribute `callback` (listing 4.13) which defines an event prefix concatenated with the invokers identifier and delivered to the interpreter's external queue as soon as the invoker is instantiated. This allows to define an internal transition which will send initial messages as soon as the invoker is available.

```
1   <state id="callback.event">
2     <invoke type="scxml"
3             id="callback.event.invoker"
4             callback="init"
5             src="invoke-problem.inc.scxml" />
6
7     <transition event="init.callback.event.invoker">
8       <send target="#_callback.event.invoker"
9             event="finish" />
10    </transition>
11    ...
12  </state>
```

Additional `callback` attribute to specify the prefix of an event `[callback].[invokeid]` to enqueue when the invoker is ready.

When the event is delivered, we can directly communicate with the invoked component.

Listing 4.13: Solving the "send on entry" problem by introducing a new `callback` attribute.

It might seem strangely specific to elaborate on this apparently obscure situation in such detail, but we ran into this issue many times when writing SCXML documents. Another possibility is to attempt to recognize this problem via a static analysis and warn an application developer by raising a respective issue via `interpreter.validate()` (see section 4.4 above), but this is hardly trivial to do. Yet another resolution could be to enqueue messages addressed to an invoker until a stable configuration is passed, but this implicitly changes the semantics of SCXML with unclear side-effects.

4.7.3 Modal States

A concept established with graphical user interfaces are “modal windows”, wherein all interactions but those specific to a modal window are suppressed until the issue presented in the modal window is resolved. As an example, a dialog to save a file might be modal and suppress all interaction with the actual application until the user either chooses a file and saves the document or cancels the dialog.

The SCXML equivalent would be a state or state hierarchy which disables or *masks* transitions within other state hierarchies. This is already possible within the same compound state hierarchy for which only a single atomic state can be active, as states deeper down are implied to be *more specific* per SCXML semantic and can overwrite transitions from their parent states. As such, it would be possible to define all transitions for events that are allowed in a modal state and provide a final, side-effect free transition with a catch-all event descriptor `*` to prevent any other transitions higher up in the state hierarchy from becoming enabled. However, if multiple atomic states are active due to a `<parallel>` state higher up in the state hierarchy, this approach is inapplicable.

One solution to this problem is to define such modal states as direct children of the topmost SCXML root state. Transitioning into those will exit all other states as only a single state of the topmost element can ever be active (the `<scxml>` root is implied to be an *or*-state). Now, in order to restore the state configuration once the issue is resolved, one would need to return via a deep `<history>` into the state-hierarchy with the rest of the control for the actual application. This still leaves the issue with invokers from the original state-hierarchy being terminated once the hierarchy is left. However, the proposed extension of persistent invokers would help to retain them and solve the issue.

4.7.4 History Stacks

Sometimes, it is desirable when navigating through a set of items (e.g. graphical screens), to remember the state of previous items to restore their state when the user transitions back. It is tempting to organize the set of items as state hierarchies beneath a common compound state and transition between them when the user chooses to navigate to another item. However, it is rather difficult and not at all obvious to model the functionality of a “back” button using this approach. It is possible to append the current item’s identifier to a stack structure in the data-model before transitioning out and regard this identifier when the back button is pressed to return via a `<history>` pseudo-state (listing 4.14).

```
1  <scxml datamodel="ecmascript">
2    <datamodel>
3      <data id="stack">[]</data>
4    </datamodel>
5
6    <parallel id="main">
7      <state id="controller">
8        <transition event="to.item1"
9                    target="item1.history">
10         <script>stack.push("item1");</script>
11       </transition>
12       <transition event="to.item2"
13                   target="item2.history">
14         <script>stack.push("item2");</script>
15       </transition>
16
17       <transition event="back"
18                   cond="stack[stack.length-1] === 'item1'"
19                   target="item1">
20         <script>stack.pop();</script>
21       </transition>
22       <transition event="back"
23                   cond="stack[stack.length-1] === 'item2'"
24                   target="item2">
25         <script>stack.pop();</script>
26       </transition>
27     </state>
28
29     <state id="items">
30       <state id="item1">
31         <history type="deep"
32                  id="item1.history" />
33         <!-- interaction would be established here -->
34       </state>
35       <state id="item2">
36         <history type="deep"
37                  id="item2.history" />
38         <!-- interaction would be established here -->
39       </state>
40     </state>
41   </parallel>
42 </scxml>
```

`ECMAScript` array to act as a stack for the items in the history.

If we are to transition to another item, push its name to the stack before transitioning. This has to be specified for each possible subsequent item.

If we are to transition *back* in the history of elements, regard the item stack in the data-model and transition into the old target state via its `<history>` pseudo-state. Again, has to be specified for each possible item.

Individual states corresponding to the various items in the history stack. This is where the screen with all interaction would be established.

Listing 4.14: History stack with a data-model structure.

However, this approach becomes very verbose and error-prone as it easily forgotten to push the current state identifier in any of the outgoing transitions. It is also not applicable to model a general stack of history items: while we store the name of a state on the `stack` in the data-model, its sub-configuration is only stored implicitly in the item's `<history>` element. This entails that a history stack, containing an item more than once will only restore its most recent history when reentered via the `back` transition. This is no problem if the items are organized in a tree and cannot be visited multiple times.

Another, arguably more elegant approach is to use nested invokers. Selecting a new item will instantiate a new state-chart and keep the complete state of the state-charts higher up in the invocation chain. To go back is to finish an invocation, which will then reestablish the original item with all its state and reenable interaction. In this approach, a top-most controller document will instantiate the first item (listing 4.15) and for every subsequent navigation to another item, a new nested state-chart is invoked (listing 4.16). In essence, we substitute the explicit stack from the data-model by the stack implicit in nested invocation of state-charts.

```
1  <scxml datamodel="ecmascript">
2    <state id="main">
3      <invoke src="history-invokers.item.scxml"
4              autoforward="true">
5        <param name="item" expr="'main'" />
6      </invoke>
7      <transition event="done.invoke"
8                  target="done" />
9    </state>
10   <final id="done" />
11 </scxml>
```

Instantiate a new nested state-chart as the topmost item `main`. Passing the `<param>` named `item` will, initialize an eventual corresponding `<data>` element in the invoked state-chart.

Do nothing but to wait for the invoker to finish.

Listing 4.15: Topmost controller for history stacks via invokers.

```
1  <scxml datamodel="ecmascript">
2    <datamodel>
3      <data id="item" />
4      <data id="childItem" />
5    </datamodel>
6
7    <state>
8      <state id="show">
9        <!-- interaction would be established here -->
10       <transition event="to.item1" target="down">
11         <assign location="childItem" expr="1" />
12       </transition>
13       <transition event="to.item2" target="down">
14         <assign location="childItem" expr="2" />
15       </transition>
16       <transition event="back" target="done" />
17     </state>
18
19     <state id="down">
20       <invoke src="history-invokers.item.scxml"
21               autoforward="true">
22         <param name="item" expr="childItem" />
23       </invoke>
24     </state>
25     <transition event="done.invoke" target="show" />
26   </state>
27
28   <final id="done" />
29 </scxml>
```

The `<param>` from our parent's invocation will cause `item` to contain e.g. the screen to display. The variable `childItem` is only declared.

In this state, the actual item with all its interaction would be established. If another item is selected, we will transition into `down` and invoke the respective state-chart.

Recursively invoke the next item and ignore all events but the completion of the invoked item. We also not specify the new item via a `<param>` element but even invoke a nested state-chart with any other document.

Listing 4.16: Individual item for history stacks via invokers (`history-invokers.item.scxml`).

There are some problems remaining with the latter approach, most notably the data-model instances will be isolated as invoked components share no state and relevant information will have to be synchronized via messages sent to the targets `#_parent` and `#_[invokeid]` respectively. Furthermore, invoking a new state-chart is more resource intensive than the stack in the data-model as a new interpreter is instantiated for each item on the stack. As such one has to be much more careful with the depth of the invocations with no means to prune the oldest items in the history.

4.7.5 Globally Unique State Identifiers

If a state in an SCXML document is assigned an identifier via its `id` attribute, it is required to be unique within the state-chart. This hampers the application of templating wherein an SCXML document is described via a set of more general concepts / patterns and generated upon request by instantiating respective templates. This problem is already evident if we allow for a simple mechanism to simply include parts of the XML document from another source, e.g. via XML inclusions or external XML entity declaration (see listing 4.17).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE scxml [
3  <!ENTITY help SYSTEM "help.xml">
4  <!ENTITY bedroom SYSTEM "bedroom.scxml">
5  <!ENTITY kitchen SYSTEM "kitchen.scxml">
6  ]>
7  <scxml xmlns="http://www.w3.org/2005/07/scxml"
8         xmlns:xi="http://www.w3.org/2001/XInclude" >
9
10   <parallel>
11     &help;
12     &bedroom;
13     &kitchen;
14     <xi:include href="bathroom.scxml" />
15   </parallel>
16
17 </scxml>
```

Declare new XML entities via the content of a XML file.

Include an XML DOM from a file.

Listing 4.17: Templating for XML document via external XML entities and XML inclusions.

It is not clear if there can be an intuitive semantic to resolve ambiguous state references introduced by duplicate identifiers in the various fragments of such a state-chart while retaining the possibility to refer to states in other fragments. One proposal has been to introduce *canonical* state references by anchoring them at some roots and refer to a state by its relative identifier, e.g. `help.introduction` but this is problematic as every sequence of characters might possibly already constitute a state identifier. It might be possible to refer to a state by an `XPath` expression and introduce an additional attribute for state references, e.g. `targetpath` to differentiate semantics. As of now, the standard requires all state identifiers to be unique and regards any other document as invalid.

Part II.
Formal Aspects and Verification

The previous chapters introduced important terms, presented State Chart eXtensible Markup Language (SCXML) both, as a XML dialect and as a dialog management technique in the context of the W3C Multimodal Dialog System (MDS) and generally established the relevance of SCXML for multimodal dialog management in pervasive environments.

This chapter will explore some formal aspects of SCXML as such, without necessarily regarding its applicability for dialog management. Its concepts of states, transitions and queues warrant a discussion of its properties with regard to theoretical computer science and invite an exploration of the applicability of formal methods such as equivalent transformations and model checking. In the context of this thesis' main argument, this chapter establishes important prerequisites which are essential for the claim that "these dialog models can be made accessible to the formalisms of temporal logic" in the next chapter. The overall approach described in this and the next chapter is depicted in figure 5.1.

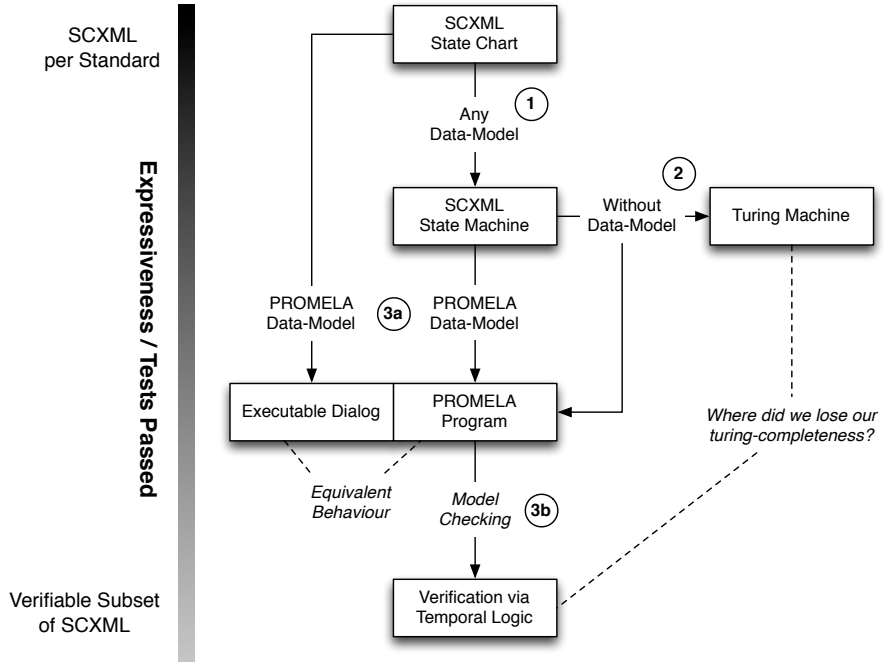


Figure 5.1.: Approach to make SCXML dialog models accessible for formal verification with temporal logic.

1. We start by describing a semantic-preserving transformation of arbitrary SCXML state-charts onto SCXML state-machines (section 5.3), wherein only a single state can ever be active. This transformation is agnostic of the employed scripting language but requires a few slight modifications of a compliant interpreter to encompass all of SCXML's language features. In essence, this step removes much of SCXML's semantics specific to configuration management, transition preemption and event name matching required by an interpreter by *pre-calculating* them at transformation time. The resulting SCXML state-machines, however, are by no means ordinary Deterministic Finite Automaton (DFA)s as many of the remaining SCXML semantics are not found in this puristic state-machine conception.

2. This leads to the important consideration of determining the computational model of SCXML as its rank on the Chomsky hierarchy (section 5.4). Here, we will begin by showing its computational model to be at least capable to recognize context-free grammars by embedding a Push-Down Automaton (PDA) in SCXML via nested invokers (we actually already did so in section 4.7.4). Subsequently, we will conclude SCXML's model of computation as being even Turing complete (section 5.4.2) by embedding a Deterministic Queue Automaton (DQA) via the *broadcast communication* mechanism available with the `<raise>` element and SCXML's semantic variance of *instantaneous states* (compare section 3.2.2.3 and [vdB94]). Both, the PDA and the DQA can be embedded in SCXML without requiring any embedded scripting language as a data-model, implying that the expressiveness of the data-model is inconsequential with regard to the computational model of SCXML.

Both implications have important ramifications for our application of model checking as we require a finite enumerable global state space and will have to identify some limitations to reduce the computational model to DFA equivalence.

3. With these formal properties shown, the next chapter will describe our approach to verify claims given in a temporal logic regarding a state-chart's behavior via a model-checker. To this effect we

   a) present the PRocess MEta LAnguage (PROMELA) data-model as an embedded scripting language for SCXML documents that is (i) suitable to be transformed for formal verification and (ii) available to actually execute verifiable dialogs (section 6.3) and

   b) describe a largely automated transformation to create semantically equivalent input files for the SPIN model checker for a subset of SCXML state charts employing the NULL or PROMELA data-model (section 6.4).

In totality, the overall approach allows us to employ the formalisms of model-checking for a subset of SCXML documents by transforming them onto a state-machine and, subsequently, onto a PROMELA program.

The description of this main approach from figure 5.1 is accompanied by some complimentary considerations for (i) the upper bound of states in the state-machine representation in section 5.3.8 with a Monte-Carlo experiment to compare the compactness of the state-chart and state-machine representation and (ii) the description of a dynamic approach to minimize the number of states in the state-machine representation.

Whenever applicable, all the approaches in this and the next chapter are evaluated via the 232 official tests (detailed in section 5.1) accompanying the SCXML specification as part of its "Implementation and Report Plan". In particular, the tests are used to

- Show semantic equivalence of the transformation from SCXML state-charts to state-machines with all tests passed in section 5.3.7. This implies the assumption that every language feature is tested for.
- Evaluate the tightness of the proposed upper bound for the number of states in the state-machine representation in section 5.3.8.
- Give an impression about the potential reduction for the number of states with the dynamic minimization in the state-machine representation in section 5.5.
- Evaluate the expressiveness of the PROMELA data-model as a scripting language in SCXML as the number of tests passed in the next chapter, section 6.3.4.
- Evaluate the overall approach of formal verification for SCXML documents with the PROMELA data-model, again in the next chapter in section 6.4.13.

The overall approach for model-checking of SCXML documents in this chapter has already been described and published [RNSW14], though with much less details and without the evaluation via the SCXML tests.

5.1 State Chart XML Tests from the W3C

The SCXML Implementation Report Plan (IRP)[1] defines 233 tests of all functional requirements for a compliant interpreter (table 5.1). Virtually all of the non-trivial language features of SCXML are only available in the presence of an embedded scripting language as a data-model (see table 4.3). Therefore, all tests are given in a custom XML dialect that needs to be XSLT transformed for a specific data-model to keep the actual functional requirement independent of the employed data-model. With normative specifications for the ECMAScript, XPath[2] and NULL data-model given in the SCXML specification, respective XSLT transformation files are supplied as part of its IRP tests. Of these 233 tests, 51 do not test a functional requirement of SCXML, but a requirement specific to one of the three normative data-models and are inapplicable to other data-models, leaving a subset of 182 actually data-model agnostic tests.

Each test defines two final states `pass` and `fail` and encodes a given functional requirement in such a way that a compliant interpreter will terminate via `pass` and all others via `fail`. This is not possible for a subset of seven manual tests (bracketed in table 5.1), where the output of the interpreter or any other aspect of its external behavior will have to be manually inspected by a tester.

[1] `http://www.w3.org/Voice/2013/scxml-irp/`

[2] The normative specification of the XPath data-model was dropped after the first "Last Call Working Draft" due to an insufficient number of implementation reports.

In addition to the XSLT transformation for ECMAScript and XPath, our implementation provides transformations for Prolog, Lua and PROMELA, with the latter playing an important role when evaluating the PROMELA data-model in section 6.3.4. Whenever possible all of the transformations described in the following sections are subjected to the tests to verify the transformation and get an idea about the expressiveness retained and the functionality lost.

Class	Section	#Tests
Core Constructs		**40 (1)**
General	3.2	2
State	3.3	1
Final	3.7	2
OnEntry	3.8	2
OnExit	3.9	2
History	3.10	4
Events	3.12	4
Transition Selection	3.13	23 (1)
Executable Content		**13**
Raise	4.2	1
If	4.3	3
Foreach	4.6	7
Evaluation	4.9	2

Class	Section	#Tests
Data Model and Manipulation		**50 (4)**
Data	5.3	7
Assign	5.4	4
Donedata	5.5	1
Content	5.6	3
Param	5.7	3
Script	5.8	4 (1)
Expressions	5.9	8 (3)
System Variables	5.10	20
External Communications		**51 (3)**
Send	6.2	19 (1)
Cancel	6.3	3
Invoke	6.4	29 (2)
Data Models		**51**
NULL	B.1	1
ECMAScript	B.2	20
XPath	B.3	30
Event I/O Processors		**28 (1)**
SCXML	C.1	16
Basic HTTP	C.2	12 (1)
Total		**233 (9)**

Table 5.1.: Number of tests in the SCXML Implementation and Report Plan with corresponding section from specification. Brackets indicate manual tests.

Table 5.2 lists the subset of tests applicable for the various data-models implemented in our SCXML interpreter. The number of tests is constituted by the 182 data-model agnostic tests, plus the tests specific to the given data-model. For additional data-models with no normative description in the SCXML specification, only the agnostic tests are applicable, with the exception of the Lua data-model, were the ECMAScript specific tests were added as well.

Data-Model	Passed / Tests	Remarks
ECMAScript	`202 / 202`	All tests pass for both Google's V8 and Apple's JSC implementation.
XPath	`107 / 212`	Transformation is from SCXML IRP but our implementation is incomplete. Data-model was eventually dropped completely from SCXML specification due to missing implementation reports.
Lua	`158 / 202`	Due to syntactic and semantic similarities, set of ECMAScript tests was transformed (set obtained from Nvidia).
PROMELA	`163 / 182`	Discussed in detail in section 6.3.4.
Prolog	`60 / 182`	XSLT file for transformation not completed.

Table 5.2.: Tests passed by the data-models implemented in our SCXML interpreter.

5.2 Nomenclature

Before we continue with the discussion of SCXML's formal aspects, this section will introduce important sets and relations specific to SCXML as an XML dialect. It is intended as a reference when reading through the subsequent, formal descriptions. For the electronic version of this dissertation, the respective terms are even linked from the various formulae.

With regard to the names of the various sets, the basic rule is that sets and entities from the original state-chart under consideration have no decoration, sets or entities from the state-machine representation we transform onto are decorated with a tilde and important intermediate sets are decorated with a hat. The step i of an entity is its first occurrence in a breadth-first traversal of possible configurations in the original state-machine.

S Set of proper states (`<state>`, `<parallel>`, `<final>`) from the original state-chart.

$\mathcal{P}(S)$	Power set (set of all subsets) of proper states from the original state-chart.
S_a	Sets of states $\subseteq \mathcal{P}(S)$ that can form a legal active configuration in a state-chart.
$s_a(i)$	Set of states $\in S_a$ active in the state-chart at step i.
$s_d(i)$	Set of states $\subseteq S$ with nested data that were already visited prior to step i on the path from the initial active configuration $s_a(0)$ to the configuration at step i.
$s_h(i,h)$	Set of states $\subseteq S$ to be remembered for history element h at step i.
$s_h(i)$	Assignments of remembered states $\subseteq S$ for all history elements at step i.
$\widetilde{s}(i)$	The global state at step i as the union of the current active configuration $s_a(i)$, all states with nested data elements already visited $s_d(i)$ and all history assignments $s_h(i)$.
$\prec$	The *precedes* relation for active configurations of the state-chart, where $s_a(i) \prec s_a(j)$ if $i < j$ in the breadth-first traversal of the state-chart's state space of possible configurations.
$\vdash$	The *ancestor of* relation for XML elements in an SCXML document, where $e_1 \vdash_1 e_2$ if e_1 is the father element of e_2 and $e_1 \vdash_m e_2$ if e_1 and e_2 are separated by exactly $m-1$ ancestors. If no subscript is given, defaults to $\vdash_m$ with m arbitrary.
$\widetilde{\mathcal{S}}$	All global states $\widetilde{s}(i)$ a state-chart will assume.
$T(i)$	All transition elements that are direct descendants of states in the active configuration at step i.
$\mathcal{P}(T(i))$	Power set of all transition elements that are direct descendants of states in the active configuration at step i.
$\widehat{\mathcal{T}}(i)$	Unsorted set of non-conflicting transition sets $\subseteq \mathcal{P}(T(i))$ that maybe taken together at step i.
$p(t)$	The priority of a transition from the original state-chart for transition selection. Such that, if a set of transitions $\subset \mathcal{P}(T(i))$ contains conflicting transitions, removing those with a lower priority will cause the set to become optimally enabled.
$p(\widetilde{t})$	Priority of global transition as the sum of its constituting transitions priorities as powers of two.
$\widetilde{\mathcal{T}}(i)$	The sorted set of transition sets $\subseteq \mathcal{P}(T(i))$ ordered by their priority that maybe taken together at step i.
$\widetilde{t}$	A global transition as the union of original transitions that maybe taken together.
$\widetilde{t}_{i,j}$	The global transition in step i at position j from $\subseteq \widetilde{\mathcal{T}}(i)$.
$\mathcal{X}(\widetilde{t})$	Executable content to be processed when taking global transition $\widetilde{t}$. It is the union of executable content in `<onexit>` of states left, those specified with a transition and those in the `<onentry>` handlers of states entered.
$\mathcal{E}(\widetilde{t})$	Longest event name that enables all constituting transitions in the global transition $\widetilde{t}$.
$\mathcal{C}(\widetilde{t})$	The condition that guards the global transition $\widetilde{t}$ as the logical conjunction of its constituting transitions from the original state-chart.

5.3 Transformation to State-Machines

This section introduces an approach to syntactically transform any SCXML document representing a *state-chart* into a semantically equivalent SCXML document representing a *state-machine*. The transformation is novel in such that it transforms the complete SCXML language, regardless of the employed scripting language, into equivalent state-machines. Being able to represent every state-chart as a state-machine does, by no means, imply that every SCXML document is expressible as a DFA as there are still , e.g., the internal and external event queues as infinite FIFOs available.

Formally, every state-machine is a special state-chart, wherein only a single state can only ever be active. This implies that we can use SCXML itself to notate the resulting state-machines but can not have any composite states. Using SCXML to notate the resulting state-machine allows us to validate the transformation via the SCXML tests accompanying the W3C specification with a compliant interpreter on the transformed documents. As semantic equivalence is claimed, all tests that passed for the original state-chart representation need to pass again in their state-machine representation, regardless of the employed data-model.

For a first intuition, an approach such as the power-set construction, as it is used to transform a Non-Deterministic Finite Automaton (NFA) to a DFA [Sch97] should be applicable to transform state-charts to state-machines. We would start with a pristine state-chart in the empty configuration ($\widetilde{s}(0)$) and either perform a micro-step for an eventual, top-most `<initial>` transition or assume the completion of the state-chart's root. In this first and every subsequent *global state* ($\widetilde{s}(i)$) we identify the combinations of active transitions that can potentially form an optimal transition set ($\widetilde{T}(i)$) and perform a micro-step for each, *resetting* the global state of the interpreter in between each set. This will eventually establish a finite graph structure when all micro-steps for optimal transition sets lead back to a global state already visited (see figure 5.2). This construction is described more formally below.

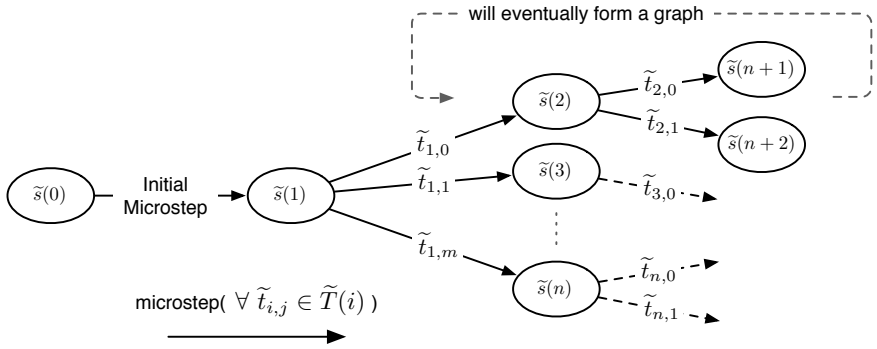


Figure 5.2.: Principal idea to flatten the state-chart into a state-machine. We will formally define the various sets and entities in the following sections.

There are several SCXML language features that complicate this approach. We cannot simply regard only an active configuration of the state-chart to constitute a new global state in the state-machine as some other SCXML language features influence the subsequent active configuration and, as such, the set of executable content processed, or the set of data-model operations within a micro-step for the same optimal transition set. Furthermore, some language features explicitly rely on the hierarchical structure of the state-chart:

- **The history of composite states:**
  `<history>` elements cause the interpreter to behave differently when used as the target of a transition. Initially entering a composite state via one of its history pseudo-states causes the interpreter to assume a configuration as if no history was given, but subsequent entries will cause the interpreter to enter nested states as they were active when the composite state was left. Depending on the type of the history, this behavior maybe limited to the direct descendent level of states or all descendant levels recursively. Furthermore, a `<history>` element may contain an unconditional, spontaneous transition to a valid nested state which is to be taken upon entering the history pseudo-state initially. This transition may even contain executable content.

- **Nested data elements**:
  `<data>` elements nested in a state and with the document's data binding attribute set to `late`, are only initialized when the respective state is entered initially and ignored on subsequent entries.

- **Data raised upon exiting a composite state**:
  `<donedata>` elements are available to associate data to `done.state.ID` events raised by the platform when a complex or parallel state is left.

- **Invoked entities**:
  `<invoke>` elements instantiate external entities when state hierarchies are entered and left. These cannot be represented in a state-machine where only a single state is ever active. Here, small adaptations to the SCXML interpreter are needed.

- **The `'in'` predicate**:
  Every data-model in SCXML is mandated to support the `in` predicate, which takes the name of a state and returns whether the given state is in the active configuration. As the original state names will get intermingled in the new state-machine's state names, adaptations to the interpreter are needed here as well.

Additionally, a huge part of an interpreter's overall state might be contained in the data-model. For the transformation from a state-chart to a state-machine, we can ignore this part of the state, though, as it will be exactly the same as long as the resulting state-machine will execute the same set of data-model instructions in the same order when it is interpreted. There is one exemption resulting from the construction detailed below: boolean expressions guarding transitions will not, necessarily, be called in the same order or be evaluated the same number of times. We have to assume that these *guard conditions* are free of side-effects. While not formally mandated by the World Wide Web Consortium (W3C) standard, it can be considered best practice with state-charts in general.

In the following sections, we will more formally introduce *global states* as the new states for the transformed state-machine and *global transitions* to connect these.

5.3.1 Global States

Ignoring the state of the embedded data-model as argued above, we can define what constitutes a *global state* for the resulting state-machine. If we were to ignore history and data elements as well, such a global state would simply be the union of the original state-chart's active states at a given time, identical to the states in the powerset construction when transforming NFAs to DFAs. But history assignments and nested data elements will also influence the subsequent active configuration, the set of executable content to be processed and the data-model operations within a micro-step. As such, we need to encode the assignments of history elements and whether data elements were already processed in a global state: (i) the set of states active when a state containing a history pseudo-state was left is needed to reenter these when the history is the target of a transition somewhen in the future and to ignore eventual executable content associated with a history's initial transition and (ii) the set of states already entered at least once before is needed to preserve the late-initialization semantics with nested data elements.

We will define S to be the set of states from the original state-chart and $S_a \subseteq \mathcal{P}(S)$ the set of legal configuration as sets of states that can be active at the same time. For every state-chart expressed in SCXML, there is a single initial set of states $s_a(1) \in S_a$ as the first configuration an interpreter will assume after the initial micro-step. Starting with this initial configuration, we can enumerate all subsequent configurations $s_a(i) \in S_a$ in a breadth-first traversal by recursing via all valid sets of transitions for each new configuration (compare figure 5.2). As argued above, we also need to take history assignments ($s_h(i)$) and late data binding ($s_d(i)$) into account when considering whether two such global states are equal, which leads us to the following formal definition of a global state $\widetilde{s}(i)$ for the state-machine. Let $s \in S$ be a state from the original state chart, i the index from the breadth-first traversal, $s_a(i) \prec s_a(j)$ the *precedes* relation on the path from the initial state $s_a(1)$ to $s_a(j)$ and $s_i \vdash s_j$ the *ancestor-of* relation from XML.

$$
\begin{aligned}
s_a(i) &:= \{s \mid s \text{ active at step } i\} \in S_a && (5.1)\\
s_d(i) &:= \{s \mid s \in s_a(j) \wedge s_a(j) \prec s_a(i) \wedge s \vdash_2 \texttt{<data>}\} && (5.2)\\
s_h(i,h) &:= \{s \mid s_h \vdash s \wedge s \in s_a(j) \wedge s \notin s_a(i) \wedge s_a(j) \prec s_a(i)\} && (5.3)\\
s_h(i) &:= \{s_h(i,1), .., s_h(i,H)\} && (5.4)\\
\widetilde{s}(i) &:= (s_a(i),\ s_d(i),\ s_h(i)) && (5.5)\\
\widetilde{\mathcal{S}} &:= \bigcup_{0..I} \widetilde{s}(i) && (5.6)
\end{aligned}
$$

A global state $\widetilde{s}(i)$ is the sorted set containing (i) the states $s_a(i) \in S_a$ that are active at step i, (ii) the set of states $s_d(i) \in \mathcal{P}(S)$, containing a `<data>` element that were already visited before and (iii) the child states to be reentered for every history pseudo-state $s_h(i)$ we left somewhen earlier . All `<data>` elements are wrapped in a `<datamodel>` element, thus the $s \vdash_2$ `<data>` relation in formula 5.2. The formalization of the histories as $s_h(i,h)$ in 5.3 is somewhat ambiguous as it does not distinguish shallow from deep histories, but a stricter formalization would be unnecessarily complicated. The important thing is, that we need to remember which states to reenter when a given history is the target of a transition. In the transformed state-machines, we will serialize $\widetilde{s}(i)$ as:

```
active:{s0,s1};visited:{s0,s1,..};history:{histId1:{s0,s1,..},histId2:{s0,s1,..},..}
```

This representation is used as the global states' identifiers and allows to identify the original state-chart's configuration, history assignments and late data binding. When a constituting set is empty, it is not written in the serialized representation. The set of global states for a state-chart is obviously finite and can be enumerated. We will see in the construction later that only those states, that are actually reachable will be part of the resulting state-machine.

5.3.2 Global Transitions

With state-charts, there are usually multiple states in an active configuration. Each state might define transitions and the set of transitions for a configuration might even conflict as they would lead to an invalid configuration. Whenever the interpreter is in a stable configuration and an event arrives, the *optimal set of transitions* (see definitions 7 - 12) has to be identified and processed in a micro-step.

In a state-machine, the first enabled transition in a global state per document-order is, per definition, optimally enabled as there can be no descendant states. Furthermore, as there will always be only a single atomic state active, this single transition even forms the complete optimal set of transitions from definition 12. This is important to realize: the first enabled transition in the state-machine has to be semantically equivalent to the complete optimal transition set from the state-chart, in such that it will enter the correct new global state and execute the exact same operations.

To this effect, we will have to identify all potential optimal transition sets, merge them into *global transitions*, sort them and write them in document-order accordingly to retain the semantics from the state-chart.

To identify all potential optimal transition sets, we start by identifying all transitions from states active in the configuration at step i as the potentially enabled transitions. We will, without loss of generality, assume that every transition is only enabled by a single event descriptor, as we can easily bring any transition into this form by duplicating it for every event descriptor. Let $s \vdash_1 t$ be the *parent-of* relation for elements in the XML representation, then we can define the set of potentially enabled transitions per step as:

$$T(i) \quad := \quad \{t \mid s \vdash_1 t \wedge s \in S_a(i),\ t \text{ a } \texttt{<transition>}\} \tag{5.7}$$

Now, $T(i)$ is the set of all potentially enabled transitions in step i. Its power-set $\mathcal{P}(T(i))$ contains all their combinations and is, therefore, a superset of all the configuration's optimal transition sets. Most of these transition sets can never form an optimal transition set and we need to discard the subset of invalid transition sets from $\mathcal{P}(T(i))$, leaving only potential optimal transition sets. By interpreting the chain of definitions (7-13) from the introduction of the SCXML semantics as necessary conditions, we can formulate criteria to remove transition sets from $\mathcal{P}(T(i))$.

Let $\mathcal{E}(t)$ be the set of event names which enable transition t, e the name of an event:

Criterion 1: (From definition 7 for an active transition) *t in state s is active if s is in the state-chart's active configuration*: Here, t is a transition and s an atomic state, active in the current configuration. All active atomic states and their ancestors *are* the active configuration $s_a(i)$ of an interpreter, therefore, $T(i)$ as defined above will contain all active transitions for an interpreter in step i.

Criterion 2: (From definition 8.2 for a matched transition) *t matches the name of e*: This is a necessary condition for a transition to be matched. For a set of transitions in $\mathcal{P}(T(i))$ to form a potential optimal transition set, there needs to be an event that matches all constituting transitions. Therefore, we can remove those transition sets that have no common event name.

$$Inv_1(i) := \{T \in \mathcal{P}(T(i)) \mid \nexists e : \forall t \in T, e \in \mathcal{E}(t)\} \tag{5.8}$$

There is another criterion to invalidate transitions sets in $\mathcal{P}(T(i))$ implied. As eventless and eventful transitions are processed in different steps of the interpretation algorithms, they cannot be mixed in an optimal transition set.

$$Inv_2(i) := \{T \in \mathcal{P}(T(i)) \mid \exists t_i, t_j \in T : \mathcal{E}(t_i) = \epsilon \wedge \mathcal{E}(t_j) \neq \epsilon\} \tag{5.9}$$

Criterion 3: (From definition 10.2 for an enabled transition) *t lacks a 'cond' attribute or its 'cond' attribute evaluates to "true"*: We cannot know the boolean value of the evaluation of a guard condition at transformation time and as such, will have to account for all possible valuations. We will see later that we can resolve this issue by providing all subsets of potential optimal transition sets and merge the conditions of their constituting transitions via a logical conjunction. For now, we cannot use this criterion to invalidate any transition sets in $\mathcal{P}(T(i))$.

Criterion 4: (From definition 11.2) *no transition that precedes t in document order in t's source state is enabled by e in s*: This is another criterion to remove transition sets from $\mathcal{P}(T(i))$. For a set of transitions $T \in \mathcal{P}(T(i))$ to form an optimal transition set, all its constituting transitions need to be optimally enabled, this cannot be the case if two transitions from the same source state are contained.

$$Inv_3(i) := \{T \in \mathcal{P}(T(i)) \mid \exists t_i, t_j \in T, s \in S : s \vdash_1 t_i \wedge s \vdash_1 t_j)\} \tag{5.10}$$

Criterion 5: (From definition 11.3) *no transition is enabled by e in s in any descendant of t's source state*: This necessary attribute of an optimally enabled transition allows us to remove those sets $T \in \mathcal{P}(T(i))$, that contain two transitions with nested source states as at least one of those cannot not be optimally enabled.

$$Inv_4(i) := \{T \in \mathcal{P}(T(i)) \mid \exists t_i, t_j \in T, s_j, s_i \in S : (s_i \vdash_1 t_i) \vdash (s_j \vdash_1 t_j)\} \tag{5.11}$$

Criterion 6: (From definition 12.2) *no transition conflicts with another transition in the set*: If two optimally enabled transitions conflict as having overlapping exit sets, they cannot both be part of the optimal transition set.

$$Inv_5(i) := \{T \in \mathcal{P}(T(i)) \mid \exists t_i, t_j \in T : t_i \text{ conflicts } t_j\} \tag{5.12}$$

We will invalidate transition sets containing both, but keep sets that contain either. As above, we will have to make sure that the set containing the transition with the higher priority preceded the other in document order.

By removing these invalid sets from $\mathcal{P}(T(i))$, we arrive at a preliminary, unsorted set of all potential optimal transition sets of the original state-chart at step i and their subsets:

$$\widehat{\mathcal{T}}(i) := \mathcal{P}(T(i)) \setminus \{Inv_1(i) \cup Inv_2(i) \cup Inv_3(i) \cup Inv_4(i) \cup Inv_5(i)\} \tag{5.13}$$

There is an optional optimization that can be performed to reduce $\widehat{\mathcal{T}}(i)$ some more: Transition sets in $\widehat{\mathcal{T}}(i)$ can be removed if there is a superset in $\widehat{\mathcal{T}}(i)$ that just adds transitions without a `cond` attribute:

$$Opt_1 := \{T_1 \in \widehat{\mathcal{T}}(i) \mid \exists T_2 \in \widehat{\mathcal{T}}(i), T_2 \supset T_1, \forall t \in \{T_2 \setminus T_1\} : t \text{ is unconditional}\} \tag{5.14}$$

This has no semantic consequences as supersets will need to precede subsets anyway and, without an additional term for the conjugated `cond` attribute of the superset, it is always enabled when the subset is. It foremost improves readability of the resulting state-machines by removing superfluous transitions.

We can now define a *global transition* $\widetilde{t}$ as the union of the individual transitions $t \in T \in \widehat{\mathcal{T}}(i)$ with the following properties: let $\mathcal{E}(t)$ be the set of events enabling transition t and $\mathcal{C}(t)$ the data-model specific condition that guards t:

$$\mathcal{E}(\widetilde{t}) = \bigcap \mathcal{E}(t), t \in T, T \in \widehat{\mathcal{T}}(i) \tag{5.15}$$

$$\mathcal{C}(\widetilde{t}) = \bigcup \mathcal{C}(t), t \in T, T \in \widehat{\mathcal{T}}(i) \tag{5.16}$$

The global transition's enabling event descriptor is the intersection of the event descriptors that enable the individual transitions. We know that such an event name exists per equation 5.8 and we do know that it is a single event descriptor as we postulated (w.l.o.g) that every transition has a single event descriptor. It is the shortest event descriptor from all constituting transitions in $T \in \widehat{\mathcal{T}}(i)$ as this event descriptor is enabled by all longer event names. The global transition's guard condition is the logical conjunction of all boolean expressions that guard the individual transitions. Note that this is, again, a purely syntactic transformation as we can just concatenate the individual conditions with a logical `and` from the data-model's language.

In the transformed state-machine, we only have the selection criterion for transitions with regard to document order and have to sort the global transitions in $\widehat{\mathcal{T}}(i)$ such that the first enabled transition represents all transitions in the optimal transition set of the state-chart for a given event received in step i. With the criterion for optimal transition sets derived from the SCXML definitions above, we can identify the properties such a sorting will have to provide:

Ordering 1: (From criterion 3) *Accounting for all assignments of truth in a potential optimal transition set*: At transformation time, we cannot know the value of a transition's guard condition as it will most likely rely on the state of the data-model. Still, we need to select the largest set of enabled transitions as the optimal set. As none of the criteria above removed the subsets of potentially optimal transition sets from $\mathcal{P}(T(i))$, all subsets for every $T \in \widehat{\mathcal{T}}(i)$ are still contained as well. With a global transition's guard condition being the logical conjunction of the constituting transitions, we only have to make sure that supersets precede their subsets.

Ordering 2: (From criterion 4) *For each transition's source state, no proceeding transition is enabled*: This criterion necessitates that all sets of transitions $T \in \widehat{\mathcal{T}}(i)$ be sorted such that for all pairs of transition sets, that differ only in transitions from a given source state, are sorted by these transitions' document order.

Ordering 3: (From criterion 5) *No transition in a nested state is enabled*: This criterion necessitates a similar ordering to the one above, but for pairs of transitions that differ only in transitions with nested source states, whereas the more deeply nested state has to come first.

Ordering 4: (From criterion 5) *For conflicting transition's, the set containing the transition with the higher priority comes first*: This ordering is implied by the two criteria above.

In essence we have to make sure that the global transitions containing the highest priority transitions from the original state-charts are first in document-order per state. That is, if *a transition t is enabled by named event e in atomic state s* (compare definition 10) no transition with a lower priority can be enabled via the same state and therefore, precede the respective global transition in document-order per state.

There is a rather elegant approach to sort the transition sets in $\widehat{\mathcal{T}}(i)$ to conform to this ordering. We will first formally introduce a transition's *priority* in the original state-chart as an integer value $p(t) \in [0, |T| - 1]$, with higher values signifying precedence of a transition when determining the optimal transition set. Let T be the set of all transitions in the original state-chart, $t \in T$ and $s \in S$ any state:

$$p(t) := \mid T \mid \tag{5.17}$$
$$- |\{t' \in T \mid \exists s : (s \vdash_1 t) \wedge (s \vdash_1 t') \wedge t' \text{ precedes } t \text{ in document order}\}| \tag{5.18}$$
$$- |\{t' \in T \mid \nexists s, m : s \vdash_m t \wedge s \vdash_1 t' \wedge t' \text{ precedes } t \text{ in document order}\}| \tag{5.19}$$
$$- |\{t' \in T \mid \exists s, m : s \vdash_1 t \wedge s \vdash_m t'\}| \tag{5.20}$$

We start by assuming that the transition t has the highest priority given as the number of total transition elements $\mid T \mid$ in the state-chart (equation 5.17) and subsequently subtract the number of transitions with an even higher priority. There are three classes of transitions with a higher priority than a given transition t: (i) every sibling transition that precedes t in document order in the same state will have a higher priority (equation 5.18), (ii) every transition in a state that is no ancestor of t's source state and precedes t in document order will have a higher priority (equation 5.19) and (iii) every transition that is contained in one of the descendant states of t's source. Coincidentally, the priority of a transition is simply the sequence of visiting the transitions in a reverse post-order traversal of the state-chart's tree.

This corresponds to the semantics of *priority* used throughout the definitions in the SCXML recommendation (i.e. definition 13). Now, we can interpret the priority of a transition $t \in T$ as the order of magnitude in a positional notation with base-2 and define the priority of a set of transitions $p(\widetilde{t}), \widetilde{t} \in \widehat{\mathcal{T}}(i)$ via its constituting transitions from T as:

$$p(\widetilde{t}) := \sum_{j=1}^{|p(\widetilde{t})|} 2^{p(t_j)}, t_j \in \widetilde{t} \tag{5.21}$$

This definition of a transition set's priority can also be conceived as a binary encoded number, where the bit at position k is enabled if $k = p(t), t \in p(\widetilde{t})$ and disabled otherwise. This ordering has a number of desirable properties:

1. **No two transition sets have the same priority**: Just as with binary encoded numbers, a given priority can only be attained by a specific sequence of transition indices. If two transition sets have the same priority, the sets themselves are equal.

2. **Supersets precede subsets**: Removing any transition from a set will reduce the set's priority as all of the summands in equation 5.21 are positive and at least one will be missing.

3. **More specific event descriptors precede less specific**: This follows directly from the property above. If a transition set has a more specific common event-descriptor, it will contain more transitions as those enabled by the more specific descriptor are also contained.

4. **Every transition set containing (t_j) precedes those without it, except for (t_k) with $p(t_k) > p(t_j)$**: Transition sets containing the highest indexed transition precede all those not containing it. Within those classes, sets containing the next highest precede those without. This ensures that the first enabled set contains only optimal enabled transitions and that the set is maximal.

This allows us to, finally, define the sorted set $\widetilde{\mathcal{T}}(i)$ of potentially optimal transition sets from $\widehat{\mathcal{T}}(i) \subseteq \mathcal{P}(T(i))$ as:

$$\begin{aligned} \widetilde{\mathcal{T}}(i) := (T_1, .., T_N \mid T_j \in \widehat{\mathcal{T}}(i), \\ \forall k, l (1 \leq k < l \leq N) : \\ p(T_k) > p(T_l)) \end{aligned} \tag{5.22}$$

We can now merge the constituting transitions for all $T \in \widetilde{\mathcal{T}}(i)$ according to equation 5.15 and 5.16 and have them as sorted, transformed global transitions $\widetilde{t}_{i,j}$ in the global state at step i.

There is a final optional optimization to reduce the set of global transitions from the sorted set $\widetilde{\mathcal{T}}(i)$. If a global transition $\widetilde{t}_{i,l}$ is preceded in $\widetilde{\mathcal{T}}(i)$ by an unconditional global transition $\widetilde{t}_{i,k}$, reacting to the same event, $\widetilde{t}_{i,k}$ will always be enabled whenever $\widetilde{t}_{i,l}$ is, rendering the $\widetilde{t}_{i,l}$ superfluous:

$$Opt_2 := \{\widetilde{t}_{i,l} \in \widetilde{\mathcal{T}}(i) \mid \exists \widetilde{t}_{i,k} \in \widetilde{\mathcal{T}}(i) : \mathcal{E}(\widetilde{t}_{i,k}) = \mathcal{E}(\widetilde{t}_{i,l}) \wedge \mathcal{C}(\widetilde{t}_{i,k}) = \emptyset \wedge k < l\} \tag{5.23}$$

5.3.3 Executable Content & Transient State Chains

For an SCXML state-machine to be semantically equivalent to an SCXML state-chart, it will not only have to transition through respective global states via global transitions, but foremost, to interpret the same *executable content* when exiting or entering states and taking transitions. While it might already be relevant for an external system to determine whether a given state is active or not, it is this executable content that generates most of the state-charts externally observable behavior.

We cannot simply aggregate and copy the various states' executable content from their `onexit` and `onentry` handlers into the transformed global states where these are active as some states may not have been entered / left via one transition originating in a given state, but entered / left via another transition. Instead, we will associate all executable content with the global transitions. Let $\widetilde{t} \in \widetilde{\mathcal{T}}(i)$ be a global transition originating in global state $\widetilde{s}(i)$ and $t \in \widetilde{t}$ a single transition from the potential optimal transition set represented by $\widetilde{t}$. We define its executable content $\mathcal{X}(\widetilde{t})$ as:

$$\begin{aligned}
\mathcal{X}_{exit}(\widetilde{t}) &:= \{x \mid s \vdash_1 x, s \in s_a(i) \wedge s \text{ is exited by } \widetilde{t}, x \text{ an } \texttt{<onexit>} \text{ element}\} && (5.24)\\
\mathcal{X}_{trans}(\widetilde{t}) &:= \{x \mid t \vdash_1 x, t \in \widetilde{t} \text{ a } \texttt{<transition>} \text{ element}\} && (5.25)\\
\mathcal{X}_{entry}(\widetilde{t}) &:= \{x \mid s \vdash_1 x, s \text{ is entered by } \widetilde{t}, x \text{ an } \texttt{<onentry>} \text{ element}\} && (5.26)\\
\mathcal{X}(\widetilde{t}) &:= (\mathcal{X}_{exit}(\widetilde{t}) \cup \mathcal{X}_{trans}(\widetilde{t}) \cup \mathcal{X}_{entry}(\widetilde{t}))\} && (5.27)
\end{aligned}$$

The set $\mathcal{X}(\widetilde{t})$ contains all executable content that is interpreted when the original state-chart transitions from the configuration represented by $\widetilde{s}(i)$ via the optimal transition set in $\widetilde{t}$. A naive approach would now just move all this executable content into the respective global transition, but there are some caveats:

- Error semantics in SCXML specify that *if the processing of an element causes an error to be raised, the processor must not process the remaining elements of the block.* Here, a block is any sequence of executable content contained in an `<onentry>`, `<onexit>` or `<transition>` element. As such, these blocks have to preserved for the interpreter to continue with the next block if an error was raised.
- Nested `<data>` elements with a document's data binding set to `late` will cause initialization of data-model specific variables whenever such a state is entered. Executable content may behave differently when such data is (un)initialized. Therefore, we will have to preserve `<data>` elements and make sure they are interpreted at the correct time.
- The mandated `In('id')` predicate depends on the states potentially already entered and exited.

The solution is to decompose global transitions in the presence of executable content into *transient state chains*. Here, a global transition will not target the next global state directly, but enter a series of transient states, connected via spontaneous transitions and ending in this next global state (see figure 5.3).

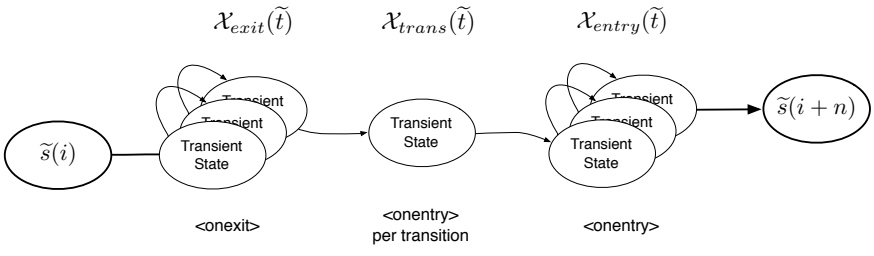


Figure 5.3.: Transient state chain to retain semantics of executable content in a global transition $\widetilde{t}$.

- Each of the global transition's `onexit` elements in $\mathcal{X}_{exit}(\widetilde{t})$ will be copied verbatim into a transient state with a global state-identifier reflecting the active configuration at the time of exiting the corresponding state in the original state-chart (see the required extensions for an interpreter in section 5.3.5). If an original state defined multiple `<onexit>` handlers, they can share a transient state, but the individual blocks must be preserved.

- Executable content from transitions in $\mathcal{X}_{trans}(\widetilde{t})$ is copied into the next transient state as `<onentry>` elements. It is just as valid to use `<onexit>` elements, the important thing is to use blocks to maintain the error semantics.

- The `<onentry>` elements in $\mathcal{X}_{entry}(\widetilde{t})$ have to be copied into the next sequence of states, interleaved by transient states with nested `<data>` elements and again with state identifiers reflecting the original states already entered as with $\mathcal{X}_{entry}(\widetilde{t})$.

5.3.4 Construction

Now we have all the formalisms in place to describe the actual transformation depicted in figure 5.2 from a state-chart expressed in SCXML to an equivalent state-machine in SCXML. We have defined above:

- The global state of an interpreter $\widetilde{s}(i) \in \widetilde{\mathcal{S}}$ at step i containing (1) its active configuration $s_a(i) \subseteq \mathcal{S}$, (2) the states with nested data elements $s_d(i) \subseteq \mathcal{S}$ that we already visited if the data binding is late and (3) the set of states to be reentered $\widetilde{s}_h(i) \subseteq \mathcal{S}$ when a history pseudo-state is the target of a transition.

- A sorted set of global transitions $\widetilde{t}_{i,j} \in \widetilde{\mathcal{T}}(i)$ per global state as potential optimal transition sets connecting the global states. With the first enabled transition representing the optimal transition set from the original state-chart for a given event and condition.

- Transient state chains as an approach to maintain the semantics of executable content, with regard to (i) error handling, (ii) the sequence of its execution (iii) the interleaving of initializing nested data and (iv) the states active when the respective content is executed.

On a practical note, we employed an actual SCXML interpreter to help with the transformation. By inheriting the interpreter's base-class, we can intercept the invocation of executable content and the initialization of nested `<data>` elements while relying on the implementation for the general SCXML semantics, e.g. transitioning between configurations and the correct assignment of history pseudo-states.

We start by defining $\widetilde{s}(0)$, the initial global state for the new state-machine, containing the empty active configuration, no states with nested data elements already visited and empty state assignments for all history pseudo-states:

$$\widetilde{s}(0) \quad = \quad (\emptyset, \emptyset, \emptyset) \tag{5.28}$$

Now we start the interpreter and let it assume the state-chart's initial active configuration. This will, potentially, cause the interpretation of executable content as transitions are taken and states are entered. We save this executable content in $\mathcal{X}(\widetilde{t})$ for the spontaneous global transition leading from $\widetilde{s}(0)$ to the initial configuration $\widetilde{s}(1)$. We push $\widetilde{s}(1)$ and the transition onto a queue $\mathcal{Q}$ and continue by performing the steps given as pseudo-code in algorithm 2 until the queue is exhausted.

We pop the global configuration $\widetilde{s}$ and the global transition $\widetilde{t}$ that lead to $\widetilde{s}$ from the queue (line 2) and set $\widetilde{t}$ target to $\widetilde{s}$ (line 3). If we have already seen this state, we can continue with the next item on the queue $\mathcal{Q}$ or terminate if $\mathcal{Q}$ is empty (line 4-6). If not, we increase the step i and assign it as an attribute to $\widetilde{s}$, essentially establishing $\widetilde{s}(i)$ (line 7-9). If the active configuration s_a encoded in $\widetilde{s}$ contains a top-level final state, the transformed state-machine would stop and we do not need to follow this branch any further (line 10-12). Now, if $\widetilde{s}$ was yet unseen and did not contain a top-level final state, we will identify its potential, sorted optimal transition sets $\widetilde{\mathcal{T}}(i)$ (line 13) and perform a micro-step for each, resetting the interpreters state in between (line 14-17). For any global state $\widetilde{s}(k)$ we reach via a global transition, we enqueue both, the state $\widetilde{s}(k)$ and the respective transition $\widetilde{t}_{i,j}$ on the queue $\mathcal{Q}$ (line 18) and continue to pop the next pair from $\mathcal{Q}$. Not shown is the creation of $\mathcal{X}(\widetilde{t})$, which is implicit in a micro-step as we overrode the respective functions in the interpreter $\mathcal{I}$.

When we established all global states and their transitions, we finally copy any global `<data>` and `<script>` elements into the state machine's `<scxml>` element and write an XML file.

Input : Queue $\mathcal{Q}$<configuration, transition>, Interpreter $\mathcal{I}$
Output: Global states $\widetilde{\mathcal{S}}$, global transitions $\widetilde{T}$

while $\mathcal{Q}$ *not empty* **do**
  $(\widetilde{s},\ \widetilde{t}) \leftarrow \texttt{pop}(\mathcal{Q})$
  $\widetilde{t}_{.target} \leftarrow \widetilde{s}$
  if $\widetilde{s} \in \widetilde{\mathcal{S}}$ **then**
    continue
  end
  $i \leftarrow i + 1$
  $\widetilde{s}_{.step} \leftarrow i$
  $\widetilde{\mathcal{S}} \leftarrow \widetilde{\mathcal{S}} \cup \widetilde{s}$
  if *top-level final state* $\in s_a$ **then**
    continue
  end
  for $\widetilde{t}_{i,j} \in \widetilde{\mathcal{T}}(i)$ **do**
    $\mathcal{I}_{.active} \leftarrow s_a$
    $\mathcal{I}_{.history} \leftarrow s_h$
    $\mathcal{I}_{.visited} \leftarrow s_d$
    $\widetilde{s}(k) = \mathcal{I}.\text{microstep}(\widetilde{t}_{i,j})$
    $\texttt{push}(\widetilde{s}(k), \widetilde{t}_{i,j})$
  end
end

Algorithm 2: Breadth-first transitioning of the state space to establish $\widetilde{\mathcal{S}}$ and $\widetilde{T}$.

5.3.5 Required Extensions to the Interpreter

We argued in the introduction of this section, that there are five language features in SCXML that are problematic when mapping state-charts to state-machines: (i) history elements, (ii) data elements in nested states, (iii) donedata elements, (iv) the invoking of components and (v) the `In` predicate for the data-models. Our approach of incorporating history element assignments and states with nested data elements that were already visited and initialized into global states solves the first two issues. Supporting the `<invoke>` element, `<donedata>` and the `In` predicate requires some adaptations of an interpreter though.

The <donedata> Element

Whenever final states within a composite state are entered, a compliant interpreter is required to raise respective `done.state.ID` events on the interpreter's internal queue. Here, the `<donedata>` element can be used to specify additional data to be attached to these events' `data` fields. The element will accept `<content>` and `<param>` child elements much like the `<send>` element does. However, with a state-machine there can be no nested or parallel state hierarchies as only a single state can only ever be active. Consequentially, the interpreter will have to raise these events on the internal queue explicitly.

A first idea might be to substitute every occurrence of a `done.state.ID` being raised by an explicit `<send>` element with an `#_internal` target attribute and just insert the `<donedata>`'s child elements. However, the `<send>` element has slightly different error semantics than the automated raising of done events: When the evaluation of a an expression in either the `param.expr` or `content.expr` attribute fails, the `<send>` element will enqueue an `error.execution` event and disregard everything else, wherein the `done.state.ID` events raised automatically by the platform will just drop any data attached via `<donedata>` in the presence of evaluation errors.

A viable solution is to extend the `<raise>` attribute. Per standard, no child elements are allowed within a `<raise>` element and only the `event` attribute to specify the name of the event to be pushed onto the internal queue is available. As such, it has no error semantics related to the evaluation of expressions. We will allow `<content>` and `<param>` child elements and define the error semantics as with the `<donedata>` element for `done.state.ID` events raised by the platform: In the presence of evaluation errors, the attached data is dropped but the event is delivered on the internal queue.

The `<invoke>` Element

The SCXML specification mandates that *the invoke element is executed after the state's onentry element and causes an instance of the external service to be created.* Thus, entering a stable configuration with an `<invoke>` element as a child of an active state causes the invocation of an external service. Similarly, with regard to its termination, it is mandated that *if the invoking state machine exits the state containing the invocation before it receives the done.invoke.id event, it cancels the invoked session.* In other words, the invocation is cancelled when the interpreter's active configuration does no longer contain the invocation's source state at the end of a macro-step. This causes a problem in a state-machine representation, because any invoked component would be cancelled whenever a state transition occurs as there can be no composite states and the active configuration will only ever contain a single state.

To retain the invocation / cancel semantics of invoked systems, we will not cancel invocations as soon as their source state is left, but introduce an explicit `<uninvoke>`. Now, when we transition through the state machine, any invoked component will be left instantiated until an explicit `<uninvoke>` with the invocation's `id` is the found as the child of a global state. This is consistent with the proposed extension of the `<invoke>` element for modal states and multiplicity in section 4.7.

The `In` Predicate

With regard to the `In` predicate, SCXML specifies that *all data models must support the 'In()' predicate, which takes a state ID as its argument and returns true if the state machine is in that state.* With state-machines, the interpreter is only ever in a single state, nevertheless, the identifiers of global states encode all active states (e.g. `active:c0,a01`) and the active states from the state-chart representation can be retrieved by parsing the string.

5.3.6 Transformation Examples

This section will illustrate the transformation detailed above with a few examples. For all examples, the states in the transformed state-machine representation are sorted by their occurrence in a breadth-first traversal of reachable states in the state-chart. This corresponds to the *step* used throughout the description of the transformation above and is annotated with a `step` attribute per global state. I.e. the global state $\widetilde{s}(i)$ is given by the state-machine's state with a value of i for the `step` attribute. The global state's actual `id` attribute encodes all members of $\widetilde{s}(i)$, allowing to map state-machine states to state-chart configurations (i.e. for the `In` predicate).

Individual transitions in the state-charts will be referenced as t_p, with p being their priority $p(t)$ annotated as the XML `priority` attribute. For the resulting set of transitions in the state-machine, the `member` attribute lists the constituting transitions from the state-machine by their priority. Executable content, associated to a global transition $\mathcal{X}(\widetilde{t})$ will simply be referenced by its line numbers $l_{start-end}$ from the state-machine. A complete example with all language features of SCXML is found in the appendix A.1.

None of the additional attributes `priority`, `step` or `members` shown in the following examples have any semantic during interpretation. They merely provide a means to refer to elements in the text and help to exemplify the transformation. The exception is the `flat` attribute in the topmost `<scxml>` element, which is regarded to switch the behavior of the `In` predicate for multiple state names encoded in a single global state's name (see section 5.3.5 above).

5.3.6.1 Simple Example

In this example, we will describe the transformation of the state-chart in listing 5.1 onto the equivalent state-machine in listing 5.2. The original state-chart is, in fact, already a state-machine as no nesting of states is used. Nevertheless, it is useful to exemplify the basic approach of the construction and to introduce *global state tables* to visualize the transformation.

```
1  <scxml name="transform1">
2    <state id="start">
3      <transition priority="1" event="foo" target="quit"/>
4      <transition priority="0" event="bar" target="quit"/>
5    </state>
6    <final id="quit">
7      <onentry>
8        <log expr="'done'"/>
9      </onentry>
10   </final>
11 </scxml>
```

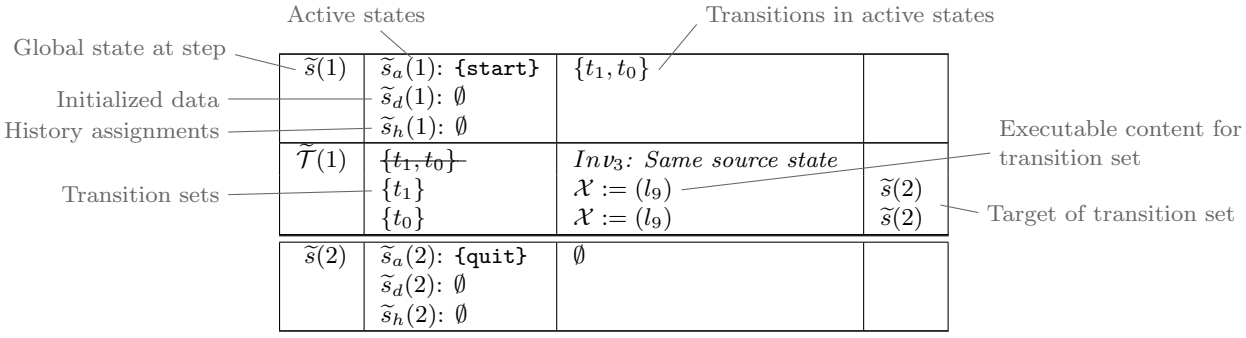
`start` is initial active state (per document order) and transitions to final state `quit` on event `foo` or `bar`.

Upon entering `quit` a message is logged via executable content.

Listing 5.1: Simple example (original state-chart).

The state-chart in listing 5.1 will simply enter the state `start` as its initial configuration and wait for the event `foo` or `bar` before transitioning per t_0 or t_1 to the final state `quit`, where a message is printed and interpretation ends. For the construction of the respective state-machine (listing 5.2), we start by introducing the initial global state $\widetilde{s}(0)$, with no states from the state-chart active as `active:{}`. From here, we have a spontaneous transition to $\widetilde{s}(1)$ representing the state-chart's initial configuration with state `start` active.

All global states representing actual configurations of the state-chart are detailed in table 5.4. The first row per global state contains its name in the first column and, the constituting sets in the second column as (i) the set of states $s_a(i)$ that are active in the current step, (ii) the states with nested data elements $s_d(i)$ that were already visited and initialized and (iii) the assignments of history states in $\widetilde{s}_h(i)$. The third column lists the transitions that are children of the active configuration, referenced by their priority. The second row lists the individual sets in the power-set of all transitions $\mathcal{P}(T(i))$ in the second column. They are sorted accordingly to form $\widetilde{\mathcal{T}}(i)$. If a subset from $\mathcal{P}(T(i))$ is invalid, the respective reason is given in the third column, otherwise its executable content $\mathcal{X}(\widetilde{t})$ is referenced by line numbers. The last column specifies the transition sets' global target state.



$\widetilde{s}(1)$	$\widetilde{s}_a(1)$: `{start}` $\widetilde{s}_d(1)$: $\emptyset$ $\widetilde{s}_h(1)$: $\emptyset$	$\{t_1, t_0\}$	
$\widetilde{\mathcal{T}}(1)$	~~$\{t_1, t_0\}$~~ $\{t_1\}$ $\{t_0\}$	*Inv₃: Same source state* $\mathcal{X} := (l_9)$ $\mathcal{X} := (l_9)$	 $\widetilde{s}(2)$ $\widetilde{s}(2)$
$\widetilde{s}(2)$	$\widetilde{s}_a(2)$: `{quit}` $\widetilde{s}_d(2)$: $\emptyset$ $\widetilde{s}_h(2)$: $\emptyset$	$\emptyset$	

Table 5.4.: Global state table with transitions for the simple example.

With the details given in table 5.4, we introduce $\widetilde{s}(1)$ as a new state (`step="1"` in listing 5.2) with an `id` attribute encoding the three sets of active and visited states as well as the history assignments. For this global state, we create two global transitions containing t_0 and t_1 respectively. Both transitions have executable content attached, thus their targets are transient state chains with one state each and spontaneous transitions to their actual targets which is $\widetilde{s}(2)$ in both cases.

```
1  <scxml flat="true" name="transform1" initial="active:{}">
2    <state step="0" id="active:{}">
3      <transition members="   " target="active:{start}"/>
4    </state>
5    <state step="1" id="active:{start}">
6      <transition members="1  " event="foo" target="active:{quit}-via-0"/>
7      <transition members="  0" event="bar" target="active:{quit}-via-1"/>
8    </state>
9    <state transient="true" id="active:{quit}-via-0">
10     <onentry>
11       <log expr="'done'"/>
12     </onentry>
13     <transition target="active:{quit}"/>
14   </state>
15   <state transient="true" id="active:{quit}-via-1">
16     <onentry>
17       <log expr="'done'"/>
18     </onentry>
19     <transition target="active:{quit}"/>
20   </state>
21   <state step="2" id="active:{quit}" final="true"/>
22 </scxml>
```

Start state $\widetilde{s}(0)$, spontaneously transitions to $\widetilde{s}(1)$ with state-chart's first configuration.

Global state $\widetilde{s}(1)$ with initial configuration of state-chart and global transitions.

Transient state from $\widetilde{s}(1)$'s first transition to $\widetilde{s}(2)$ with executable content.

Transient state from $\widetilde{s}(1)$'s second transition to $\widetilde{s}(2)$ with executable content.

Listing 5.2: Simple example (transformed state-machine).

5.3.6.2 Overriding Transitions

A more interesting example is the case where multiple states can be active at the same time, as states more deeply nested can override transitions specified higher up. This can be understood to be a form of specialization, as the state-chart can have a default reaction to an event in one of the upper states that is overridden in a more deeply nested state.

The state-chart in listing 5.3 employs a compound state `c0`, where the second child state will override a transition from the compound parent state. This transition is only in effect if `a02` is in the active configuration and takes precedence as its deeper nesting level implies a higher priority than the transition for the same event in `c0`.

```
1  <scxml name="transform2">
2    <state id="c0">                                          | Transitions to quit when respective event is received.
3      <transition priority="0" event="toQuit" target="quit"/>
4      <state id="a01">
5        <transition priority="2" event="toA02" target="a02"/>
6      </state>
7      <state id="a02">                                       | Overrides transitions above to quit2 when a02 is active.
8        <transition priority="1" event="toQuit" target="quit2"/>
9      </state>
10   </state>
11   <final id="quit"/>
12   <final id="quit2"/>
13 </scxml>
```

Listing 5.3: Overriding transitions (original state-chart).

Following the construction, we can establish the global state table in 5.5. We can see that the configuration with `c0` and `a02` is found as global state $\widetilde{s}(2)$, with both transitions in $\widetilde{\mathcal{T}}(2)$. The subset containing both transitions is invalidated: nested transitions can never occur in an optimal transition set as they are enabled by the same atomic state (here `a02`). The transition sets containing either t_1 or t_0 both have the form of an optimal enabled transition set but the latter will never be selected as the former is unconditional and always selected for the event `toQuit`.

$\widetilde{s}(0)$	$\widetilde{s}_a(0)$: $\emptyset$ $\widetilde{s}_d(0)$: $\emptyset$ $\widetilde{s}_h(0)$: $\emptyset$	$\{t_{initial}\}$	
$\widetilde{\mathcal{T}}(0)$	$\{t_{initial}\}$	$\emptyset$	$\widetilde{s}(1)$
$\widetilde{s}(1)$	$\widetilde{s}_a(1)$: {c0, a01} $\widetilde{s}_d(1)$: $\emptyset$ $\widetilde{s}_h(1)$: $\emptyset$	$\{t_2, t_0\}$	
$\widetilde{\mathcal{T}}(1)$	~~$\{t_2, t_0\}$~~ $\{t_2\}$ $\{t_0\}$	*Inv*$_4$: *Nested transitions* $\emptyset$ $\emptyset$	 $\widetilde{s}(2)$ $\widetilde{s}(3)$
$\widetilde{s}(2)$	$\widetilde{s}_a(2)$: {c0, a02} $\widetilde{s}_d(2)$: $\emptyset$ $\widetilde{s}_h(2)$: $\emptyset$	$\{t_1, t_0\}$	
$\widetilde{\mathcal{T}}(2)$	~~$\{t_1, t_0\}$~~ $\{t_1\}$ ~~$\{t_0\}$~~	*Inv*$_4$: *Nested transitions* $\emptyset$ *Opt*$_1$: *Earlier unconditional match*	 $\widetilde{s}(4)$
$\widetilde{s}(3)$	$\widetilde{s}_a(3)$: {quit} $\widetilde{s}_d(3)$: $\emptyset$ $\widetilde{s}_h(3)$: $\emptyset$	$\emptyset$	
$\widetilde{s}(4)$	$\widetilde{s}_a(4)$: {quit2} $\widetilde{s}_d(4)$: $\emptyset$ $\widetilde{s}_h(4)$: $\emptyset$	$\emptyset$	

Table 5.5.: Global state table for example with overriding transitions.

With this global state table, we can generate the transformed state-machine in listing 5.4. Note that the absence of executable content causes the transformation not to generate any transient state-chains.

```
1  <scxml flat="true" name="transform2" initial="active:{}">
2    <state step="0" id="active:{}">
3      <transition members="     " target="active:{c0,a01}"/>
4    </state>
5    <state step="1" id="active:{c0,a01}">
6      <transition members="2    " event="toA02" target="active:{c0,a02}"/>
7      <transition members="    0" event="toQuit" target="active:{quit}"/>
8    </state>
9    <state step="2" id="active:{c0,a02}">
10     <transition members="  1  " event="toQuit" target="active:{quit2}"/>
11   </state>
12   <state step="3" id="active:{quit}" final="true"/>
13   <state step="4" id="active:{quit2}" final="true"/>
14 </scxml>
```

Outer transition to `quit` is in effect.

Transition to `quit` is overridden to state `quit2`.

Listing 5.4: Overriding transitions (transformed state-machine).

5.3.7 Evaluation of the State-Machine Equivalence

SCXML documents, subjected to the transformation detailed above and interpreted with an extended interpreter (see section 5.3.5) retain all their original semantics. In particular, all tests from table 5.2, regardless of the employed data-model, still pass when transformed onto state-machines before interpretation (see figure 5.4). Even if this is no formal proof, it gives a good approximation as every functional requirement from the SCXML specification is tested.

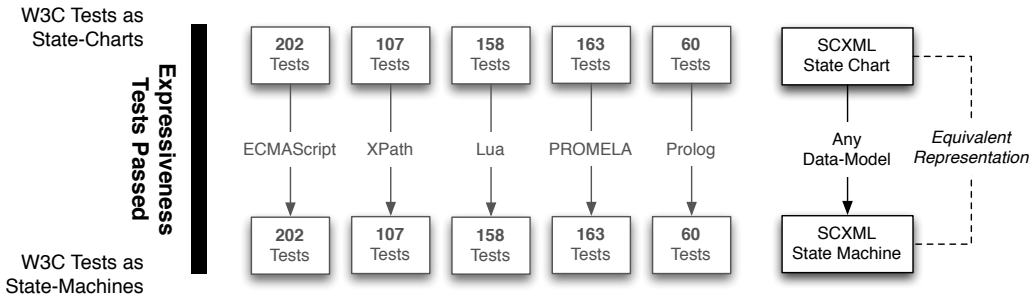


Figure 5.4.: All of the original SCXML tests pass in flattened form as well.

Therefore, with all tests that our interpreter passed for the SCXML state-charts from the W3C IRP also passing as state-machines, we will claim semantic equivalence of our SCXML state-machine representation. Furthermore, with all functional requirements for a compliant interpreter tested as part of the IRP, we will also claim the domain of the transformation onto SCXML state-machines as being all of SCXML, i.e.:

Lemma 1 (State-Machine Equivalence): *Every SCXML state-chart can be transformed onto a semantically equivalent SCXML state-machine with* $\forall t : |s_a(t)| = 1$ *for a slightly extended interpreter.*

5.3.8 Upper Bound for Number of Global States

The construction above allows us to give an upper bound for the number of states (*global states*) in the resulting state machine and some best-practices to reduce this state space. We can recursively calculate the maximum number of states $|\widetilde{\mathcal{S}}|_u$ as follows, let $s_0 \in S$ be the root of the state-chart:

$$|\widetilde{\mathcal{S}}|_u = b(s_0) \times 2^d \times \prod_0^{|H|} h_i \tag{5.29}$$

$$b(s) = \begin{cases} 0 & \text{if } s \text{ is unreachable} \\ 1 & \text{if } s \text{ is atomic} \\ \sum\ b(s_i), s \vdash_1 s_i & \text{if } s \text{ is compound} \\ \prod\ b(s_i), s \vdash_1 s_i & \text{if } s \text{ is parallel} \end{cases} \tag{5.30}$$

$$d = \begin{cases} |\{s \in S \mid s \vdash_1 \text{data}\}| & \text{if data binding is late} \\ 0 & \text{else} \end{cases} \tag{5.31}$$

$$H = \{b(s), s \in S \mid s \vdash_1 \text{deep history}\} \cup \{|s_i| \mid s \vdash_1 s_i \wedge s \vdash_1 \text{shallow history}\} \tag{5.32}$$

The first term $b(s)$ calculates the number of states as they were if the original state chart contained no nested data elements or history, wherein (i) atomic states lead to one state in the state-machine's space, (ii) parallel states multiply the state space of their child states as all children can assume configurations individually and concurrently and (iii) compound states add the state space of each of their children as each child can be in any valid configuration, but only one child is active at a given time. We can improve the tightness by determining the reachability of a state and disregarding the contribution of unreachable states to the machine representation's state space. Though, a state-chart with unreachable states could also be considered mildly defective to begin with. For the reachability we can define a superset $\mathcal{R}^+$ of reachable states as follows: The root `<scxml>` state is always $\in \mathcal{R}^+$, every other state $s \in \mathcal{R}^+$ if (i) s is the target of a transition t from a reachable state $\widetilde{s} \vdash_1 t$ and $\widetilde{s} \in \mathcal{R}^+$ or (ii) s is the initial state in a reachable, nested state $\widetilde{s} \in \mathcal{R}^+$ either per document order or `<initial>` element or (iii) s is in a reachable, parallel state $\widetilde{p} \vdash_1 s$ and $\widetilde{p} \in \mathcal{R}^+$. The first property, with s being the target of a transition from a reachable state would need to ignore the transition's condition and event as we cannot know, in the general case, when or even if the transition can ever be in an optimally enabled transition set.

Due to the SCXML semantics of initializing nested data (with a late data-binding) only when the containing state is entered initially, each such state doubles basic state space $b(s_0)$. These states are simply counted in d and multiplied as 2^d with $b(s_0)$ to get the basic state space with all possible { `visited` / `not yet visited` } combinations for states with nested data elements.

Finally, every possible assignment for history pseudo-states has to be resolved in the state-machine's state space. In the case of shallow histories, where only the direct descendant states of a containing state are remembered, this multiplies the overall state space by the number of children. In the case of deep histories, the complete set of possible states $b(s)$ for the containing state has to be taken into account.

With this equation we can see the biggest contributors in terms of state explosion:

- Deep history pseudo-states are the largest contributors to the state explosion. Every occurrence multiplies the state space by the number of legal configurations of its parent state's child states.
- Each state with nested data elements cause the complete state space to double.
- Shallow histories still multiply the complete state space by the number of child states of their parent.

Figure 5.5 depicts a histogram of the upper bound for the state-machine representation of all W3C tests applicable for the ECMAScript data model along with a distribution of the upper bound's tightness as the difference to the actual number of states.

First of all, we can see that the proposed upper bound actually is an upper bound for the given set of state-charts as there are no instances where the number of actual global states exceeds the upper bound. Furthermore, it is noteworthy that these test's state-machine representations have only a few states, which is to be expected as they tend to be minimal and only test a singe functional requirement. For the vast majority of this tests, the upper bound is equal to the actual number of states. When the upper bound differs, there is a subset of global states in {`legal configurations` × `history assignments` × `visited states`} that is never assumed during interpretation. The transformation to state-machines will only visit reachable states and as such, there is a difference between the calculated upper bound and the actual number of states. The difference is most pronounced if there are deep history states or nested data elements with a late data binding, as argued above. The single outlier is `test387` with an upper bound of 769, but only 8 actual global states in its state-machine representation. It tests a series of deep history elements but will only have a very short, completely pre-determined execution sequence.

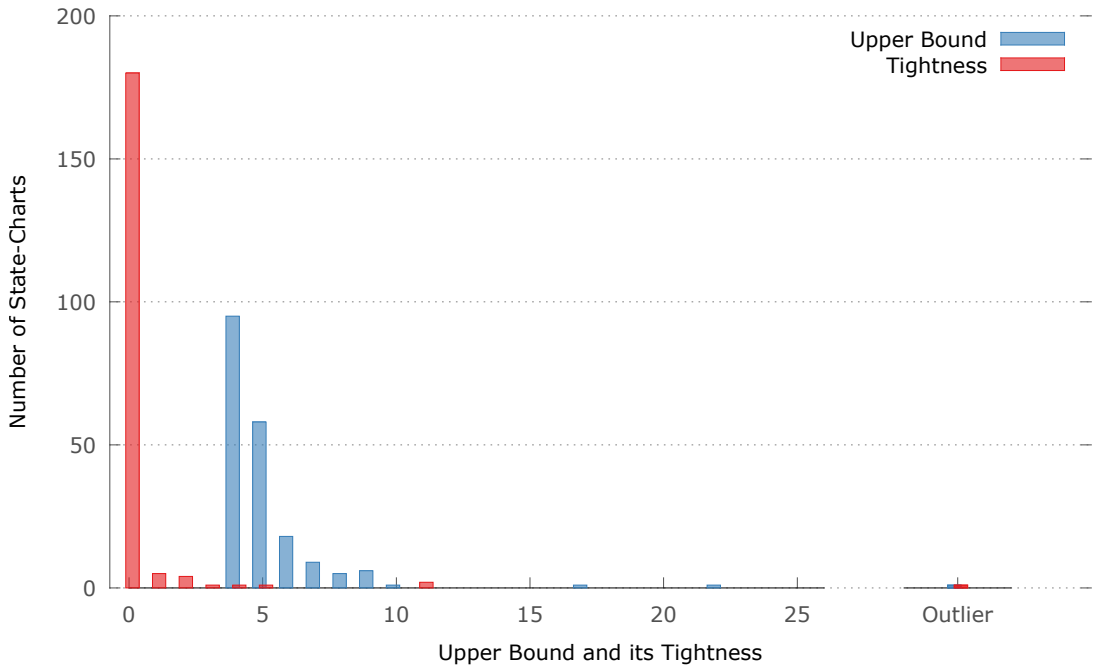


Figure 5.5.: Histogram of estimated upper bound for number of states and its *tightness* as the difference to the actual number of states for the 191 automated and 5 manual W3C tests applicable to the ECMAScript data-model.

With this upper bound, we can also give an idea about the compactness of the SCXML state-chart representation when compared to state-machines. If we interpret an instance of an SCXML document as the result of a random process and determine the upper bound for each, we arrive at the graph given in figure 5.6. The green line represents the maximum upper bound observed for state-charts with the given number of states from a set of 500.000 random state-charts. Here, we consider `<state>`, `<parallel>`, `<final>` and `<history>` as a state. The pseudo-state `<initial>` is not created by the random process (see below), nor are nested `<data>` elements with a late data binding. Each red cross represents a single SCXML document for a random subset of 1.000 instances from the total data set of 500.000 documents to give an impression about the distribution of the upper bound per number of states in a state-chart.

The random process chosen to create an SCXML document works as follows:

1. Create the topmost `<scxml>` element as the root state and recursively create child states in the next step.
2. Create $[min..6]$ new child states, with min depending on the type of the parent state:

$$min = \begin{cases} 2 & \text{if parent is } \texttt{<parallel>} \\ 0 & \text{else} \end{cases}$$

   The probability distribution for the different types of child states was chosen as follows. It is a subjective estimation of the individual types' frequency:

Type	Probability Mass
`<state>`	0.5
`<history>` (deep)	0.2 x 0.4
`<history>` (shallow)	0.2 x 0.6
`<parallel>`	0.2
`<final>`	0.1

   If the child state's type is `<parallel>` or `<state>` and there are not already 8 child states in total goto step 2, otherwise stop this recursion branch.

3. Terminate and write SCXML document.

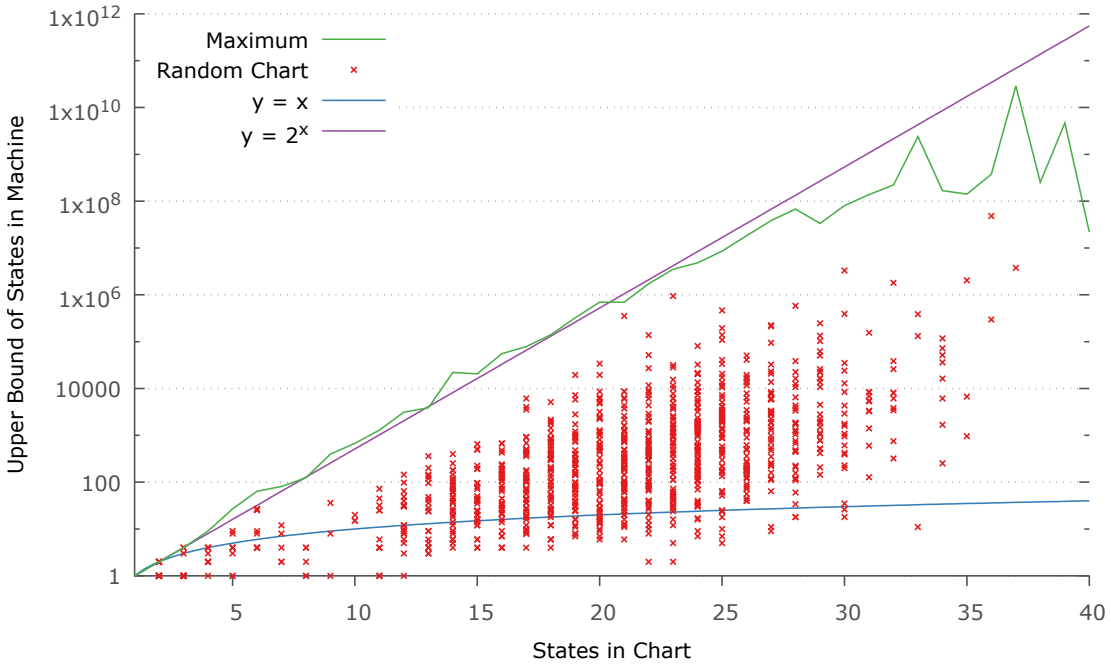


Figure 5.6.: Monte-Carlo experiment with the upper bound for the number of global states in 500.000 random SCXML documents.

The reachability of a state in the state-chart representation is disregarded to calculate the upper bound, as we could always introduce a respective `<transition>` element without contributing to the state-chart's state space. Nested `<data>` elements with late data-binding are disregarded as well, as the maximum would simply increase by, at most, 2^s with each state containing such an element.

The resulting graph in figure 5.6 is to be interpreted somewhat cautiously and only relevant if we believe the upper bound $|\widetilde{\mathcal{S}}|_u$ to be a good estimate of the relation in the number of states between a state-chart and state-machine representation. Nevertheless, it does provide a useful intuition about the compactness of an SCXML state-chart.

5.4 Computational Model of State-Chart XML

To get an idea of the expressive power and limits of SCXML, a good start is to identify its rank in the Chomsky hierarchy (table 5.6). To classify a given a computational formalism in this hierarchy, one has to show that it will accept the same set of *languages* as one of the reference models in the hierarchy. This is usually done by embedding a recognizing automaton in the computational formalism under consideration and vice versa.

Type	Additional Restriction	Recognized by
Type 0: Recursively enumerable	$w_1 \rightarrow w_2$ (no restriction)	Turing Machine, Deterministic Queue Automaton
Type 1: Context-sensitive	$w_1 \rightarrow w_2 : \lvert w_1\rvert \leq \lvert w_2\rvert$	Linear-Bounded Non-Deterministic Turing Machine
Type 2: Context-free	$w_1 \rightarrow w_2 : w_1 \in V$	Non-deterministic Push-Down Automaton
Type 3: Regular	$w_1 \rightarrow w_2 : w_2 \in T \cup TV$	Deterministic Finite Automaton, Regular Expressions

Table 5.6.: Languages in the Chomsky hierarchy and their recognizing automatons.

Languages, in this context, refer to words producible by a formal grammar via its permitted production rules. The various types in the Chomsky hierarchy impose increasing restrictions with regard to the form these production rules may employ, consequentially limiting the scope of languages that can be expressed. The most restrictive languages

can be recognized by simple DFAs, while the most expressive grammars describe languages only recognizable by a Turing machine or one of its equivalents.

We will see in section 6.2 that our approach to verification with temporal logic formulae is only defined for the most restricted computational model and, as such, requires equivalence to a DFA. The easiest approach to make sure that a formalism is equivalent to a DFA is to guarantee finite enumerability. In the following sections we will abuse two unbound concepts in SCXML to embed a PDA (Type-2) and even a Turing machine (Type-0) and thereby identify SCXML language features which cannot be formally verified (figure 5.7). In both case, it is possible to limit the respective unbound concept to an arbitrary upper bound to restore finite enumerability.

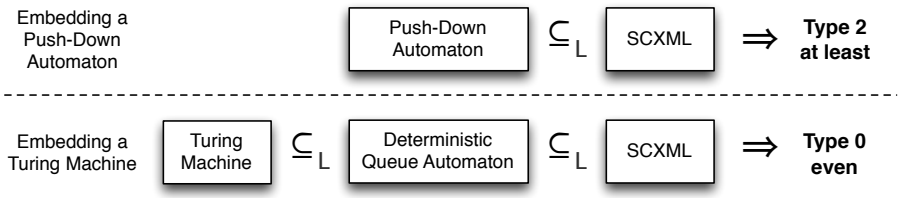


Figure 5.7.: Relation of SCXML with other automatons and implications shown in this section.

What we do not show here, is the resulting equivalence of a DFA and the restricted SCXML, i.e. we will not embed this bound SCXML subset in a DFA. But this is implicitly done in chapter 6 for the subset of SCXML that will be transformed onto the input language of the model-checker.

5.4.1 Embedding a Push-Down Automaton

In this section, we will embed a PDA (figure 5.8a) in SCXML, implying its computational model to be at least as expressive as any type-2 formalism in the Chomsky hierarchy.

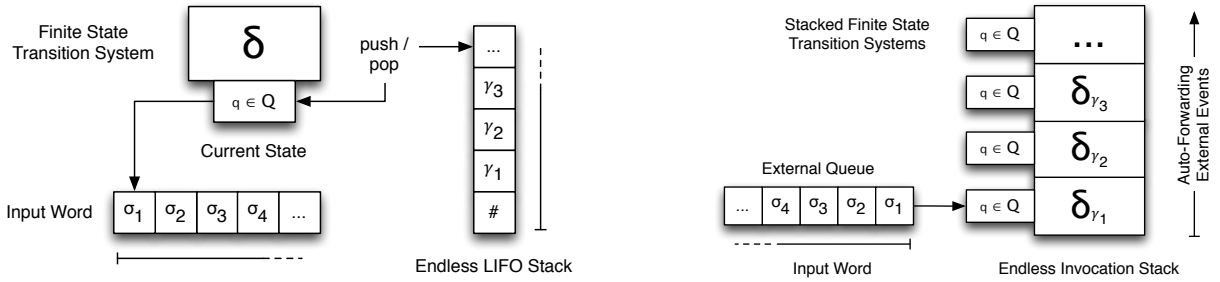
Definition 14 (Push-Down Automaton): *A PDA is formalized as* $\{Q, \Sigma, \Gamma, \delta, q_0, Z, F\}$*, where:*

- Q *is the set of states*
- Σ *the input alphabet*
- Γ *the stack alphabet*
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}_e(Q \times \Gamma^*)$ *the transition function ($\mathcal{P}_e$ the set of finite subsets [Sch97])*
- $q_0 \in Q$ *the initial state*
- $Z \in \Gamma$ *the initial stack symbol*
- $F \subseteq Q$ *the accepting states*

We can obviously model Q as the set of states and the start state q_0, as well as the accepting states F in SCXML. The input alphabet Σ will be a set of events with respective names and Γ, as the stack alphabet, will be a set of special SCXML documents, pushed onto the stack by invoking them. The invocation depth is potentially unlimited and can be abused to model the stack from the PDA (see figure 5.8b). We actually already embedded a PDA in section 4.7, when we implemented history stacks via invoked SCXML documents.

In general, the transition function δ will pop one element from the stack and, in accordance with the concrete transition rules given, write a new set of symbols onto the stack. With SCXML, popping an element from the stack is equivalent to terminating the topmost SCXML document, whereas writing a set of symbols corresponds to recursively invoking additional SCXML interpreters. We could model this behavior by passing the set of symbols to be written (i.e. the remaining SCXML interpreters to be invoked) via a list in a `<param>` to the `<invoke>`, but this would require the presence of a data-model with respective expressiveness, which would dilute the argument. Instead, we will conceive the pushing of a set of symbols onto the stack as invoking only the interpreter corresponding to the topmost stack symbol and enter a sequence of states, wherein the termination of this topmost interpreter will cause the immediate invocation of the next interpreter, corresponding to the next symbol to be pushed. That is, pushing e.g. `ab` onto the stack would be modeled as starting interpreter `a` and, upon its termination, invoking interpreter `b`.

The quintessential example for a language that can be recognized by a PDA but not a DFA is $L = \{a^n b^n\}$. Listing 5.5 / 5.6 displays an SCXML document without any data-model that recognizes this exact language, implying SCXML to be at least as expressive as a PDA.



(a) Schematic of a PDA.

(b) PDA stack with invoked SCXML interpreters.

Figure 5.8.: Embedding a PDA in SCXML.

```
1  <scxml>
2    <initial>
3      <transition target="onTop">
4        <send event="a" delay="1s" />
5        <send event="a" delay="2s" />
6        <send event="b" delay="3s"/>
7        <send event="b" delay="4s"/>
8      </transition>
9    </initial>
10
11   <state id="onTop">
12     <transition event="a" target="inBetween" />
13   </state>
14
15   <state id="inBetween">
16     <invoke src="pdaPush.scxml" autoforward="true" />
17     <transition event="done.invoke" target="accept" />
18   </state>
19
20   <final id="accept" />
21 </scxml>
```

We start as the finite state transition system on top of the stack and are responsible for events. We only wait for an `a` to push another transition system on the stack.

We invoke the transition system in listing 5.6 to the top of the stack, forward all events received, and wait for its completion.

Listing 5.5: Topmost transition system to recognize $L = \{a^n b^n\}$.

```
1  <scxml>
2    <state id="onTop">
3      <transition event="a" target="inBetween" />
4      <transition event="b" target="popped" />
5    </state>
6
7    <state id="inBetween">
8      <invoke src="pdaPush.scxml" id="stack" autoforward="true" />
9      <transition event="done.invoke" target="onTop" />
10   </state>
11
12   <final id="popped" />
13 </scxml>
```

When we on top of the stack, we are responsible to read input symbols as handling events. Reading an `a` will push another transition system, reading a `b` will pop this one.

Invoke another instance of this transition system to the top of the stack and wait for its completion.

Listing 5.6: Transition system `pdaPush.scxml` as it will be invoked onto the stack.

The input word is constituted by a series of external events, named `a` and `b` respectively. For every reception of an `a`, a new SCXML interpreter is pushed onto the stack, with event forwarding enabled, and the former top-most interpreter transitions into the state `inBetween`, where it ignores all events but the completion of the interpreter on the stack above. If an event `b` is received, all but the top-most interpreter will ignore it, and the top-most interpreter will terminate, causing the next interpreter on the stack to transition to `onTop` again. There are some impurities with the approach, in fact every language starting with an `a` and containing more `b`'s than `a`'s is accepted, still the basic idea stands and it is just a matter of introducing respective failure states to account for these. The important thing is that we can abuse the unbound invocation depth in SCXML to model the stack from a PDA, thus surpassing DFA expressiveness.

This already implies that we can inhibit the potential to embed a PDA via the invocation stack by either limiting the nesting depth for invocations or prevent recursive invocations. Both solutions are equivalent as we could always implicitly *rename* recursively invoked SCXML interpreters.

5.4.2 Equivalence to Turing Machines

In this section we will show that SCXML, even without any data-model, can perform the same operations as a DQA, which is equivalent to a Turing machine (Type-0 in the Chomsky hierarchy). This is possible by abusing one of the state-chart's queues, necessary for *broadcast communication* among states together with SCXML's semantic variance of instantaneous states, to store an infinite amount of symbols from the tape alphabet. Formally, we only need the one-directional implication of SCXML being type-0 by embedding a Turing machine in a DQA and, in turn, a DQA in SCXML.

5.4.2.1 Turing Machines

We start by defining a Turing machine as the archetypical reference automaton to recognize recursively enumerable (Type-0) languages. A Turing machine (figure 5.9) consists of an infinite, left-bound tape and a read-write head that can be moved along this tape. A control automaton given as a finite transitioning system will read the current cell's content and, with regard to its internal state, write a new or the same symbol into the cell and (optionally) move the read-write head by one step.

Definition 15 (Turing Machine): *A Turing machine can be formalized as the septuple* $\{Q, \Sigma, \Gamma, \delta, q_0, b, F\}$*, where:*

- Q *is the set of states*
- Σ *the input alphabet*
- $\Gamma \supset \Sigma$ *the tape alphabet*
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, 0, R\}$ *the transition function*
- $q_0 \in Q$ *the initial state*
- $b \in \Gamma \setminus \Sigma$ *the blank symbol*
- $F \subseteq Q$ *the accepting states*

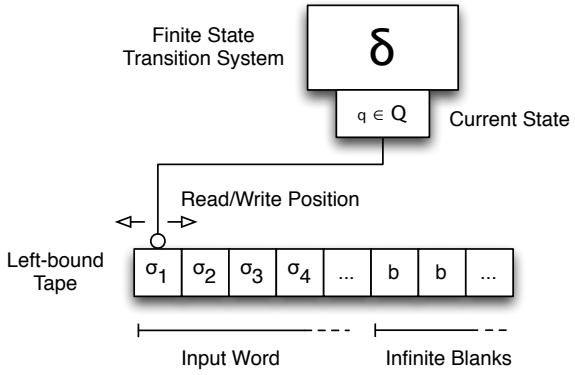


Figure 5.9.: Schematic of a Turing Machine.

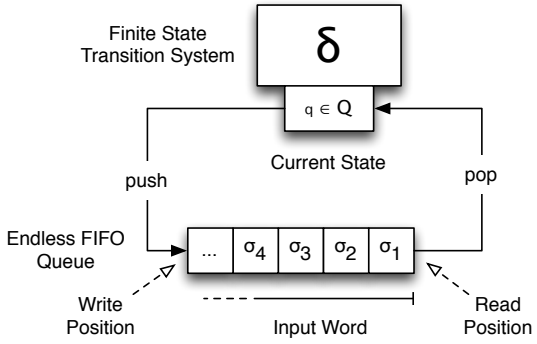


Figure 5.10.: Schematic of a Deterministic Queue Automaton.

At the beginning, the tape contains the input word followed by an infinite numbers of blanks. The read-write head is at the tape's first cell and the control automaton in the first state. The machine's configuration is changed according to the transitions given in δ. The Turing machine accepts a given input word if its configuration reaches a final state $q \in F$ with an empty tape. Sometimes the set of final states is dropped in the definition and acceptance is defined via the empty band only.

It is not exactly obvious, how a compliant SCXML interpreter without a data-model could mimic the behavior of such a Turing machine. While it is no problem to model the state transition system, there is no direct correspondence to the infinite, left-bound tape and the movable read-write head. Here, it is more constructive to show equivalence of SCXML with a DQA as an equivalent to a Turing machine.

5.4.2.2 Deterministic Queue Automaton

A more suitable analogue of an SCXML interpreter is the Deterministic Queue Automaton (DQA). Its behavior and formal definition is very similar to a Turing machine, but the infinite, left-bound tape is replaced by a first-in first-out (FIFO) queue with only push and pop operations available.

Definition 16 (Deterministic Queue Automaton): *A DQA is defined as the septuple* $\{Q, \Sigma, \Gamma, \delta, q_0, b, F\}$*, where:*

- Q *is the set of states*
- Σ *the input alphabet*
- $\Gamma \supset \Sigma$ *the queue alphabet*
- $\delta : Q \times (\Gamma \cup \epsilon) \to Q \times (\Gamma \cup \epsilon)$ *the transition function*
- $q_0 \in Q$ *the initial state*
- $b \in \Gamma \setminus \Sigma$ *the initial queue symbol*
- $F \subseteq Q$ *the accepting states*

Most semantics carry over from the Turing machine, only the transition function differs as the tape is replaced by a queue. If δ consumes an actual symbol ($x \in \Gamma$) a pop operation from the front of the queue is performed and, similarly, if δ produces a symbol, a push operation to the back of the queue is performed. Equivalence to a Turing machine (TM) is shown by modeling a DQA in the formalism of a Turing machine and vice versa. A proof of this equivalence is a common exercise for university courses in theoretical computer science and described in various lecture notes. The argument is as follows:

$TM \subseteq DQA$**:** As most definitions are equivalent, we only have to show that we can emulate the semantics of a band on the queue. In a DQA, the read-write head is actually split into a reader fixed to the front of the queue and a writer fixed to its back.

To simulate "write x and move left" we can simply pop the current symbol and push the new symbol onto the stack. This will implicitly move the reader and writer one position to the left as all of the queue scrolls one cell beneath their fixed positions.

The more complicated operation is "write x and move right" as we need to scroll through the complete queue. To support this operation, we need to introduce a new set of queue symbols: For every $x \in \Gamma_{TM}$ we define $\widehat{x}$ as the simulated TM's tape symbol where its read-write head is positioned. Now we introduce a new set of states to buffer the last read operation in order to prepend the new symbol. When we arrive at the TM's read-write marker we will, push the buffered symbol before pushing the unmarked equivalent of the current symbol.

$DQA \subseteq TM$**:** We can simulate a DQA using a 2-band TM (which is equivalent to a regular TM): As with the opposite direction, we only need to show that we can simulate the push/pop semantics of the queue on our bands. We will assume that the first band initially contains the input word. For a better intuition we will envision the band as a cyclic buffer, that is, moving the head onto the right-most blank symbol will reset its position to the first cell. Now, to pop a symbol, we mark the current position, copy all of the first band's contents to the second band with the exception of the marked symbol and reassume a position of the read/write head, one symbol to the left. Similarly, to push a symbol, we again mark the current position, copy the band and insert the symbol to be pushed at the marked position. Afterwards we reassume the read/write head's position one cell to the right.

5.4.2.3 State Chart XML Automaton

To show that SCXML can recognize the same languages as Turing machines, we have to show that we can embed a DQA as an equivalent computational formalism. For this implication, it is valid to regard only a required subset of SCXML's semantic features. As such, we can simplify the automaton, described by SCXML as the one depicted in figure 5.11 and formalized as follows:

- Q is a set of atomic `<state>` elements.
- Σ the input alphabet is a set of event names.
- $\Gamma \supset \Sigma$ the queue alphabet as an additional set of event names not yet employed in the input alphabet.
- $\delta : Q \times (\Gamma \cup \epsilon) \to Q \times (\Gamma \cup \epsilon)$ the transition function modeled via `<transition>` elements with `<send>` or `<raise>` as executable content to push symbols.
- $q_0 \in Q$ the initial state as the first `<state>` in document order or referenced via the `initial` attribute at the outermost `<scxml>` element.
- $b \in \Gamma \setminus \Sigma$ the initial queue symbol as any free event name.
- $F \subseteq Q$ the accepting states as `<final>` states in SCXML.

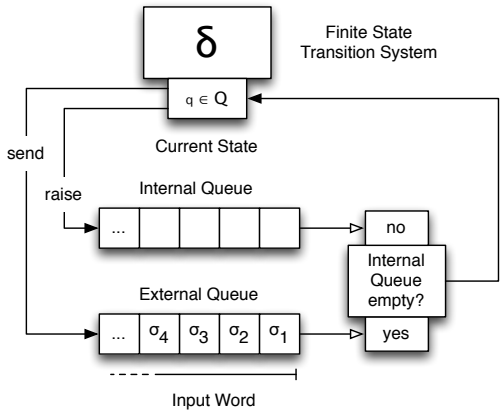


Figure 5.11.: Simplified automaton described by SCXML.

The basic idea to embed a DQA is to model the states of its finite transition system by encoding Q as atomic states, q_0 as the first state in document order and F as final states in SCXML. This subset of the formalization is even equivalent as we have shown above that we can encode a SCXML state-chart in an SCXML state-machine, thus no expressiveness is gained by the state-chart representation.

$DQA \subseteq SCXML$**:**

The input alphabet Σ will be constituted of names for events and the queue alphabet Γ is a superset of those. Any free event name can form the initial queue symbol b. Care has to be taken, not to evoke the more refined semantics of SCXML event name matching (see definition 9), trivially done by disallowing dots in the event names. Any free event name can form the initial queue symbol b.

For the transition function δ to work as with a DQA we need an approach to push and pop symbols into and from an infinite FIFO queue in addition to the trivial functionality to transition between state. SCXML actually defines two, potentially endless, FIFO queues: the internal and the external event queue, addressable via `<raise>` and `<send>` respectively, whereby events from the external queue are only popped if the internal queue is empty.

Formally, it makes no difference whether we start with the input word on the internal or the external queue. We can, however, not copy the external queue onto the internal queue as popping an event will always favor the internal queue. Using the external queue exclusively, is already sufficient to model δ from a DQA. Pushing a symbol is to `<send>` the respective event with an empty target attribute as part of a transition. Popping a symbol is done automatically whenever there are no more spontaneous transitions or those enabled by the current event. Popping the empty symbol can be achieved by a simple spontaneous transition to the next, relevant state.

$SCXML \subseteq DQA$**:** Showing the reverse implication, i.e. an SCXML interpreter can be modeled in a DQA, would actually show equivalence of the two computational formalisms. Though, we cannot do so here as our simplified formal model does not capture all of SCXML's formal semantics.

We did, however show the equivalence of a DQA with a Turing machine and, therefore, can simply invoke the Church–Turing thesis [Kle52] stating that *every effectively calculable function (effectively decidable predicate) is general recursive* and as such decidable by a Turing machine. Or simply put: if something is computable at all, it is computable by a Turing machine. As SCXML is certainly computable, it can be embedded in a DQA.

5.4.3 Retaining DFA Equivalence

If we are to employ the formalism of model-checking about to be introduced in section 6.2 with SCXML documents, we are required to limit its computational model to type-3 on the Chomsky hierarchy, i.e. DFA equivalence. As every system with a finite state space can be expressed by a DFA, simply by enumerating the states and introducing respective transitions, all we need to do is to disallow any *unbound concepts* in the semantics of SCXML. In particular, this means to limit invocation depth to inhibit the embedding of a PDA and to limit the length of both, the internal and external queue to prevent the embedding of a DQA. For the length of the event queues, we will make an effort to

determine their maximum length in the section 6.4.10, but in general, this is not possible and the user will have to adjust this limit to a sane number.

Do note that all these considerations are only valid for SCXML documents without a data-model as an embedded scripting language. Providing a data-model will, in most cases, introduce Turing completeness regardless of any limits put to the invocation depth or queue length in SCXML as a single `<script>` element would suffice. In fact, this is why we will have to introduce the PROMELA data-model as a finite enumerable scripting language in section 6.3.

5.5 Dynamic Minimization of State Chart XML State-Machines

To conclude this chapter, this section will briefly present an approach to minimize the SCXML state-machines from section 5.3. In general, it is not possible to reduce a SCXML state-chart onto a minimal, functionally equivalent state-chart. Some classes of redundancy can be identified, e.g. specific transitions that can never be part of an optimal transition set, obviously unreachable states or executable content. But in general, even without events arriving from external entities, features such as dynamic event names and targets via the `<send>` element's `eventexpr` and `targetexpr` respectively make it impossible to enumerate all sequences of events without enumerating the complete state-space. With SCXML being Turing complete (compare previous section), this is impossible.

The transformation to state-machines was only minimal insofar that only global states that are actually reachable via transition sets were created. Nevertheless, some transition sets might still be superfluous e.g. when all incoming paths raise an internal event that will always enable a specific transition set. However, if (i) the state-machine does not rely on external events to trigger transitions or (ii) the set of sequences of external events can be enumerated and (iii) the interpreter terminates, a state-machine can be reduced by marking visited elements at runtime and subsequently removing removing unmarked elements. This is possible with the state-machine representation but not obvious for the state-chart representation, recommending this technique when a state-machine representation is required anyway.

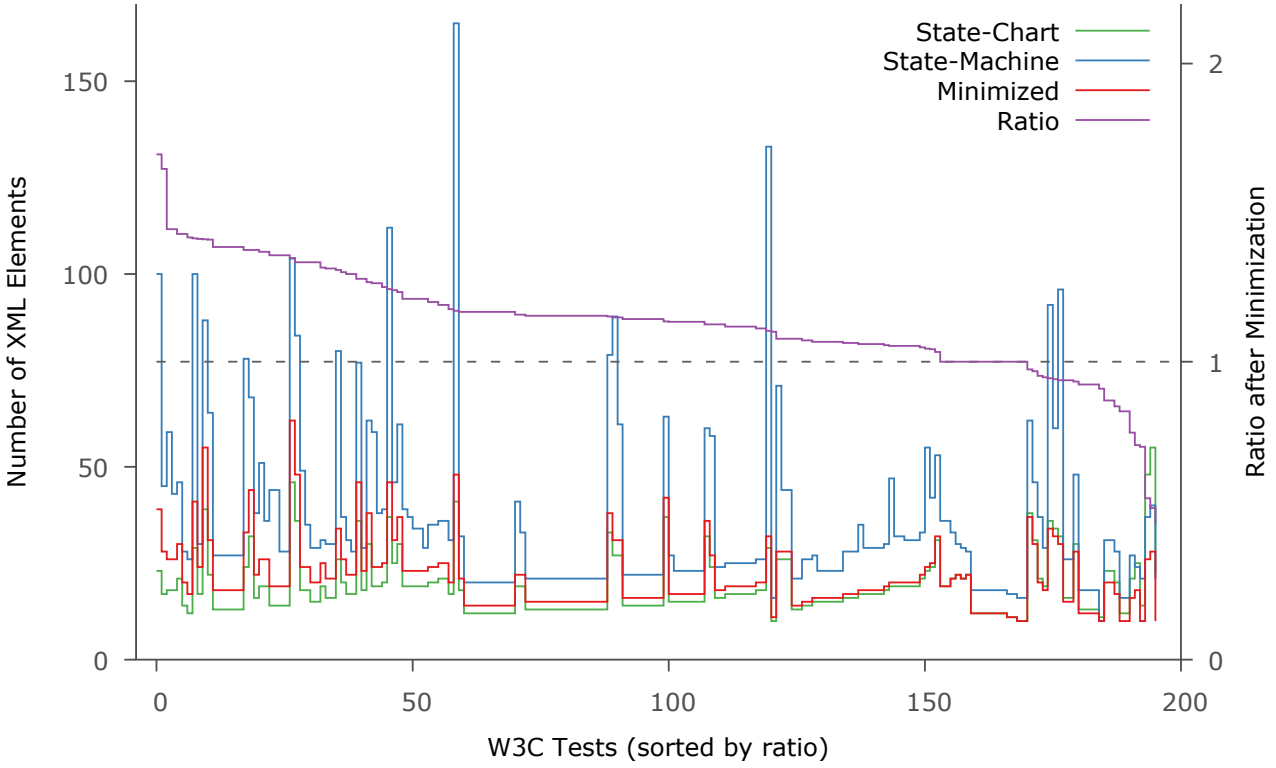


Figure 5.12.: Number of XML elements in state-chart → state-machine → minimized W3C tests.

The criteria above are indeed given for the W3C tests accompanying the SCXML standard, as their transitions are triggered solely by events originating from the state-chart itself. Figure 5.12 gives an impression about the benefits of this technique. For all 191 automatic and 5 manual W3C tests of the ECMAScript data-model, the number of XML elements in the original SCXML state-chart is counted. The individual tests are then flattened onto a state-machine via the transformation described in section 5.3 and afterwards minimized by removing XML elements that were never

visited by the interpreter. Here, the mean factor for the increase in XML elements when transforming from the original state-chart to the flattened state-machine is `1.84` (σ: `0.55`), whereas the subsequent minimization reduces this amount by a factor of `0.64` (σ: `0.11`) for an overall mean of `1.18`. However, being based solely on the W3C tests, these values can give only an idea about orders of magnitude and are difficult to generalize.

The more important aspect is, that all W3C tests are passed again, implying a general applicability of the approach for all functional requirements of SCXML. Thus, every terminating state-chart with an enumerable set of input sequences can be reduced by the following process:

1. Transform the SCXML state-chart into an SCXML state-machine
2. Start the SCXML interpreter with the state-machine
3. Deliver one sequence of input events
4. Mark XML elements visited during of processing
5. Repeat steps 2-4 until all sequences of input were processed at least once
6. Remove unmarked elements

This process ensures that every marked XML element was actually used by one of the input sequences and the unmarked elements can be removed with all the original state-chart's functionality retained.

The previous chapter introduced a transformation of State Chart eXtensible Markup Language (SCXML) state-charts onto semantically equivalent SCXML state-machines with slight adaptations required for an interpreter. We have also identified the computational model of SCXML as Turing complete and argued for some limitations to retain Deterministic Finite Automaton (DFA) equivalence as it will be required to apply formal verification with our approach. With these prerequisites given, this chapter will tackle the next proposition in the main argument (see figure 6.1), i.e. make SCXML documents as dialog models accessible for the formalisms of temporal logic.

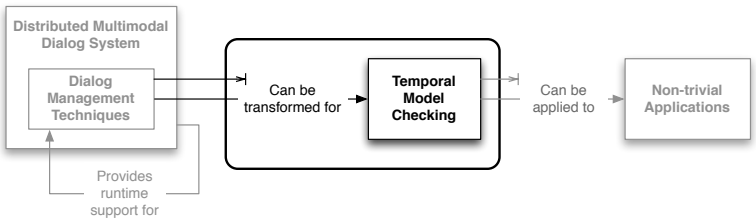


Figure 6.1.: This chapter tackles the proposition "*These dialog models can be made accessible to the formalisms of temporal logic.*" from the main argument of this thesis (cf. figure 1.1).

This can be very beneficial or even crucial for some interactions in pervasive environments as it enables a dialog author to guarantee certain aspects of a dialog's behavior and thus the behavior of the resulting system. Being able to guarantee, e.g., that the user will always be able to access help or that a given user input always results in a given behavior regardless of the system's state is of obvious benefit. In general, there are different techniques and formalisms available for the analysis of such a system, each allowing to ascertain various classes of statements. The approaches fall into two general classes:

1. Dynamic program analysis performed at the system's runtime and
2. Static analysis performed on the system's syntactical description.

For a dynamic analysis, constraints and assertions that need to hold are usually written in the program's description or introduced via an instrumented runtime. The program is then run and exposed to a series of inputs. If the constraints are fulfilled and the assertions hold, one can say that the program fulfills these assertions for the given input sequence. In the general case, this is not exhaustive as there can be an infinite number of input sequences. If, however, the input sequences are finite e.g. when only m different types of inputs are processed and the program is restarted or resets to a previous state after the nth input, this class of program verification can very well be exhaustive.

Static analysis is being performed on the system description as such without it necessarily being interpreted. There are different techniques, ranging from informal heuristics to exact formalisms. An important subclass of static analysis is the formal verification of a program with regard to a set of specifications.

Early approaches trace back to Floyd-Hoare logic [Flo67][Hoa69], wherein a computer program is manually annotated with formal pre- and post-conditions for which a calculus was provided to reason about the program's correctness. More recent approaches became known as *model checking* [CE82, QS82, VW86], wherein statements of the form $(M, s) \models p$ can be validated. With M being a suitable representation of the system, s a start state in this representation and p a property to be validated. The applicability and expressiveness of the approach depends on whether the system can be expressed in the given formalism and the semantic richness of p. For a survey of model checking and its historical development refer to [JM09]. In the following sections we will show, that:

1. There is a transformation from any SCXML document with the empty data-model onto M.
2. We can provide an expressive data-model that can still be expressed in M.
3. p is expressive enough to ascertain *interesting* properties.

6.1 Related Work

In the following, we will transform a large subset of SCXML onto PRocess MEta LAnguage (PROMELA) as the input language of the SPIN model-checker. As this constitutes the major contribution of this thesis, it is useful to identify related work and more clearly delineate the contribution. The overall approach to verify systems by (semi-) automatically transforming them onto input languages of model checkers is quite popular and respective scientific work numerous, e.g. [ABB+96, HT10, JAP+11]. Apart from PROMELA, another popular transformation target is the input language of the Symbolic Model Verifier (SMV) model checker and its derivates. Where SMV allows to verify systems with Computation Tree Logic (CTL), SPIN enables the verification with Linear Temporal Logic

(LTL) expressions (see section 6.2.1 below). Introducing related approaches explicitly here, serves two purposes: (i) to show that the general approach is deemed useful not only by this thesis, but a plethora of related work and (ii) to illustrate the problem with the uncountable variances in state-chart semantics, as many approaches essentially have the same function for different formalizations: transform a description of state-charts onto the input language of a model-checker. This last point strengthens the case of standardized semantics for state-charts as is attempted with SCXML.

These transformations can be classified by (i) the formal semantics of their domain,(ii) the scope of the transformation as the set of language features they retain and (iii) the codomain of the transformation as the specific target language and the respective formalisms for verification they enable. However, none of these dimensions provides a convincing classification: The domain is most often the distinguishing contribution and, therefore, differs in virtually all publications. The transformation is actually dependent on the domain (or rather vice versa) as it defines the source set of expression that can be transformed. Finally, the codomain is usually just an input language (or subset thereof) for one of the various model-checkers and their distribution (e.g. SMV versus SPIN input languages) suggests an apparent ease for interchangeability.

There are two relevant surveys which attempt to provide an overview of the respective techniques and their applications. The first one by Pnueli [Pnu86] lists various general applications of temporal logic to formally verify reactive systems, but predates the visual formalism of state-charts from Harel by one year and, as such, does not account for their specifics. The second survey by Bhaduri and Ramesh [BR04] is more current and explicitly lists approaches related to model checking for state-charts. It starts to differentiate the various approaches by their target language, but gets somewhat blurry once the obvious input languages for SMV, SPIN are exhausted. Another possible classification, which we will follow, is to align the domain of the presented transformation with the general context in which it is introduced. Here, foremost comparisons to "Statemate", as one of the first commercially available state-chart modeling environments and Unified Modeling Language (UML) state-diagrams are of note.

Statemate related

The Statemate system was developed by I-Logix, a company founded by Harel and Pnueli in 1987 and provided a platform to functionally decompose system descriptions into state-charts and some other related artifacts. As one of the earliest de-facto interpreters of state-charts, it had a certain normative function with regard to the semantics of state-charts [HN96] for many of the transformations from state-charts onto the input language of a model-checker.

One of the earliest non-trivial systems, described in a state-chart variant and verified with temporal logic was the aircraft collision avoidance system "TCAS II". The system is modeled in Requirements State Machine Language (RSML) [LHHR94] and subsequently transformed onto the input language of the SMV model-checker [ABB$^+$96]. The initial description of RSML is in reference to Statemate and its state-chart semantics, but various adaptations were deemed necessary in order to accommodate for unambiguous semantics with a strong mathematical basis and intelligibility of the system's description.

In [HK97], the authors present a set of tools related to the Software Cost Reduction (SCR) notation [Hag89], a tabular specification for state-machines to describe reactive systems. They introduce a series of techniques, both for static and dynamic verification of systems described in SCR and detail an approach to transform these onto PROMELA for the SPIN model-checker [BH97]. Their work is related to Statemate only insofar that they delineate their notation from Harel state-charts.

Another related work from Mikk et al. [MLS97] is explicitly employing the Statemate formalization of state-charts from Harel & Namaad [HN96] and introduces an intermediate representation of Extended Hierarchical Automatons (EHA) which is subsequently transformed onto PROMELA [MLSH98]. This intermediate EHA representation is not unlike the SCXML state-machines introduced in section 5.3 but introduces a series of severe limitations as to the language features of Statemate state-charts that are approachable. E.g. it will not allow interlevel transitions, nor does it account for history states.

Finally, another approach to explicitly transform the Statemate semantics of state-charts is found in the work of Clarke and Heinle [CH00]. They present their STP translator, which does not attempt to transform the largest possible set of state-chart semantics onto the input language of a model-checker (SMV in this case), but attempts to retain the state-charts hierarchical structure to preserve the levels abstractions modeled via state hierarchies.

UML related

Another class of approaches starts the transformation onto the input language of the various model-checkers in the domain of UML state-charts. The major problem with all of these approaches is the fact that there are no formal semantics for UML state-charts and all respective contributions have to start by defining such

a formal semantic or by referring to one. A survey from 2005 by Crane and Dingel [CD05] compares 26 different formalization for the semantics of UML state-charts, giving an impression about the sheer amount of possible variations. A noteworthy formalization, though without a transformation onto the input language of a model checker is given by van der Beeck [vdB01], who identified the 19 different variations in state-chart formalizations [vdB94] we presented in section 4.1 to classify SCXML. In fact, the work of van der Beeck is in reference to other formalizations, some with transformations onto model-checkers and attempts a unification of their semantics, as many others did as well.

One of the first attempts to formally define the semantics of UML state-charts can be found in the work of Latella et al. [LMM99b]. As with the work of Mikk et al. [MLS97], their formalization employs the mathematical construct of hierarchical automatons and only regards a subset of UML state-chart language features, e.g. they do not consider histories, actions, activities nor variables or complex events. This restricted subset of the UML state-chart language is subsequently transformed onto PROMELA for the SPIN model-checker [LMM99a].

An attempt for a complete formalization of UML state-chart semantics is found in [LP99]. It is noteworthy for its intermediate representation of state-charts as state-machines, very comparable to our transformation described in section 5.3. The approach is implemented in the vUML authoring environment for UML state-charts and supports a transformation onto PROMELA for the SPIN model-checker. Even though, the authors claim a complete formalization of UML state-charts, they do relativize their approach when transforming onto PROMELA as, e.g. the creation of new objects and timed events are explicitly excluded. This work is explicitly referenced in the work of Latella et al. [LMM99a] introduced above, as the authors claim for the flattened state-machine representation to be more approachable to a transformation when compared to the hierarchical automatons, a claim Latella et al. dispute.

There are many more approaches to formally define the semantics of UML state-charts to enable a transformation onto the input language of a model-checker, e.g. [MC01, NPS09, AYK+13] and also more attempts to unify state-chart semantics into some common formalization, e.g. [Esh09], but the plethora of approaches indicates a reoccurring theme, wherein an arbitrary formalization is deemed beneficial for one reason or another and subsequently transformed onto the input language of a model-checker. And indeed, the contribution presented in this thesis is at risk of being just another attempt to verify an arbitrary formalization. This is why the majority of the first part of this thesis attempted to convince the reader of the relevance of SCXML as the basis for our transformation, both, as a deterministic semantic for state-charts in general and as a description language for dialog models in particular: a renowned organization such as the World Wide Web Consortium (W3C) might lend enough authority to SCXML as yet another arbitrary but fixed formalization of state-chart semantics to succeed.

Apart from the transformation for the various state-chart formalizations onto the input languages of model-checkers, there is also related work dealing explicitly with a transformation of dialog models, not necessarily in the form of state-charts, onto such languages.

An early approach by Abows et al. [AWM95] employs Olsen's Propositional Production System (PPS) [Ols90], a tabular format to specify dialog models to create state transition networks, which are subsequently transformed for the SMV model-checker. In [PS01], Paterno et al. describe an overall development process to express user interfaces starting from ConcurTaskTrees and transform these onto LOTOS as the input language of the CADP model checker. Another approach by Shi et al. [SRB05] employs Communicating Sequential Processes (CSP) to express the dialog model for a Spoken Dialog System (SDS), which are subsequently transformed for the Failures-Divergence Refinement (FDR) model-checker. In a more recent publication, Brat et al. [BMP13] model an interactive, safety-critical application to display a weather radar in a plane's cockpit as an Interactive Cooperative Object (ICO) model and employ the Java Path Finder model-checker.

As with the transformation of the different state-chart variants above, the main idea is to describe the user interface in a formalism that can be made accessible to the formalisms of model-checking. A more complete overview of related approaches is found in [BBS13].

6.2 Model-Checking

All model checking techniques allow to approach statements of the form $M \models p$. Where M is a formal representation of the program under consideration and p a formal property to be proven. In order to get a better understanding of the techniques involved with model checking and their limitations, we will describe one of the popular approaches for formal verification of programs, namely *automata-based model checking of linear temporal logic expressions*.

The idea with this type of model checking is to represent both M and p as special finite automatons. This is always possible for programs with a finite state space as we can simply enumerate all states and introduce transitions with

respect to the possible inputs in a given state and the main reason why we were required to retain DFA equivalence in section 5.4. Model checking usually makes the additional assumption that the system might run for an infinite amount of time and as such, employs Büchi automatons [Büc62] as an extension of DFA on ω-words as inputs of infinite length.

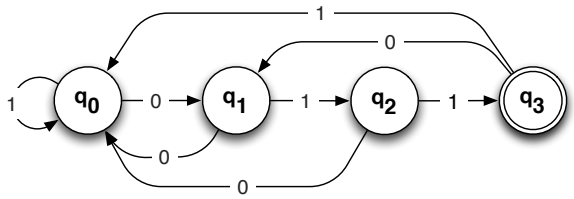


Figure 6.2.: Deterministic Büchi Automaton accepting any ω-word containing (011) infinitely often.

Definition 17 (Deterministic Büchi Automaton): *A deterministic Büchi automaton is the quintupel* $\mathcal{B} = (Q, \Sigma, \delta, q_0, F)$ *with*

- Q *the automaton's finite state set.*
- Σ *the set of valid input symbols.*
- δ *the transition relation as* $Q \times \Sigma \to Q$.
- $q_0 \in Q$ *the start state.*
- $F \subseteq Q$ *the set of accepting states.*

The definition is very similar to the original definition for DFAs, the sole exception being the acceptance condition for an input word: a Büchi automaton will accept an input ω-word of infinite length, if processing passes one of the accepting states infinitely often. Now, for a system M to fulfill a property p, it needs to be shown that every ω-word as a valid *execution trace* through M fulfills the property formalized by p. Therefore:

$$M \models p \Leftrightarrow L(\mathcal{B}(M)) \cap \overline{L(\mathcal{B}(p))} = \emptyset \tag{6.1}$$

We negated the language recognized by the Büchi automaton for the formal property $\mathcal{B}(p)$ and formed the intersection with the language recognized by system description $\mathcal{B}(M)$, if this resulting language is not empty, we have a set of input sequences that will invalidate the property p. It is implied here, that Büchi automatons are closed under complement and intersection, which is the case [Büc62, PSVW87, Kur87].

Describing a system or a property thereof directly as Büchi automatons is hardly a viable approach to ascertain properties of a dialog system. To make model-checking more accessible to authors, the following sections will introduce equivalent and more suitable representation for which automated projections onto Büchi automatons exist.

6.2.1 Formal Property Specifications

Formal specifications for properties of a system are usually given in a temporal logic. There are two complementary classes of temporal logic that are applicable to express these formal properties p a system M has to fulfill: Linear Temporal Logic (LTL) and Computation Tree Logic (CTL). There are some similarities, as both are subsets of *Computation Tree Logic** (see figure 6.3).

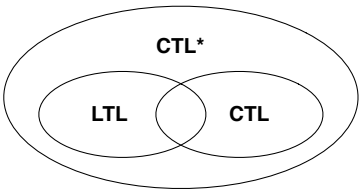


Figure 6.3.: Hierarchy of temporal logic.

Both classes, CTL and LTL can be used to make statements about a system's properties given as a set of boolean values in a system's computation tree and put them in relation over time via their execution traces. We will start by formally introducing LTL and subsequently describe the differences to CTL. Let $p_1 \dots p_n$ be a set of boolean properties of a system, then the following are valid LTL expressions:

The **atomic property** p_i is a valid expressions and true if p_i is true.

For two valid LTL expression ψ and ϕ, the following **boolean compositions** are valid LTL expressions:

Conjunction: $\psi \wedge \phi$ is true if both ψ and ϕ are true.

Disjunction: $\psi \vee \phi$ is true if either ψ or ϕ is true.

Negation: $\neg\psi$ is true if ψ is false.

Implication: $\psi \Rightarrow \phi$ is true if $\neg\psi \vee \phi$ is true.

For two valid LTL expression ψ and ϕ, the following **temporal operators** form valid LTL expressions:

Next: $X\psi$ is true if ψ is true in all of the directly following steps.

Future: $F\psi$ is true if all paths lead to a state in which ψ is true. This is sometimes also referred to as *eventually* or *finally* and abbreviated with $\Diamond$ the symbol.

Globally: $G\psi$ is true if ψ is always true on all paths. Also referred to as *always* and abbreviated with the $\Box$ symbol.

Until: $\psi U \phi$ is true if ψ is true until ϕ becomes true on all paths. The until operator is sometimes differentiated from a *weak until*, which can be read as *unless*. With the weak until / unless, it is not required for ϕ ever to become true at all.

Release: $\psi R \phi$ is true if ϕ is true until released by ψ on all paths. The difference to *until* or *unless* is that there needs to be at least one state for which both expressions are true.

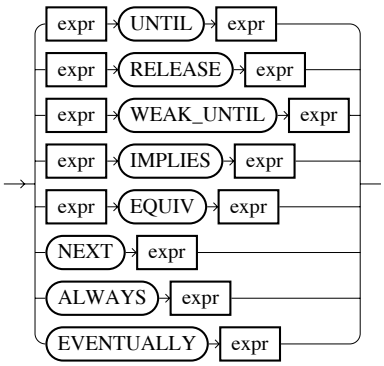


Figure 6.4.: Syntax diagram of LTL expressions in SPIN/PROMELA.

This set of operators is not minimal as some constructs can be expressed by compounding others but is more descriptive than the minimal set. Other formalizations specify other names for the operators, e.g. the syntax diagram in figure 6.4 describes the grammar rules for LTL expressions in the SPIN/PROMELA implementation. Every temporal operator (X, F, G, U, R) is implicitly all-quantified as it has to hold for all paths (execution traces) in a system's computation tree. It has been shown that every LTL expression is equivalent to a Büchi automaton [VW94, GPVW96] that accepts all ω words that satisfy the temporal logic expression. Therefore, dialog authors can employ LTL expressions to provide formal properties a system has to fulfill.

The syntax of CTL is quite similar. Atomic properties and boolean compositions are defined as with LTL but the set of temporal operators differs somewhat:

For two valid CTL expression ψ and ϕ, the following **temporal operators** form valid CTL expressions:

Exists Next: $EX\psi$ is true if ψ is true in one of the directly following steps.

Always Next: $AX\psi$ is true if ψ is true in all of the directly following steps.

Exists in Future: $EF\psi$ is true if there is a path to a state in which ψ is true.

Always in Future: $AF\psi$ is true if all paths lead to a state in which ψ is true.

Exists Globally: $EG\psi$ is true if there is one path on which ψ is always true.

Always Globally: $AG\psi$ is true if ψ is always true on all paths.

Exists Until: $E(\psi U\phi)$ is true if there is a path on which ψ is true until ϕ becomes true.

Always Until: $A(\psi U\phi)$ is true if ψ is true until ϕ becomes true on all paths.

Exists Release: $E(\psi R\phi)$ is true if there is one path where ϕ is true until a released by ψ.

Always Release: $A(\psi R\phi)$ is true if ϕ is true until a released by ψ on all paths.

There are temporal expressions in LTL that are inexpressible in CTL and vice versa. For instance, every temporal operator in CTL consists of a path quantor (A, E) and an operator (X, F, G, U). It is not possible to nest the operators to form compound temporal operators, e.g. $AGFp_i$ to express that "on all paths there will always again be sequences of states for which p_i is true" is not expressible in CTL. On the other hand, some CTL cannot be expressed in LTL. For instance, there is no semantic feature to argue about the existence of a single execution trace with a given temporal property, as all expressions have to hold for all paths through the computation tree. A simple CTL expression such as EGp_i to validate that there is always at least one path for which p_i is eventually true is inexpressible in LTL as the property has to hold for *all* paths. In summary, LTL allows to express more complicated temporal claims for all execution traces, whereas CTL allows to argue about individual traces.

Whether CTL or LTL is more suited to express temporal properties of reactive systems has been a point of much debate in the model-checking community [Var01, CD89]. In [MP90], Manna and Pnueli argue that three basic types of temporal expression can "*cover the majority of properties one would ever wish to verify*", namely:

- *Invariance* to state that a given property P holds true for all execution traces, readily available in LTL as:
  `always P`
- *Response* to express that a given property P will always lead to the eventual occurrence of another property Q for all execution traces:
  `always (P => eventually Q)`
- *Precedence:* to state that, given a certain property P, another property R is always preceded by an interval in which Q was true (if R occurs at all):
  `always (P => Q unless R)`

In [FFC+10] it was observed that 70% of all their temporal claims for constraints related to a library system had a simple form of `always (P -> Q)`, something that is readily available in both temporal logic formalisms.

The patterns for LTL expressions in the distribution of the SPIN model-checker[1] provide a good, concluding intuition about the expressiveness of LTL. Let P, Q, R and S be any boolean expression (i.e. any property of a dialog model), $\texttt{until}_\texttt{S}$ the *strong until* and $\texttt{until}_\texttt{W}$ the *unless*, then the following are valid LTL expressions:

- P is false between Q and R:
  `always ((Q and !R and eventually R) -> (!P until`$_\texttt{S}$ `R))`
- P occurs at most twice:
  `(!P W (P until`$_\texttt{W}$ `(!P until`$_\texttt{W}$ `(P until`$_\texttt{W}$ `always !P))))`
- P becomes true after Q until R:
  `always (Q and !R -> (!R until`$_\texttt{W}$ `(P and !R)))`
- S precedes P between Q and R:
  `always ((Q and !R and eventually R) -> (!P until`$_\texttt{S}$ `(S or R)))`
- S responds to P between Q and R:
  `always ((Q and !R and eventually R) -> (P -> (!R until`$_\texttt{S}$ `(S and !R))) until`$_\texttt{S}$ `R)`

[1] from `Examples/LTL/patterns.pml` in the SPIN distribution

6.2.2 Formal System Models

The formalism of model-checking described above assumes that both, the formal property and the system to be tested can be expressed as Büchi automatons. With LTL, we already introduced an expressive formal language to specify temporal properties of a system. In this section we will briefly introduce Kripke structures as a representation for the behavior of a system as transitions through a series of boolean properties.

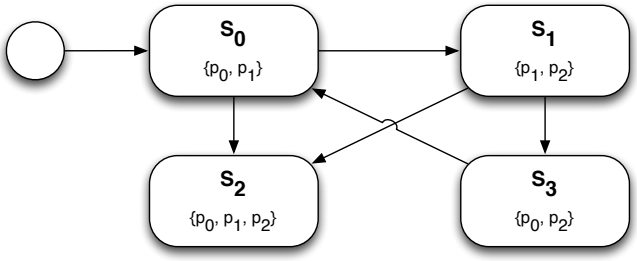


Figure 6.5.: Kripke structure formalizing the transitions of a system's atomic properties.

Definition 18 (Kripke Structure): *A Kripke structure is the quintupel* $\mathcal{K} = (S, I, AP, R, L)$ *with*

- S *a finite state set.*
- $I \subseteq S$ *the set of initial states.*
- AP *a set of atomic propositions.*
- R *a total transition relation with* $R \subseteq S \times S$ *such that there is a transition from every state.*
- L *a labeling function with* $L : S \to 2^{AP}$.

Such a Kripke structure, again, closely resembles finite automatons and can be interpreted as a Moore machine with no input alphabet and, as such, only unconditional transitions. It will only model sequences of states as they can occur in a system. The set of all paths through such a structure forms its ω-language as all sequences of assignment of boolean properties.

There is a direct correspondence of Kripke structures to Büchi automatons by simply moving the state's labels onto the transitions and introducing a new start state. As such, a system modeled as a Kripke structure can be used with the formalism for model checking described above. But still, modeling a system as a Kripke structure is hardly any more applicable for a dialog author than specifying the Büchi automaton directly. There are, however, additional transformations from high-level languages onto Kripke structures as we will see in the next section.

6.2.3 The PROMELA Language

With the formalism of Kripke structures and its correspondence to Büchi automatons, different approaches were established to provide even higher-level languages on top of Kripke structures for authors to express system models. One of this approaches is the PRocess MEta LAnguage (PROMELA) employed by the SPIN model-checker. PROMELA allows to express systems as concurrent processes, synchronized by sending events on channels or via global variables. The processes' modifications of the system's state can efficiently and implicitly be transformed onto a Kripke structure by the SPIN model-checker and validated via expressions in linear temporal logic. This section will briefly introduce the PROMELA language as the target for the transformation from SCXML state-machines and as the input language of SPIN.

The most essential property to enable model checking of PROMELA programs is its explicit bounding: all entities are bound with an upper limit to their size. This enables an enumeration of the system's state and thus a representation as a DFA and, per extension for never ending programs, a Büchi automaton.

Name	Minimum	Maximum
bit or bool	0	1
byte	0	255
short	$-2^{15}-1$	$2^{15}-1$
int	$-2^{31}-1$	$2^{31}-1$
unsigned : N (<32)	0	$2^{N}-1$

Table 6.1.: Data types in the PROMELA language.

All data types are integer with a bit-width varying from 1 to 32 bit (see table 6.1). These native data types can also form arrays of a fixed size and be composed into non-recursive, user-defined, compound structures.

Another very important property to understand the semantics of a PROMELA program is the *executability* of statements. In reference to Dijkstra's guarded command language [Dij75], statements in PROMELA are either executable or blocked. When multiple processes are running concurrently and multiple statements are executable, a PROMELA interpreter will "branch-out" and follow each variation of control flow, thus every possible interleaving of instructions is considered.

Some statements such as assignments, declarations, assertions as well as `break` and `goto` and the no-op `skip` are always executable. Every other statements can be blocked depending on its semantics:

- **Expressions** are executable if they evaluate to a non-zero value or boolean *true*.
- **Sending** on a channel is blocked, per default, if the channel is full. A runtime option for the SPIN interpreter allows to drop messages to full channels instead.
- **Receiving** from a channel is blocked if the channel is empty.
- **Timeout**s are executable if no other statement is executable, which is useful as a fall-through option in an `if` block.
- **If** and **do** blocks are executable if at least one of their conditions is executable. In contrast to their familiar semantics, the interpreter will branch out for every executable condition and follow every path during verification.

A PROMELA program, when interpreted by SPIN, starts by executing the `init` process, which usually just instantiates other, concurrent processes and waits for their completion. There are a few noteworthy omissions of features usually found in Turing complete languages that are unavailable, per design, in the PROMELA language:

- The language does not offer a data type for strings and, consequentially, none of the usual operation for strings such as concatenation, substring extraction or searching. We will see later how we can, nevertheless, represent strings and even provide a small set of operations.
- No floating point numbers are available to *encourage abstraction from the computational aspects of a distributed application*[2].
- There is no concept of *time*. All operations are assumed to be instantaneous as every possible interleaving is considered when verifying temporal claims. Our transformation will, however, honor the sequence of events as it results from delaying events via the `delay` attribute for `<send>` elements in SCXML.
- No source of indeterminism. Though, if the interpreter is run in simulation mode, as opposed to verification mode, only a single executable statement is selected randomly at a time and the interpreter will not branch-out.

Other language features are introduced as they are needed in the next section. Subsequently, we will use PROMELA as the target of our transformation from SCXML state-charts, enabling model-checking via SPIN. The following sections will detail this process.

[2] `http://spinroot.com/spin/Man/float.html`

In SCXML, a data-model provides embedded scripting capabilities and considerably enhances the expressiveness of the state-charts action language (see section 4.1.3). If we are to transform our complete SCXML state-chart onto a Kripke structure, we have to make sure that such a transformation also exists for the employed data-model. This is obviously the case for the trivial `null` data-model, which only mandates the `In` predicate to be available and does not mandate any other semantics. Beyond the trivial `null` data-model, all programming languages are eligible in principle, even despite their usual Turing complete nature, if we are careful not to provide unbound concepts. If every structure employed by the data-model implies or explicitly specifies an upper bound, it can be expressed by a Turing machine with a limited band and, per state-enumeration, a DFA and thus a Kripke structure. Most languages, however, do not imply such a bounding and there can be no DFA equivalent as the state-space cannot be finitely enumerated. In practice, though, such an upper bound *always* exists with current computers as available memory is always finite.

To exclude these considerations when providing an embedded scripting language via a data-model in SCXML, we will simply provide the PROMELA language itself and its explicit bounding of all language constructs. Using this approach, we can just insert the respective PROMELA snippets verbatim when transforming the state-chart.

The SCXML standard specifies several elements and attributes where the data-model will be considered to adapt its behavior during interpretation. Table 6.2 gives an overview of SCXML language features relying on the data-model, sorted by their type (compare also table 4.1).

Element	**Attribute**
Boolean Expressions	
`if`	`cond`
`elseif`	`cond`
`transition`	`cond`
String Expressions	
`log`	`expr`
`send`	`eventexpr`
`send`	`targetexpr`
`send`	`typeexpr`
`send`	`delayexpr`
`invoke`	`typeexpr`
`invoke`	`srcexpr`
Numeric Expressions	
`foreach`	`index`
Statements	
`script`	*text child nodes*
`finalize`	*text child nodes*

Element	**Attribute**
Arbitrary Data Structures	
`foreach`	`item`
`data`	`expr`
`data`	*text child nodes*
`assign`	`expr`
`assign`	*text child nodes*
`param`	`expr`
`content`	`expr`
`content`	*text child nodes*
Array of Arbitrary Data Structures	
`foreach`	`array`
Variable Identifiers	
`send`	`idlocation`
`data`	`id`
`cancel`	`sendidexpr`
`assign`	`location`
`param`	`location`
`invoke`	`idlocation`
Array of Variable Identifiers	
`send`	`namelist`
`invoke`	`namelist`

Table 6.2.: Data-model dependent elements and attributes in SCXML sorted by type.

In order to provide a PROMELA data-model via SCXML, an interpreter will need to syntactically analyze and semantically evaluate respective language fragments. Both steps are already implemented in the SPIN model-checker as it will accept PROMELA programs as input files. A closer inspection of the implementation in SPIN revealed, though, that both steps are too deeply engrained in its inner workings and the extensive use of global variables makes its implementation unsuitable to be used in an SCXML interpreter where multiple instances might require strict compartmentalization. Nevertheless, we were able to extract the PROMELA grammar given as a `yacc` input file in the SPIN distribution and generated a tokenizer via `flex`.

We restricted the grammar in the PROMELA data-model for SCXML to *expression*s, *declaration*s and a subset of *statement*s of the original grammar. All of which are parsed into an abstract syntax tree and evaluated when interpreting the state-chart. We, specifically, excluded those language features for which a corresponding SCXML feature exists (e.g. `if`, `for` or message channels). As a consequence, this allows us to ignore the issues of branching out in verification mode and randomly selecting an executable statement in simulation mode when evaluating a state-chart with a PROMELA data-model.

6.3.1 PROMELA Expressions

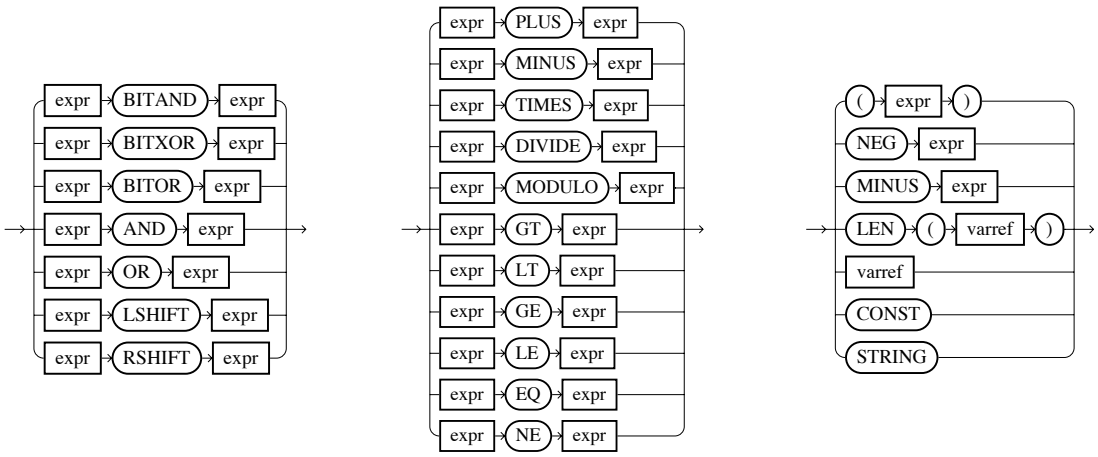


Figure 6.6.: Split syntax diagram of the non-terminal `expr` for expressions available in the PROMELA data-model. Note the additional rule to resolve `expr` as string literals (right column, down-most).

In the original PROMELA grammar, there are three classes of expressions: (i) those referring only to constant values (non-terminal `const_expr`), with a scalar value that can be calculated already during syntactic analysis (e.g. for an array or channel declaration's length), (ii) expressions used to probe message channels (e.g. `full`, `empty`) that need to be non-negatable due to technical reasons with SPIN's partial order reduction[3] during verification and (iii) the usual expressions with constants and variables mixed via various operators.

The grammar for expressions in the PROMELA data-model is given in figure 6.6 as a syntax diagram. We simplified the three classes into one non-terminal `expr`. Whenever a non-constant expression is used, where a constant expression is required, we will simply raise an error during semantic analysis. Non-negatable expressions can be ignored completely as we will not support message channels in the data-model.

An important addition to the original expression grammar is the introduction of string literals whenever an expression is expected. This raises the issue about how to represent such a construct in the transformed PROMELA program and the set of operations available for string literals. For now, we will just introduce these as valid expressions and discuss them in more detail when presenting the actual transformation to PROMELA programs in the next section.

Expressions are valid in all instances, where a boolean, numeric or string expression or an array thereof is required from the data-model as per table 6.2. Using a boolean in a numeric expression will evaluate to `0` or `1` depending on the boolean's value and every numeric value but `0` is `true`. All string literals but the empty string are `true` and using these as a numeric expression is a semantical error. The string representation of numerals and booleans is their actual number in base 10 or the literals `true` and `false` respectively. Compound data structures are possible via user-supplied type definitions (`typedef`). Table 6.3 shows a matrix of conversion rules for expressions of different types.

Expected \ Provided	`boolean`	`numeric`	`string`	`structure`
`boolean`	`N/A`	`false` $if = 0$ `true` $else$	`false` $if = \epsilon$ `true` $else$	`false` $if = \emptyset$ or atom false `true` $else$
`numeric`	`1` or `0`	`N/A`	semantic error	numeric atom or semantic error
`string`	`true` or `false`	numerals	`N/A`	string atom or semantic error
`structure`	boolean atom	numeric atom	string atom	`N/A`

Table 6.3.: Type conversions for expressions in the PROMELA data-model.

6.3.2 PROMELA Declarations

Declarations in PROMELA are available to (i) introduce and optionally initialize typed variables, (ii) to define user supplied, compound types and (iii) to introduce enumerations. The grammar available for declarations in the

[3] `http://spinroot.com/spin/Man/empty.html`

PROMELA data-model is given as a syntax diagram in figure 6.7. It is equivalent to the original PROMELA grammar from the SPIN distribution with the exception of not enforcing user-defined type definitions (non-terminal `TYPEDEF`) to be in the global scope. This simplifies the grammar somewhat by reducing the number of non-terminals and makes no difference for the declaration of user-supplied types as we will just copy them verbatim into the global scope in the resulting PROMELA program when transforming.

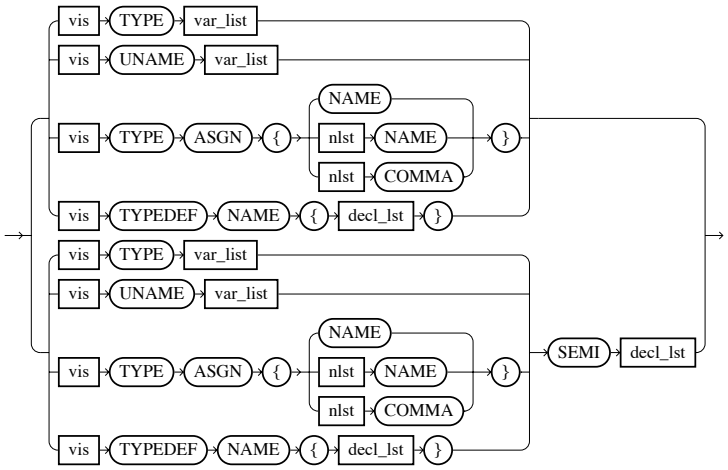


Figure 6.7.: Syntax diagram of non-terminal `decl_lst` for declarations available in the PROMELA data-model.

In order to introduce the new type `string` for variables, we had to adapt the grammar to allow the literal "string" to be a valid terminal token for the non-terminal `TYPE`. As such, the following is a valid declaration for the grammar of the SCXML PROMELA data-model, but not in the grammar as employed by the PROMELA implementation in the SPIN model-checker:

```
string foo = 'This is a string!'
```

We will see later how we can, nevertheless, transform this expression into something in the original PROMELA grammar and have the SPIN interpreter process it for formal verification. The declaration grammar is only available in SCXML `<data>` elements. The following listings illustrate an SCXML snippet on the left and the PROMELA pendant in the right column. There are three variations.

1. Write complete declarations with the grammar given above into a text child node of a `<data>` element without any additional attributes. This is the most flexible variation as the complete declaration grammar is available:

```
1 <data>int Var1 = 0;</data>
2 <data>mtype = { ack, nak, err };</data>
3 <data>mtype Var2 = ack;</data>
```

```
1 int Var1 = 0;
2 mtype = { ack, nak, err };
3 mtype Var2 = ack;
```

2. Separate a variables' declaration into the `<data>`'s attributes `id`, `expr` and `type`. The latter is an optional extension and defaults to `int`. This variation is more in line with the original syntax and semantics of the `<data>` element as per SCXML specification:

```
1 <data id="Var1" type="short" expr="24" />
2 <data id="Var2" type="mtype" expr="nak" />
3 <data id="Var3" type="int[3]" />
```

```
1 short Var1 = 24;
2 mtype Var2 = nak;
3 int Var3[3];
```

3. Declare a variable's name via the `id` attribute of a `<data>` element and supply a JSON structure in the `expr` attribute or as a text child node. This is the most convenient variation as the data-model will automatically declare and initialize all relevant fields, even for compound types. The data-model makes no effort, though, to reduce the bit-width of any of the numeric fields and just declares them as type `int`. Furthermore, recursive declarations are not supported as they are inexpressible in the PROMELA language:

```
1 <data id="Var1" expr="[1,2,3] />
```

```
1 int Var1[3];
2 Var1[0] = 1;
3 Var1[0] = 2;
4 Var1[0] = 3;
```

```
<data id="Var2">
  { foo: 12, bar: 4 }
</data>
```

```
typedef Var2_t {
  int foo;
  int bar;
};
Var2_t Var2;
Var2.foo = 12;
Var2.bar = 4;
```

When working with data from external systems in SCXML, the data formats of JSON and XML play a prominent role underlining the necessity and usefulness of the last variation. However, while it is possible to model most JSON expressions in PROMELA, support for a representation of data via the XML DOM is plainly unpractical to implement.

6.3.3 PROMELA Statements

Statements in the PROMELA language are available to modify the program's state, direct control flow and generally perform actions. As per table 6.2, the statement grammar is only available in text child nodes of `<script>` and `<finalize>` elements in SCXML documents employing the data-model.

The original PROMELA grammar for statements is given in figure 6.8. It features most of the syntactical constructs one would expect in a programming language with the omission of any string operations. It is noteworthy, that the original grammar allows for function calls via the `INAME` → `'('` → `args` → `')'` → `Stmnt` rule (left column, third rule from bottom), but the feature is undocumented. An inquiry with the original author confirmed that usage of the feature is experimental and discouraged at the present time.

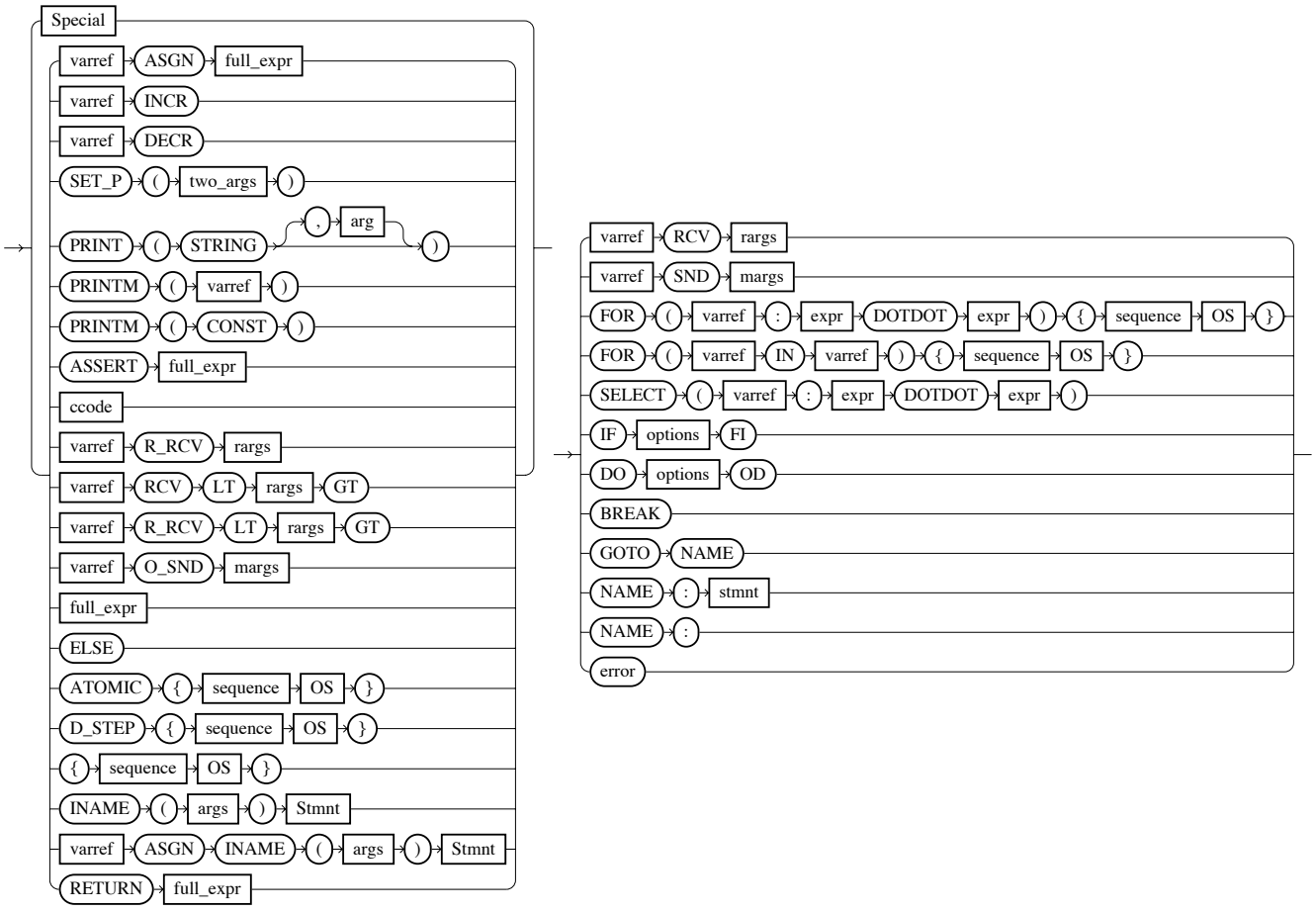


Figure 6.8.: Syntax diagram of the non-terminal `stmnt` (left) and `Special` (right) for statements in the original PROMELA grammar.

In the presence of SCXML executable content (see section 4.1.3), most of the original PROMELA language constructs are either redundant or have non-obvious (even misleading) semantics when provided as part of a SCXML data-model. Trimming functionality to a convenient subset with no correspondence in SCXML language features clarifies semantics and simplifies the required analysis in the data-model's implementation, thus reducing a source

for errors. Furthermore, the semantics of some statements differ with the mode of interpretation employed by SPIN: In simulation mode, the interpreter will randomly choose one of the executable statements (e.g. conditions in an `if` block), whereas in verification mode, all of them are chosen and control flow branches. The following list discusses the statement rules dropped from the grammar available via the PROMELA data-model and provides a rationale as to why they were excluded:

Setting the process' priority:

```
SET_P → '(' → two_args → ')'
```

In the original PROMELA language, it is possible to set the priority of a process to some positive, integer value. The semantics of the statement depend on the mode of interpretation of the PROMELA program. In simulation mode, only statements from processes with the highest priority are chosen until no more such statements are executable, only then are statements from processes with a lower priority considered. In verification mode, no branching for executable statements for lower priority processes is created, thus no interleaving of their statements is considered until all higher priority processes exhausted their executable statements. These semantics are foreign to an SCXML interpreter and as such unavailable via the grammar from the PROMELA data-model.

Print a line on STDOUT:

```
PRINT → '(' → STRING → prargs → ')'
PRINTM → '(' → varref → ')'
PRINTM → '(' → CONST → ')'
```

PROMELA allows to print string literals with embedded conversion specifiers, variables and constant expressions to the standard output, just like `printf` from the C language. Similar functionality is already available in SCXML documents via the `<log>` element in executable content and, consequentially, dropped from the grammar in the PROMELA data-model.

Embedded C-Code:

```
ccode
```

PROMELA allows an author to embed arbitrary C code to be inserted verbatim into the resulting binary for an exhaustive search. The `c_code` statement is used in conjunction with the `c_expr` expression and `c_state` declaration to extend PROMELA models via the C language. The grammar of the PROMELA data-model does not support this feature.

Sending and receiving on channels:

```
varref → RCV → rargs
varref → RCV → '<' → rargs → '>'
varref → R_RCV → rargs
varref → R_RCV → '<' → rargs → '>'
varref → SND → margs
varref → O_SND → margs
```

One of the major language features to synchronize concurrent processes in PROMELA is to enqueue events in message channels. Various variations and idioms are available, from blocking or unreliable queues to rendezvous channels. Message channels can also be sorted and tested for the existence of events with a specific set of values. Sending events to an interpreter's queues is central to SCXML as well, but there it is available via the `<send>` element. Introducing additional message channels to communicate between e.g. two state-charts dilutes the semantics of the `<send>` element as it offers functionality very similar to already existing semantics. As such, we drop statements for sending and receiving events from channels, just as we dropped channel declarations above.

Expressions as statements:

```
full_expr
```

In PROMELA, an expression can be used as a statement to cause the process to block until the expression evaluates to something other than `0`. When using global variables in such an expression, it can also be used to synchronize with other processes. The semantics of *block until another process proceeds beyond a certain point* is already expressible in SCXML by sending events to another state-chart. Within the scope of SCXML, an author would not expect an expression such as `(3 - Var1)` to pause execution until `Var1` has a value other than `3`. Therefore, we drop expressions as statements from the grammar available in the PROMELA data-model.

Conditional execution:

```
IF → options → FI
ELSE
```

The semantics of `if` blocks in PROMELA is somewhat unusual: the options specify a set of conditions, each followed by a list of statements to be executed when the condition is true. However, not necessarily the list of statements with the first matching condition is evaluated. In simulation mode, one matching condition is picked at random, whereas in verification mode, the interpreter branches out for every matching condition. To implement the usual semantics where only the statements of the first matching conditional are evaluated, one has to nest further conditions in an additional `else` block. Given the different semantics and the fact that conditional execution and control flow branching is already available in SCXML via the usual `<if>` / `<elseif>` / `<else>` blocks, we dropped these statements from the grammar.

Atomic and discrete blocks of statements:

```
ATOMIC → '{' → sequence → OS → '}'
D_STEP → '{' → sequence → OS → '}'
'{' → sequence → OS → '}'
```

Blocks of statements in PROMELA can be marked as atomic or deterministic, this will cause the SPIN interpreter not interleave these statements with statements from concurrent processes. Semantics differ somewhat between the two type of blocks in such that atomic blocks may contain blocking statements, in which case atomicity is lost and interleaving with statements from other processes may occur. Whereas encountering a blocking statement in a block of statements marked via `d_step` is a semantic error. This language feature is not available as all executable content in SCXML will implicitly assume to run in an atomic block as is implied by the execution semantics of SCXML where interleaving can only occur in between events.

Function calls / Inline functions:

```
INAME → '(' → args → ')' → Stmnt
varref → ASGN → INAME → '(' → args → ')' → Stmnt
RETURN → full_expr
```

As briefly mentioned above, the original PROMELA grammar specifies syntactical constructs for function calls but its usage is discouraged as experimental. An alternative to actual function calls in PROMELA is given as *inline functions*: every invocation is substituted verbatim by the complete function body. These can be thought of as simple macros and do not have the same semantics as function calls with regard to recursion or scope. Neither approach is exposed via the PROMELA data-model.

Iterating arrays:

```
FOR → '(' → varref → ':'  → expr → '..'  → expr → ')' → '{' → sequence → OS → '}'
FOR → '(' → varref → IN → varref → ')' → '{' → sequence → OS → '}'
```

Iterating arrays is already available in SCXML via the `<foreach>` element with the exact same semantics as in PROMELA. Thus, it is dropped from the grammar available in the PROMELA data-model.

Non-deterministic value selection:

```
SELECT → '(' → varref → ':'  → expr → '..'  → expr → ')'
```

The `select` statement is a convenience construct to select a random integer value from a given range. The indeterminism is resolved in verification mode by branching out for every possible value. While it might ultimately be useful to have a source of indeterminism within the PROMELA data-model, we did not implement this class of statements either.

Looping until a condition is met:

```
DO → options → OD
BREAK
```

Looping a sequence of statements until a condition is met might actually be useful when available via the PROMELA data-model. However, the same effect can be achieved by using eventless transitions, reentering a (nested) state. In order to keep the language features of SCXML with the PROMELA data-model orthogonal, this class of statements is not available either.

Labels and goto:

```
GOTO → NAME
NAME → ':'  → stmnt
```

`NAME` → `':'`

Providing labels for any sequence of statements and allowing to redirect control flow respectively via `goto` is in direct conflict to the concept of states and transitions. It is not clear how these semantics can be applied to state-charts, let alone in a way that is not completely confusing to an author of SCXML documents. There are, however, three special instances where labels have semantics other than as a target for `goto`: (i) Labels starting with the string literal `end` identify sequences of statements for which a process blocking there on program termination is a valid end state. (ii) Label names prefixed by `accept` can be used in embedded linear temporal logic expressions to identify sequences of statements for which such an expression is considered accepted. (iii) A prefix of `progress` signifies a set of statements required to ascertain the *liveliness* of a system . These classes of labels are available via special comments in the SCXML document but have no semantics when the state-chart is interpreted as a SCXML file. When transforming the state-chart onto a PROMELA program later, these comments will cause the generation of respective labels.

The remaining grammar for statements actually available in the PROMELA data-model is given in figure 6.9. In essence, only assignments, post- and prefix incrementation as well as assertions remain available. It can be argued that assignments are also already expressible via the SCXML `<assign>` element and retaining them in the grammar for statements in the PROMELA data-model is more convenience than necessity. All other PROMELA language constructs for statements either have a corresponding SCMXL language feature with familiar semantics or are inapplicable. Maybe the large overlap between PROMELA statements and SCXML language features should not come as a surprise as SCXML was specified as a control-flow language and most missing PROMELA statements are specific to model-checking.

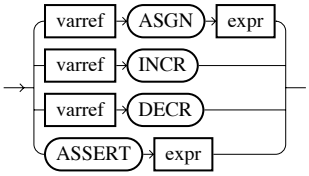


Figure 6.9.: Syntax diagram of non-terminal `stmnt` for statements available in the PROMELA data-model.

Ultimately, we do provide all of PROMELA's grammar via special XML comments in executable content, where arbitrary PROMELA statements can be inserted. These comments have no semantic when encountered as part of SCXML interpretation and will be inserted verbatim when the generating the PROMELA programs.

6.3.4 Evaluating the PROMELA Data-Model

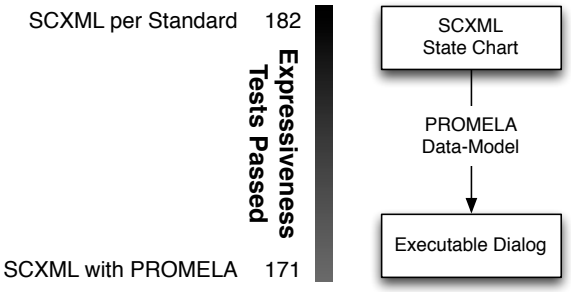


Figure 6.10.: Only a subset of the original SCXML tests pass with the PROMELA data-model.

The SCXML implementation report plan defines 233 tests (see table 5.1) for all functional requirements defined as part of the SCXML specification. Of those, 182 are agnostic of the employed data-model and can be transformed per XSLT for a specific data-model. XSLT transformation descriptions for the ECMAScript and XPath are provided as part of the SCXML Implementation Report Plan (IRP) and were adapted to generate tests specific to the PROMELA data-model. 171 of these 182 tests are passed by our PROMELA data-model using the approaches described above (figure 6.10). The failed tests are detailed in table 6.4 with the reason for their failure explained in table 6.5.

Class	Section	#Tests
Core Constructs		**40 / 40**
Executable Content		**12 / 13**
Foreach	4.6	6 / 7
`test525`: *Iterate on shallow copy of array*		
Data-Model and Data Manipulation		**49 / 50**
Data	5.3	6 / 7
`test280`: *Declare vs. assign with late binding*		
External Communications		**49 / 51**
Invoke	6.4	27 / 29
`test224`: *Generate idlocation as 'stateid.platformid'* `test530`: *Content 'src' is evaluated at runtime*		
Data-Models		**0 / 51**
Specific to XPath, ECMAScript, NULL → inapplicable		
Event I/O Processors		**22 / 28**
SCXML Event I/O Processor	C.1	14 / 16
`test190`: *'#_scxml_sessionid' targets external queue* `test350`: *Target '#_scxml_sessionid' supported*		
Basic HTTP Event I/O Processor	C.2	8 / 13
`test509`: *Accept HTTP POST requests* `test518`: *Namelist entries as POST parameters* `test519`: *Param elements as POST parameters* `test520`: *Content element becomes request body* `test534`: *Event for send becomes '_scxmleventname'*		
Total		**171 / 182**

Table 6.4.: Number of W3C SCXML IRP tests failed with the PROMELA data-model.

Cause	#	Tests
String operations required	8	`190`, `224`, `350`, `509`, `518`, `519`, `520`, `534`
Declared vs. defined	1	`280`
Dynamic arrays	1	`525`
XML DOM node in variable	1	`530`

Table 6.5.: Reasons for failing W3C SCXML IRP tests with the PROMELA data-model.

It is important to recall that the PROMELA data-model is provided to ultimately enable model-checking of non-trivial SCXML state-charts by transforming the complete document onto a PROMELA program as an input file for the SPIN model-checker. Later, in section 6.4, we will see that only a subset of the 171 tests passed by the PROMELA data-model are actually accessible to model-checking for various reasons (mainly due to relying on errors raised by the platform). Therefore, even if a test is passed by an interpreter with the PROMELA data-model, it might still not be possible to apply the formalisms of model-checking on a respective state-chart document. It can be argued that only the formally verifiable subset should be passed with the PROMELA data-model, or in other words: a state-chart employing the PROMELA data-model should *always* allow formal verification or already fail when processed by an SCMXL interpreter. However, even if formal verification for a state-chart is unattainable in its given form (due to features missing when interpreted by SPIN), having a skeleton PROMELA program and a list of issues still allows to eventually remodel the problematic sections in the PROMELA program and abstract from the problematic parts.

We will conclude the discussion of the PROMELA data-model by detailing the reasons why some of the tests fail. All of these failed tests will occur again when the set of tests failing formal verification is discussed in section 6.4.13.

String operations:

We extended the PROMELA grammar to recognize the basic type `string`, but did not define any operations other than literal identity of two strings. While it would be no problem to introduce additional operators / relations for more elaborate string operations (e.g. concatenation, finding substrings) into the grammar and provide semantic support via the data-model, we will see later that such an approach would be very difficult to model in the PROMELA language when the actual model-checking is performed.

Declared vs. defined:

In `test280`, late data binding is used to declare a variable when a nested state is entered. Prior to entering

the state, a condition in a transition checks, whether the variable is already defined and the test fails if this is the case (i.e. `typeof Var2 === 'undefined'` in ECMAScript). There is no concept of undefined variables in PROMELA, accessing a variable that is not defined is simply a syntax error. One could rely on the default boolean evaluation of defined but uninitialized integer values always being `false`, but this is not what is intended with late data binding. Another possibility is to augment every such variable with a companion variable of boolean type, encoding whether the variable is question was already defined. But given the apparent marginality of the late-binding language feature, the effort seems to be in no proportion to the gains.

Dynamic arrays:
When processing a `<foreach>` element in test 525, the interpreter is supposed to make a shallow copy of the array to iterate, in order to be agnostic of changes to the very same array within an iteration. While this is expressible in PROMELA, the respective test will dynamically append one element to the array, which is not. As with the defined vs. declared problem, it would be possible to define dynamic arrays by introducing a companion variable pointing to the last defined index in the array and define the array as large as is ultimately needed.

XML DOM node in variable:
In `test530`, the `<assign>` element is used to bind an XML node to a variable, which is later passed to an invoked state-chart interpreter. This implies a representation of the XML DOM, or at least a simplified variant to be available in the data-model and ultimately the generated PROMELA source code. While this might be possible, the amount of work, both in terms of implementation effort and increased complexity of the generated PROMELA program would be immense.

6.4 Model-Checking for State-Chart XML Documents

With the transformation of SCXML state-charts onto equivalent state-machines and the PROMELA data-model introduced earlier in this chapter, this section will finally describe the (semi-)automatic transformation of SCXML documents onto actual PROMELA programs to enable verification via linear temporal logic expressions. The transformation allows to subject dialog models expressed in SCXML documents employing the PROMELA or NULL data-model to the formalisms of model checking (see figure 6.11). Transformations for both of these data-models is now rather trivial. In the case of the PROMELA data-model all expressions, declarations and statements are already given in the target language or can easily be transformed (e.g. JSON in assignments). For the NULL data-model, only the `In('state')` predicate is required and no actual embedded scripting functionality is exposed. However, this limitation is not inherent to the overall approach and transformations from other formal languages onto PROMELA or Kripke structures directly do exist (e.g. from a subset of ANSI C [Hol00]).

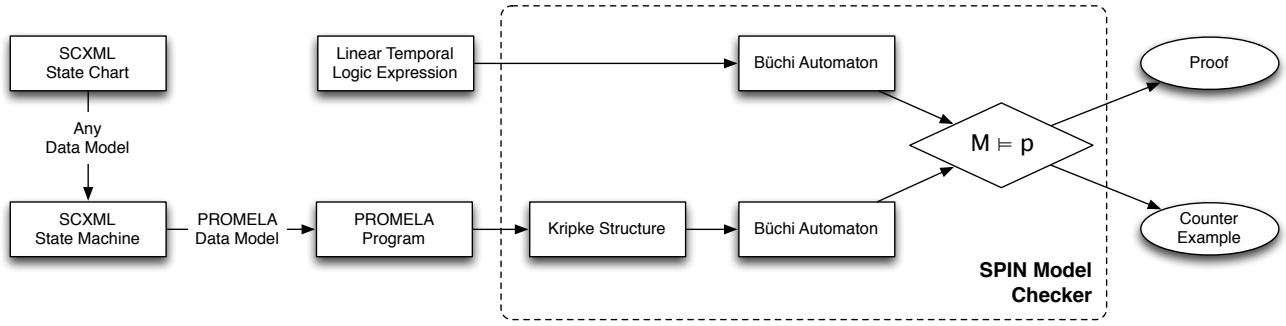


Figure 6.11.: Model checking for SCXML documents.

In order to make SCXML documents accessible for model checking, we will have to express an SCXML interpreter in PROMELA. As section 5.3 already introduced a general transformation from SCXML state-charts onto equivalent SCXML state-machines, we can ignore many of the more complicated issues of transition selection, evaluation order and configuration management as we will assume, without loss of generality, that every SCXML document is given in its state-machine representation. This trivializes the transformation and eases implementation considerably. While this is technically correct, we will see later with the transformation of the NVidia Shield's SCXML document, that this puristic approach has some issues (e.g. with state explosions and the size of intermediate documents) when applied in the real world and some additional adaptations are proposed and evaluated.

Using the W3C tests of SCXML, XSLT-transformed for the PROMELA data-model, we can evaluate the expressiveness of the approach as the set of tests for which we can formally verify that they will pass and get a clear understanding of its limitations. We will conclude this section with an overview of yet missing language features and a discussion on how they could be realized. A complete PROMELA program as the result of transforming an SCXML document with most language features can be found in the appendix in listing A.3.

6.4.1 A State-Machine in PROMELA

Expressing a simple state-machine in PROMELA is straight-forward. We enumerate the set of states as numeric values $s_1, \ldots, s_n$ and introduce an integer variable `_state` to hold the current state. The same is done for the set of input symbols $e_1, \ldots, e_m$ in the integer variable `_event`. Transitions are encoded in an `if` block where the current state and the current input symbol are considered to move to the next state. The input is read from a message queue q.

```
1  /* states */
2  #define s1   0
3  #define s2   1
4  #define s3   2
5
6  /* events */
7  #define e1   0
8  #define e2   1
9  #define e3   2
10
11 int _event;              /* current input */
12 int _state;              /* current state */
13 chan q = [100] of {int}  /* queue with input word */
14
15 proctype step() {
16   nextStep: /* pop next input */
17   q ? _event
18
19   if
20     /* transitions */
21     :: (_state == s1 & _event == e1) -> _state = s2; goto nextStep;
22     :: (_state == s2 & _event == e2) -> _state = s3; goto nextStep;
23     :: (_state == s3 & _event == e3) -> _state = s4; goto nextStep;
24     :: else -> assert(false);   /* illegal transition */
25   fi;
26   done:
27 }
28
29 init() {
30   q!e1; q!e2; q!e3; /* fill queue with input word */
31   _state = s1;      /* start state */
32   run step();
33 }
```

Listing 6.1: Simple state-machine in PROMELA.

Using the template state machine from listing 6.1, we will gradually refine the approach to model all the important semantics of an SCXML state-machine. If we ignore executable content for global transitions in $\mathcal{X}(\widetilde{t})$ from the transformation to SCXML state-machines above (see equation 5.27) and the semantics related to the internal and external queue, we can already see how this approach can be employed to express the transformed state-machines: `_state` will contain the global state and the `if` block in line 19-25 will dispatch global transitions depending on the current state and the global transition's event and condition.

6.4.2 Transition Selection

In a SCXML state-chart, transitions are either driven by events or spontaneous when no event descriptor is associated with a transition. The SCXML specification mandates two event queues: (i) an internal queue where events raised by the platform during processing, such as `error.platform`, `error.communication` or any event specified via the `<raise>` element are enqueued and (ii) an external event queue where events from invoked components, external systems and those specified via the `<send>` element are enqueued. We already introduced the principal scheme of processing events and selecting transitions in figure 4.11 earlier in this thesis. Whenever a configuration is assumed, the interpreter will

1. Check for transitions enabled by the empty event. These are spontaneous (or eventless) transitions, only conditionalized by an eventual boolean guard condition. They will not consume an event but are equivalent to normal transitions in any other way, i.e. they can cause the evaluation of executable content and trigger

state changes. Processing an optimal transition set containing only spontaneous transitions is referred to as a *micro-step* by the SCXML standard.

2. When there are no more spontaneous transitions enabled, an interpreter will consume an event from the internal event queue. These events may have been enqueued either by an explicit `<raise>` element or by the platform itself, e.g. when errors were detected. The set of spontaneous transitions leading to the dequeueing and processing of any event is referred to as a *macro-step*.

3. If there are no internal events enqueued, the interpreter will finally try to dequeue an external event. If there are no external events, the interpreter will pause execution until an event arrives either from any external system or a delayed event, sent to the session itself when executable content was processed.

At startup, eventual spontaneous transitions are exhausted until a stable configuration is assumed.

Revisiting the simple PROMELA state-machine from listing 6.1, we can see that there is no notion of spontaneous events, nor is there a distinction between the internal or external queue. The adapted PROMELA state-machine in listing 6.2 will model this behavior.

```
1  int _state = s1;           /* current state */
2  int _event;                /* current event */
3  bool spontaneous = true;   /* exhaust spontaneous transitions */
4
5  chan iQ = [100] of {int};  /* internal queue */
6  chan eQ = [100] of {int};  /* external queue */
7  ...
8  goto microStep;
9
10 t1:
11   iQ!FOO;                                  } Executable content associated with global transition t1,
12   s = s1;                                  } pushing event FOO into the internal queue and updating
13   goto microStep;                          } the global state.
14
15   /* pop an event */
16 macroStep:
17   if                                       }
18   :: len(iQ) != 0 -> iQ ? _event           } Pop an event preferring the internal queue or block until
19   :: else -> eQ ? _event                   } an external event becomes available.
20   fi;                                      }
21
22 microStep:
23   /* state and transition dispatching */
24   if
25   :: (_state == s1) -> {
26     if                                     }
27     :: (!spontaneous && _event == FOO) -> { } An eventful / non-spontaneous, inline transition without
28       _state = s2;                         } executable content updates the current state and causes
29       spontaneous = true;                  } processing of subsequent spontaneous transitions.
30       goto microStep;                      }
31     }
32     :: else -> {
33       if                                   }
34       :: (spontaneous && Var4==0) -> {     } An eventless / spontaneous transition conditionalized on
35         goto t1;                           } Var4==0 with executable content at label t1.
36       }                                    }
37       :: else {                            }
38         spontaneous = false;               } No transitions applicable in this state, trigger a macro-
39         goto macroStep;                    } step to dequeue an event.
40       }                                    }
41       fi
42     }
43     fi;
44   }
45   ...
46   fi;
```

Listing 6.2: SCXML transition selection in PROMELA.

Again, the set of states in the adapted PROMELA state-machine corresponds to the global states $\widetilde{\mathcal{S}}$ from the transformation onto SCXML state-machines (see section 5.3) and the set and order of transitions per state reflects the sorted, global transitions $\widetilde{\mathcal{T}}(i)$ introduced with the transformation. This enables us to ignore determining the optimal transition set at runtime as we did show that the global transitions reflect the proper optimal transition sets and that their order corresponds to the original selection criteria as defined in the SCXML specification.

We introduced a new internal event queue `iQ` and renamed the queue `q` as the external queue `eQ` for clarity. The boolean variable `spontaneous` causes only eventless transitions to be considered when dispatching transitions in a state and is set to `false` when no more spontaneous transitions are applicable (line 38). The transitions themselves

are given in nested `if/else` blocks (line 26-43) per state, each guarded by (i) the boolean variable `spontaneous`, to distinguish eventful and eventless transitions, (ii) their event descriptor $\mathcal{E}(\widetilde{t})$ in the case of eventful transitions and (iii) their eventual `cond` attribute $\mathcal{C}(\widetilde{t})$.

If a transition does not specify executable content ($\mathcal{X}(\widetilde{t}) = \emptyset$), it is simply inlined with assigning the destination state and retriggering transition selection (line 27-31). In the case of an eventful / non-spontaneous transition, processing will reset the boolean `spontaneous` to *true*, thereby triggering all entailing spontaneous transitions as micro-steps. If a transition did specify executable content, it is provided at a label unique to the transition (line 10-13) to keep transition dispatching per state clear and concise. When no more spontaneous transitions are applicable, the last block in transition dispatching per state will trigger a macro-step to read an event.

At startup, `spontaneous` is true and control flow is directed to transition selection without dequeueing an event (line 8), causing only spontaneous transitions to be considered. If we assume that `Var4==0` is true, control flow will be directed to `t1` (line 10-13), where the event `FOO` is enqueued on the internal queue and the next micro-step is attempted. When reentering the transition dispatching block, none of the conditions are eligible as `spontaneous` is still set and the default, to trigger a macro-step, is executed. This causes the interpreter to dequeue the event `FOO` from the internal queue and reenter transition dispatching. Now, the eventful transition at line 22 is selected and the state updated to `s2` (not shown) where processing continues.

6.4.3 Event Descriptor Matching

In the example above, it is implied that an event descriptor associated with a non-spontaneous transition will only match a single event name (here `FOO`). In fact, every event name that contains the event descriptor's dot separated tokens as a prefix will match the descriptor (see definition 9). E.g. the event descriptor `button` will not only match this literal event name, but also any event name starting with `button.`, e.g. `button.dpad.up`. In our approach of modeling SCXML state-machines in PROMELA, we simply enumerated all events and will have to resolve event descriptors for non-spontaneous transitions onto actual event names. It is assumed, that the set of individual event names is known a-priori by examining the SCXML document. This is not always possible as arbitrary events may arrive from external systems - their names will have to be mapped onto their longest prefix that is actually handled in the SCXML document (something we will have to do when *closing the system* below). We will define $\mathcal{E}$ as the set of all event names and event descriptors as:

$$\begin{aligned} \mathcal{E} \quad = \quad & \{e \mid e \in t.event \wedge t \text{ is a transition element}\} \ \cup \\ & \{r.event \mid r \text{ is a raise element}\} \ \cup \\ & \{s.event \mid s \text{ is a send element}\} \ \cup \\ & \{c \in \text{string constants} \mid \exists\ \texttt{OP}_{\texttt{EQ}}(\texttt{\_event.name}, c) \in \text{all expressions}\} \end{aligned} \tag{6.2}$$

The last subset will add any string constants (see syntactical rule in figure 6.6, rightmost column, bottom) for which a direct comparison with `_event.name` is found in any expression in the document, e.g. in the `cond` attribute of an `<if>` element (e.g. `_event.name == 'button.shoulder'`).

In order to resolve a transition's event descriptor onto the set of matching event names, we can employ a prefix trie of all $e \in \mathcal{E}$ and resolve all the transition's event descriptors with the descriptor itself and its child nodes (the optional trailing `.*` in event descriptors is removed). For example, let us assume $\mathcal{E}$ to be:

```
{ button, button.dpad, button.dpad.up, button.dpad.left, button.dpad.down, button.shoulder,
  error, error.foo, error.bar,
  baz }
```

This set of event names and descriptors will result in the prefix tree depicted in figure 6.12. Now, when a transition in the original SCXML state-machine was triggered by e.g. `button.dpad`, the actual event name and all the respective node's descendants in the prefix trie (`button.dpad, button.dpad.up, button.dpad.left, button.dpad.down`) will trigger the transition. This, in essence, removes the need for event name matching via event descriptors at runtime, but requires all event names to be known a-priori. Note that there is no `eventexpr` attribute defined for `<transition>` elements in SCXML, only an `event` attribute for literal event descriptors, i.e. the set of event names matching a transition can not dynamically change via a data-model expression.

Listing 6.3 illustrates the approach. The transition is enabled by all events matching the given event descriptor `button.dpad`. Finding the event descriptor in the prefix trie from figure 6.12 will identify four actual event names that this descriptor matches. When transformed onto a PROMELA program (listing 6.4), the transition will be enabled by all the events matching the event descriptor.

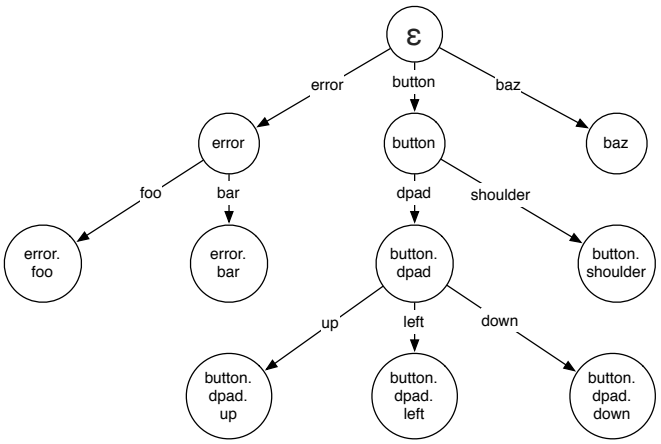


Figure 6.12.: Prefix trie of event names.

```
<state id="s01">
  <transition event="button.dpad" target="s02"/>
</state>
```

Listing 6.3: SCXML snippet illustrating event descriptor matching.

```
/* event name identifiers */
...
#define BUTTON_DPAD 6       /* from "button.dpad" */
#define BUTTON_DPAD_UP 7    /* from "button.dpad.up" */
#define BUTTON_DPAD_LEFT 8  /* from "button.dpad.left" */
#define BUTTON_DPAD_DOWN 9  /* from "button.dpad.down" */
...
/* ### current state s01 ######################### */
:: (s == s1) -> {
  if
  :: (!spontaneous &&
      _event == BUTTON_DPAD ||
      _event == BUTTON_DPAD_UP ||
      _event == BUTTON_DPAD_LEFT ||
      _event == BUTTON_DPAD_DOWN) -> {
    goto t2;
  }
...
```

Four different input events will match the `button.dpad` event descriptor.

Listing 6.4: Resolving event descriptors in PROMELA onto actual event names.

6.4.4 Strings in PROMELA

There is no language support for strings in PROMELA and, as such, no operations defined on them. This is a real limitation as most real-world application at least need to employ string literals when communicating with external entities. This is no problem when interpreting PROMELA language snippets as part of the data-model, as we introduced a new variable type `string` for which the identity operation is defined. To model its semantics when interpreted by the SPIN model checker, we will simply enumerate all string literals and replace them by integer values. With this approach, we can abuse the identity operator for integers and mimic string literals. To this effect, we first parse all data-model specific expressions, declarations and statements into abstract syntax trees and recursively search for string literals. Subsequently, we will `#define` a macro named after a canonicalized variant of the literals content and assign a unique, consecutive, positive integer to each (listing 6.5).

```
/* string literals */
#define _SESSIONID 1     /* _sessionid */
#define _NAME 2          /* _name */
#define FAIL 3           /* fail */
#define FOO 4            /* foo */
#define PASS 5           /* pass */
#define HTTP_WWW_W3_ORG_TR_SCXML_SCXMLEVENTPROCESSOR 6 /* http://www.w3.org/TR/scxml/#SCXMLEventProcessor */
#define EXITING_SUB0 7   /* Exiting sub0 */
#define EXITING_SUB01 8  /* Exiting sub01 */
```

Listing 6.5: Enumerated string literals in PROMELA.

As a final step, we replace all string literals in all PROMELA language fragments by their macro name. This causes the SPIN interpreter to *see* numeric values in place of the strings. One of the undesired consequences of this approach is, that the comparison of a string literal with an integer value might surprisingly be `true` if the integer, by chance, has the value of the string literal's enumerated integer. This is easily recognized when we process the abstract syntax trees, though, and can be raised as a semantical error during transformation.

6.4.5 Complex Events

In most real-world applications, events passed into the interpreter or sent from it carry additional data. While we can always resolve events with the same name but different data by introducing a new event name, this limitation is awkward in practice and would lead to unreadable event names.

The PROMELA language does allow to introduce user-defined types and our transformation from SCXML state-machines to PROMELA programs will employ this feature to model events not as enumerated integer values, but as complex data-structures. During transformation, we perform a syntactical analysis of all PROMELA language fragments and determine, whether there is an access to a member field in the `_event` variable. If this is the case, we will automatically derive the proper type definition for the `_event` variable:

```
<state id="s0">
  <onentry>
    <send event="event1">
      <param name="aParam" expr="2"/>
    </send>
  </onentry>
  <transition event="event1" target="s1">
    <assign location="Var2" expr="_event.data.aParam"/>
  </transition>
</state>
```

```
/* typedefs */
typedef _event_data_t {
  int aParam;
};

typedef _event_t {
  int name;
  _event_data_t data;
};
```

Listing 6.6: Event member access causes complex event structure in PROMELA program.

This approach also allows us to add data to events sent by the interpreter via the `<param>` or `<content>` child in `<send>` elements or its `namelist` attribute:

```
...
  <state id="s0">
    <onentry>
      <send event="event1" namelist="Var2">
        <param name="aParam" expr="Var1"/>
      </send>
    </onentry>
  </state>
...
```

```
...
/* typedefs */
typedef _event_data_t {
  int aParam;
  int Var2;
};

typedef _event_t {
  int name;
  _event_data_t data;
};
...
t1:
  _event_t tmpE;
  tmpE.name = EVENT1;
  tmpE.data.aParam = Var1;
  tmpE.data.Var2 = Var2;
  eQ!tmpE;
...
```

Listing 6.7: Data attached to an event via `namelist` or `<param>` in PROMELA.

6.4.6 Executable Content

In SCXML, executable content is available to (i) send events to external entities (e.g. other SCXML sessions or HTTP servers), (ii) modify the state of the embedded data-model, (iii) direct control flow within a block of executable content and (iv) cause the interpretation of custom, user-supplied XML elements.

To this effect, there are a few XML elements specified by the SCXML standard and undefined extension points for users to register custom executable content elements (compare section 4.1.3). In the following, we will present each XML element available as executable content and its representation in a PROMELA program, as well as eventual adaptations required to the PROMELA state-machine template from listing 6.2.

raise

The `<raise>` element will enqueue a simple event on the internal queue. Its only specified attribute is the event's name in `event` and it will not allow any child nodes. This is straight forward to express with the state machine template and we have already seen an implementation in PROMELA in listing 6.2 (line 11).

```
<raise event="foo" />
```

```
iQ!FOO; /* Push FOO to internal queue */
```

Listing 6.8: SCXML `<raise>` modeled in PROMELA.

if / elseif / else

These elements are available to direct control flow within a block of executable content. With the exception of the fallback `else`, they require a `cond` attribute containing a data-model specific, boolean expression. Only the executable content specified as child elements of the first matching conditional will be processed. To model these elements in PROMELA, we cannot simply employ a single `if / fi` block with conditions ordered respectively, because SPIN will branch-out for every matching condition in verification mode and pick one at random in simulation mode. We have to resolve all `elseif` blocks to nested `else / if` blocks and provide the `else` block as the most deeply nested block.

```
<if cond="expr1">
  <!-- if block -->
<elseif cond="expr2"/>
  <!-- elseif block -->
<else/>
  <!-- else block -->
</if>
```

```
if
:: (expr1) -> {
  /* if block */
}
:: else -> {
  if
  :: (expr2) -> {
    /* elseif block */
  }
  :: else -> {
    /* else block */
  }
  fi;
}
fi;
```

Listing 6.9: SCXML `<if / elseif / else>` modeled in PROMELA.

foreach

The SCXML `<foreach>` element requires an `array` and an `item` attribute and will iterate the executable content contained as its child elements for every item in the given array in the data-model, setting the variable specified in `item` for each iteration to the respective item from the array. An optional attribute `index` can specify another data-model variable to hold the current index of iterations.

As there is native language support for arrays and iteration in PROMELA, we can simply express the `foreach` element as a `for` loop.

```
<foreach index="Var1" item="Var2" array="Var3">
  ...
</foreach>
```

```
for (Var1 in Var3) {
  Var2 = Var3[Var1];
  ...
}
```

Listing 6.10: SCXML `foreach` modeled in PROMELA.

This violates the requirement for a compliant SCXML interpreter to create a shallow copy of the array to iterate. If this functionality is necessary, every array that is iterated via `<foreach>` would have to be declared twice with values copied over to a shadow array which is then iterated instead.

log

In SCXML, `<log>` is available to print messages evaluated on the data-model. An `expr` attribute is required and a `label` attribute is optional. PROMELA does feature a simple `printf` function, which can be used to provide basic support, though the absence of string variables in PROMELA is an issue here as only integer values can be used. When employed as part of the PROMELA data-model with the actual SCXML interpreter beneath, we were able to resolve the enumerated string integers to their string representation. During execution as an actual PROMELA program by the SPIN model checker, however, this statement can only print the string's numeric pendant.

One approach to alleviate this drawback is to check, whether the supplied expression is a sole string literal and to use it verbatim when writing the `printf` function call.

```
1  <log label="Outcome" expr="'pass'"/>
```

```
1  printf("Outcome: pass");
```

Listing 6.11: SCXML <log> modeled in PROMELA.

assign

The assign element requires a `location` attribute as a left-hand-side expression, identifying a variable or a member of a compound structure. To specify the value, either an `expr` attribute or a *legal data value* as a text child is given. The text child's syntax is usually expected to be JSON or some XML nodes if the data-model supports XML DOM representations as variables.

As with the extension of PROMELA declarations to support JSON structures (see section 6.3.2), the transformation will automatically resolve the assignment of JSON structures to matching compounds into a sequence of assignments valid in the PROMELA language. XML nodes, however, are not supported.

```
1  <assign location="Var1" expr="Var1 + 1"/>
```

```
1  Var1 = Var1 + 1;
```

```
1  <assign location="Var2">[2,4,6]</assign>
```

```
1  Var2[0] = 2;
2  Var2[1] = 4;
3  Var2[2] = 6;
```

Listing 6.12: SCXML <assign> modeled in PROMELA.

script

The script element is available to interpret any sequence of statements from the data-model's language, either specified as a text child or per URL in the `src` attribute. As we constructed the PROMELA data-model only to allow a subset of the actual PROMELA statement's syntax, we can just copy its contents verbatim into the resulting program.

send

Just as the `<raise>` element, the `<send>` element is available to deliver events, but where the former is only available to enqueue simple events on the interpreter's internal queue, the semantics of the latter are more versatile. It allows to enqueue events on both the interpreter's queues, deliver events to invoked components, via protocol specific I/O processors and to other SCXML sessions. The following paragraphs will present the issues and the solution implemented for all valid attributes and child elements.

`type[expr]` and `target[expr]`:

In the scope of a PROMELA program, it only makes sense to send events to channels actually modeled in the program. As a consequence, only the SCXML I/O processor type is available, others, such as the BasicHTTP I/O processor or any user-supplied I/O processor cannot be modeled as PROMELA will have no access to external resources during simulation or verification. If such an external system is crucial to the modeling of the overall system, it will have to be abstracted into a nested state-chart (compare section 6.4.7).

The following targets or expressions evaluating to those are available for the SCXML I/O processor:

- The `empty target` attribute or `_ioprocessors.scxml.location`: Both variants will enqueue an event on the interpreter's own *external* queue.
- `'#_internal'`: Enqueue an event on the interpreter's *internal* queue.
- `'#_' + invokeid`: Deliver an event to the external system invoked with the given identifier. In the PROMELA program, this only makes sense, if the invoked system is a nested SCXML state-chart.
- `'#_parent'`: If the state-chart sending the event was invoked, this variant will deliver an event to the parent state-chart who did the invocation.

The target `#_scxml_<SESSIONID>`, to send events to any SCXML session on the platform is unavailable. This is a limitation insofar that there can be no direct communication between grandfathers and grandchildren without passing events through the parents first. Technically, those machines are transformed to PROMELA state-machines and all of their queues are available, it is really a matter of not being able to concatenate the string literal `'#_scxml'` with the dynamic session identifier in the variable `_sessionid` (the same holds true, if an invoker is started with an `idlocation`, where the platform generates an identifier, as opposed to a user-supplied `id` string literal in the `<invoke>` element).

A `<send>` element with any other target or a type other than the default SCXML I/O processor will be ignored with a warning. If it is necessary to model communication with external systems outside of the scope of the interpreter, they have to be expressed as nested SCXML state-machines.

`event[expr]`:

This attribute contains either the event's name as a string literal or an expression evaluating to the same. When the machine is generated with simple events as enumerated integers, the respective integer will just be enqueued to the target message channel in PROMELA. For complex events, the value will be assigned to the event's `name` attribute before enqueued to the PROMELA target channel.

`id` and `idlocation`:

The purpose of the, mutually exclusive, `id` and `idlocation` attributes is to provide a *handle* for events sent. The first variation allows to specify an identifier explicitly, whereas the second will cause the platform to generate an identifier at the given data-model variable. Both variants are supported and will implicitly cause the transformation to create complex events, as the event structure will be defined with an additional field called `sendid`.

```
1  <send event="event1" idlocation="Var1"/>
```

```
1  typedef _event_t {
2    int name;
3    int sendid;
4  };
5  hidden int _lastSendId = 0;
6  ...
7  _event_t tmpE;
8  _lastSendId = _lastSendId + 1;
9  tmpE.sendid = _lastSendId;
10 tmpE.name = EVENT1;
```

Listing 6.13: SCXML `idlocation` of `<send>` modeled in PROMELA.

It is somewhat problematic to use auto-assigned send identifiers via the `idlocation` attribute of `<send>` as we will just pick a number by increasing `_lastSendId`. This will cause the state-space to grow considerably as every assignment of this variable causes a new state for the exhaustive search during verification. Unfortunately, SPIN does provide no way to ignore individual fields in compound types when determining the current state of the system, something which is readily available for global variables and, therefore, no problem inherent to the overall approach but rather a limitation of SPIN.

`delay[expr]`:

When the `<send>` element is used with a `delay` or `delayexpr` attribute, its content is evaluated as a CSS2 time designation: a numeric value representing a duration prepended by the unit `s` or `ms`. Due to the complete absence of string operators in PROMELA, such a representation is meaningless other than as an enumerated string literal. In order to keep its semantics, all time designations are normalized to milliseconds and the prepended unit dropped as implicit. This enables us to perform comparisons between delay durations as integers when the resulting PROMELA program is interpreted by SPIN.

Still, expressing time-related issues in PROMELA is rather complicated as there is no concept of time as such. For now, we will just proclaim that the usage of the `delay` or `delayexpr` entails the generation of complex events with an additional field called `delay`:

```
1  <send delayexpr="500" event="event1"/>
```

```
1  typedef _event_t {
2    int delay;
3    int name;
4    ...
5  };
6  _event_t tmpE;
7  tmpE.delay = 500;
8  tmpE.name = EVENT1;
```

Listing 6.14: Modeling delayed events in PROMELA.

Subsection 6.4.8 below details the implementation of delayed events in PROMELA.

`namelist`, `<param>` and `<content>`:

Sending events with data attached via the `namelist` attribute or either the `<param>` or `<content>` elements will also cause the transformation to create complex events for the PROMELA program. The syntactical analysis prior to the transformation will realize these constructs and extend the `_event_t` type accordingly:

```
<send event="event1" namelist="Var1">
  <param name="param1" expr="2"/>
</send>
```

```
typedef _event_data_t {
  int param1;
  int Var;
}
typedef _event_t {
  int name;
  _event_data_t data;
};
...
_event_t tmpE;
tmpE.data.Var1 = Var1;
tmpE.data.param1 = 2;
tmpE.name = EVENT1;
```

However, nested XML nodes either in `<content>` or `<param>` are not supported.

cancel

In SCXML, events send with a delay and not yet processed can subsequently be cancelled. The `<cancel>` element expects either a `sendid` or `sendidexpr` attribute containing or evaluating to the value of the `id` or `idlocation` attribute given when the event was send via the `<send>` element. Specifying such an attribute when sending an event causes the generation of complex events with the additional member field `sendid`. These events will be enqueued at the various target message channels. We will see in section 6.4.8 that we actually already enqueue delayed events at their target message channel but only dequeue them, when they are due. Therefore, canceling an event is to remove it from the message channels before they are processed.

PROMELA offers a language feature to remove messages with specific values from a given channel. While the question mark operator (`queue?variable`) is available to pop an event from the front of a message channel into a variable, the double question mark operator `queue??var1,..,varN,CONST,varN+2,..varM` will remove the first message matching all provided constants and fill the remaining values into the given variables. For message channels with compound data structures, the variables `var1,..,varM` correspond to the nested definition order of its type definition.

```
<send event="event1" id="foo" />
<cancel sendid="foo"/>
```

```
/* sending */
_event_t tmpE;
tmpE.name = EVENT1;
tmpE.sendid = FOO;
eQ!tmpE;

/* canceling */
do
:: eQ??tmpE.name, FOO;
:: iQ??tmpE.name, FOO;
:: else -> break;
od
```

With nested state-machines and delayed events, we do not necessarily know in which queue the event resides and will simply attempt to remove the event from all of them.

6.4.7 State Chart XML Invocations as Nested State-Machines

If an SCXML document employs a nested state-chart via an `<invoke>` with a `type` attribute of `scxml`, we can simply prepend a unique identifier (e.g. the invocation identifier) to all identifiers specific to the PROMELA representation of the respective state-chart and `run` the invoked state-chart as a concurrent process in PROMELA. There are, however, a few peculiarities with regard to the time when the invoked state-chart is available to send and receive events, as well as some issues when terminating such an invoked state-chart.

Time of the invocation

The SCXML recommendation mandates that all invoked entities are only actually instantiated at the end of a macro-step, i.e. before dequeuing an event. It might be the case that a potential invocation caused by entering a state is never executed as the respective state is left again before a stable configuration is assumed at the end of a macro-step. As such, we will enqueue all invocation requests in a special message channel when a state is

entered (e.g. `MAIN_start!INV_AE702`) and remove them again when the state is exited before the macro-step is completed (e.g. `MAIN_start??INV_AE702`). At the end of the macro-step, the remaining invocation requests are dequeued and the respective entities instantiated. In the case of nested SCXML invocation, it is necessary that the instantiated state-chart completed its initialization before any events can be processed. As such, we will make use of the optional `priority` parameter for `run` to suspend all other processes until the invoked state-chart processed its first micro-step:

```
1  MAIN_macroStep: skip;
2    int invokerId;
3    do
4    :: MAIN_start?invokerId -> {
5      if
6      :: invokerId == INV_AE702 -> {
7        run INV_ae702_run() priority 20;
8      }
9      :: else -> skip;
10     fi
11   }
12   :: else -> break;
13   od
```

The priority will be reset once we rescheduled the running state machine (see section 6.4.8 below).

Passing parameters

If the `<invoke>` element for a nested state-chart contains a `namelist` attribute or a set of `<param>` child elements, the respective variables are to be set in the invoked state-chart prior to running it:

```
1    <state id="s02">
2      <invoke type="http://www.w3.org/TR/scxml/" namelist="Var1">
3        <param name="Var2" expr="1"/>
4        ...
5      </invoke>
6    </state>
```

```
1  :: invokerId == INV_AE702 -> {
2    INV_ae702_Var1 = MAIN_VAR1;
3    INV_ae702_Var2 = 1;
4    run INV_ae702_run() priority 20;
5  }
```

Canceling delayed events

When the state containing the invocation is no longer in the active configuration at the end of a macro-step, the respective entity is to be terminated and all pending events cancelled. As we will see in the next section, when discussing delayed events, we did not actually wait for a specified amount of time before enqueueing an event at the target event queue as the PROMELA message channel, but insert according to the event's delay attribute. As such, we have to remove any pending events originating from the now terminated invoked state-chart from all other external event queues:

```
1  inline removePendingEventsForInvokerOnQueue(invokeIdentifier, queue) {
2    tmpIndex = 0;
3    do
4    :: tmpIndex < len(queue) -> {
5      queue?tmpE;
6      if
7      :: tmpE.delay == 0 || tmpE.invokeid != invokeIdentifier -> queue!tmpE;
8      :: else -> skip;
9      fi
10     tmpIndex++;
11   }
12   :: else -> break;
13   od
14 }
```

The inline function `removePendingEventsForInvokerOnQueue` is called for all external queues of other state-charts and will remove any pending events originating from the given invocation identifier.

When we identified the computational model of SCXML, nested state-chart invocations were one of the language features that could be abused to embed a Push-Down Automaton (PDA) and we argued to limit the invocation depth to retain DFA equivalence. In the current implementation, if a recursive invocation of the same state-chart is employed, the transformation process will simply not terminate as it generates ever more PROMELA state-machines with a unique identifier prefix and care has to be taken to avoid this situation.

6.4.8 Delayed Events

As mentioned briefly when discussing the `<send>` element above, the SCXML specification will allow to delay the delivery events until a certain amount of time has passed. There is no concept of *time* available in PROMELA, every

operation is assumed to be instantaneous. However, delaying events can still be expressed by sorting the events to be dequeued in the message channels and scheduling event processing respectively.

When the syntactical analysis discovers the usage of delayed events in a SCXML state-machine, the transformation will generate complex events with an additional field `delay`. To honor the semantics of such a delay, we can not actually wait for the specified time to elapse, but will merely sort the events in the message channels accordingly. In the PROMELA language, sending an event to a message channel is done via the exclamation mark operator (`eQ!tmpE`) and there is a double exclamation mark operator available for a *sorted send* to a message queue (`eQ!!tmpE`). It will prepend a message to the first existing message in the queue that succeeds it in numerical order. For compound structures the comparison is continued in declaration order of the fields until the order can be established or the structures are equal. While the semantics of the *sorted send* seem tempting to have events with a larger delay at the end of the queue, it is this last aspect related to compounds that complicates matters. If we consider an event structure as given in listing 6.14 for delayed events, determining the order of two undelayed events (`tmpE.delay = 0`) will not cause an undelayed event to be inserted as the last event with no delay, but the comparison will continue to take the event's name into account, causing the queue to be sorted by delay, then by the event's name and subsequently every other fields in the event's type declaration, which is clearly undesirable.

One possible solution is to introduce a sequence number as the second field, right behind the declaration of the delay field, but this would considerably increase the state space, as individual fields in a compound structure cannot be ignored by SPIN when determining the identity of two system states. Instead, we have to manually insert the event at the correct position in the target queue by iterating all existing events and inserting the new event with regard to its delay. Listing 6.15 shows a PROMELA inline function to do just that, it assumes that another part of the program just appended a delayed event to the end of the given queue and will resort the queue to account for the delay of this last event.

```
1  inline insertWithDelay(queue) {
2    _iwdIdx1 = 0;
3    _iwdQLength = len(queue) - 1;
4    do
5    :: _iwdIdx1 < _iwdQLength -> {
6      queue?_iwdTmpE;
7      _iwdQ[_iwdIdx1].name = _iwdTmpE.name;
8      _iwdQ[_iwdIdx1].delay = _iwdTmpE.delay;
9      ...
10     _iwdIdx1++;
11   }
12   :: else -> break;
13   od
14   queue?_iwdLastE;
15   _iwdInserted = false;
16   _iwdIdx2 = 0;
17   do
18   :: _iwdIdx2 < _iwdIdx1 -> {
19     _iwdTmpE.name = _iwdQ[_iwdIdx2].name;
20     _iwdTmpE.delay = _iwdQ[_iwdIdx2].delay;
21     ...
22     if
23     :: _iwdTmpE.delay > _iwdLastE.delay -> {
24       /* next event has larger delay -> insert here */
25       queue!_iwdLastE;
26       _iwdInserted = true;
27     }
28     :: else -> skip
29     fi;
30     queue!_iwdTmpE;
31     _iwdIdx2++;
32   }
33   :: else -> break;
34   od
35
36   if
37   :: !_iwdInserted -> queue!_iwdLastE;
38   :: else -> skip;
39   fi;
40 }
```

Last element in queue was just appended with the exclamation mark operator and might be at the wrong position as per its `delay`. Move all but the last element from the message channel into a temporary array of sufficient size.

Dequeue last item with potentially wrong position into `_iwdLastE` and prepare reinsertion.

Iterate temporary array with events from given queue and compare their individual `delay` fields to the last event in `_iwdLastE` to reinsert at the correct position in the given queue.

Last event was already at the correct position. Just reappend it to the end of the queue.

Listing 6.15: Inserting delayed events into a queue.

With this approach, it is guaranteed that the front of a message channel contains the next event due to be processed by a state-machine. However, with nested state-machines, it might still occur that a machine dequeues a delayed event, with another machine having an earlier event still enqueued. Therefore, whenever a PROMELA state-machine is about to dequeue an event, it will check whether other machines have earlier events to process. In that case, the machines

with the earlier events are prioritized and the current process yields execution. It will be rescheduled as soon as another machine is about to dequeue an event with the very same considerations.

Process priorities are expressible in PROMELA via `set_priority(pid, priority)`. Initially all state-machines will be started with a priority of `20` to ensure that they will perform their first micro-step uninterrupted. After scheduling with regard to their event delays, a running state-machine's process will have a priority of `10` and a dormant process a priority of `1`. When considering statements to be interleaved or executed, the SPIN interpreter will only regard processes with the highest priority and only advance lower priority processes if all higher prioritized processes are blocked. To realize this approach in PROMELA, we define three inline functions:

determineSmallestDelay(smallestDelay, queue)

This function is called for all external queues of PROMELA state-machines and will take the hitherto smallest delay in milliseconds and examines the front of the given queue for an eventual event with an even smaller delay. If the frontmost element has a smaller delay, `smallestDelay` is updated to this new, smaller delay.

```
1  inline determineSmallestDelay(smallestDelay, queue) {
2    queue?<tmpE>; /* receive front, but do not remove */
3    if
4    :: (tmpE.delay < smallestDelay) -> { smallestDelay = tmpE.delay; }
5    :: else -> skip;
6    fi;
7  }
```

rescheduleProcess(smallestDelay, procId, iQ, eQ)

After we have determined the smallest delay for all external event queues as PROMELA message channels, we can reschedule the state machines with events of the smallest delay for events due to have a higher priority:

```
1  inline rescheduleProcess(smallestDelay, procId, internalQ, externalQ) {
2    set_priority(procId, 1);
3    if
4    :: len(internalQ) > 0 -> set_priority(procId, 10);
5    :: else {
6      if
7      :: smallestDelay == tmpE.delay -> set_priority(procId, 10);
8      :: else -> skip;
9      fi;
10   }
11   fi;
12 }
```

advanceTime(increment, queue)

Finally, after the state-machines are scheduled with regard to the smallest delay in their respective external event queues, we have to "advance time" by decreasing all the event's delay field by the smallest delay we have identified. This ensures that delayed events will eventually move forward in the queue when events with the same delay are inserted in subsequent micro-steps.

```
1  inline advanceTime(increment, queue) {
2    tmpIndex = 0;
3    do
4    :: tmpIndex < len(queue) -> {
5      queue?tmpE;
6      if
7      :: tmpE.delay >= increment -> tmpE.delay = tmpE.delay - increment;
8      :: else -> skip;
9      fi
10     queue!tmpE;
11     tmpIndex++;
12   }
13   :: else -> break;
14   od
15 }
```

These three inline functions are called via another inline function `scheduleMachines` in a `d_step` block to prevent interleaving with statements from other processes during scheduling. By prioritizing those state-machines whose events have the smallest delay, we postpone machines with later events, thus preserving the semantics of delayed events. Still, later events send with a delay will be inserted at the correct place and eventually processed as all other delayed events are advanced by subtracting the smallest delay from all pending events.

It is important to note, that this approach still assumes that event processing is instantaneous. This is insufficient if external event sources are present (compare section 6.4.9), which will virtually deliver events infinitely fast when no delay is specified. The solution is either to always advance time by some small amount for each event dequeued, essentially associating a processing time with each step or by ensuring that external event source will enqueue events with a (short) delay.

6.4.9 Closing the System

Most actual SCXML documents will employ an I/O processor or invoke external entities to instantiate and communicate with external systems. These entities are most often essential to deliver the events that drive the state transitions. In order to formally verify the system described in the state-chart, we will need to transform it from an *open* to a *closed* system, i.e. a system that does not rely on any external entities. While it is possible to model the behavior of these external entities as composite states in a topmost, parallel state or as a nested SCXML document, it is more convenient to have respective event sources generated (semi-)automatically.

To this effect, the transformation will honor special XML comments to define list of events an external entity would deliver to the interpreter's external queue and creates these events, whenever the state-machine's queues are empty. Different approaches are available:

Semi-automatic event sources:

Providing an XML comment starting with the string literal `promela-event-all` in a single line will cause the transformation to write a block of statements to generate events for each event-descriptor associated with non-spontaneous transitions as child nodes of the comment's parent state. This is a convenient start to interleave most events that will enable transitions. The automatic generation for events is only prohibited, if the transition specifies a `cond` attribute in which an explicit access to a field in the event structure is found:

```
<scxml datamodel="promela">
  <!-- promela-event-all -->
  <state id="s0">
    <transition event="e1" />
    <transition event="e2"
                cond="_event.data == 'some string'" />
    <transition event="e3"
                cond="true" />
  </state>
</scxml>
```

```
macroStep: skip;
/* pop an event */
  if
  :: len(iQ) != 0 -> iQ ? _event   /* from internal queue */
  :: len(eQ) != 0 -> eQ ? _event   /* from external queue */
  :: else -> {
    /* external queue is empty -> generate external event */
    if
    :: _x.states[S0] -> {
      if
      :: true -> { _event.name = E1 }
      :: true -> { _event.name = E3 }
      fi
    }
    fi
  }
  fi;
```

Listing 6.16: Special XML comments can be used to automatically generate external event sources.

There is a variation available, wherein `promela-event-all` is replaced by `promela-event-all-but`, followed by a JSON list of event names that shall not automatically be generated. E.g. `promela-event-all-but [ "e3" ]` will prevent the creation of an event with the name `e3` in the listing above. Here, JSON is selected for convenience, as it allows to define a compound structure in a familiar syntax.

Qualified manual events:

The reason why `promela-event-all` will not create automatic events for transitions with a `cond` attribute in which a field of the event structure is accessed is the assumption that these transitions will rely on some attached data or properties other than the event's name. In this case, the event will have to be sufficiently qualified. This is possible with a XML comment containing a line starting with the string `promela-event`. It is expected to be followed by a JSON structure as an array with sufficiently qualified events:

```xml
<scxml datamodel="promela">
  <state id="s0">
    <!-- promela-event
      [ {"name": "e1"},
        {"name": "e2", "data": "some string"}] -->
    <transition event="e1" />
    <transition event="e2"
                cond="_event.data == 'some string'" />
    <transition event="e3" cond="true" />
  </state>
</scxml>
```

```
macroStep: skip;
/* pop an event */
    if
    :: len(iQ) != 0 -> iQ ? _event   /* from internal queue */
    :: len(eQ) != 0 -> eQ ? _event   /* from external queue */
    :: else -> {
      /* external queue is empty -> generate external event */
      if
      :: _x.states[S0] -> {
        if
        :: true -> {
          _event.name = E1;
        }
        :: true -> {
          _event.data = SOME_STRING;
          _event.name = E2;
        }
        fi
      }
      fi
    }
    fi;
```

Listing 6.17: Manually specifying sufficiently qualified events with attached data or other specific properties.

Events delivered from invokers:

If a XML comment node containing one of the special strings `promela-event`, `promela-event-all`, or `promela-event-all-but` is found as a child node of an `<invoke>` element, the respective events will only be generated when the respective invoker is started at the end of a macro-step. This is, essentially, semantically equivalent to having the comment in the containing state, but arguably more intuitive and allows us to set the events `invokeid` field to reflect the originating invoker.

Generating automatic events for external systems only when the external and internal queue of a PROMELA state-machine are empty might prevent some event sequences from ever being created, e.g. a machine always sending two events in succession within a micro-step will prevent this approach from interleaving automatic events in between. Then again, this is in line with the *perfect synchrony hypothesis* as event processing is assumed to be instantaneous and no event could be enqueued in between anyway.

`<finalize>`

Related to the issue of the invoking external entities is the `<finalize>` element containing statements to be processed when an event originating from such a system is received. Our transformation will create a block with respective statements right after a new event was established, either by popping from the external queue or by automatically generating as described above and dispatch on the event's `invokeid` field.

6.4.10 Length of the Event Queues

When we identified the computational model of SCXML as Turing complete in section 5.4, we argued for the necessity to limit the length if the internal and external event queues to retain DFA equivalence as is required for automaton based model-checking.

For a subset of SCXML documents we will discuss below, we can actually calculate the minimal required length of the queues. For all others, we will have to make an educated guess, attempt verification and increase the length progressively. Here, SPIN has the useful semantic that, per default, appending to a full message channel is a semantic error and verification can be reattempted with a manually increased size. The examples e.g. in listing 6.2 just over-provided on the length of the event queues with an arbitrary size of `100` for both.

When attempting to determine the minimum queue length required for a given SCXML state-chart, it is helpful to think about sources and sinks of events in the equivalent state-machine representation. With the state-machine as a closed system in PROMELA, all events are originating from executable content $\mathcal{X}(\widetilde{t})$ of global transitions either by `<raise>` or `<send>` or as the various `done.*` events for composite states and invocations. If no global transition enqueues events within a `<foreach>` block, we can determine the maximum number of events enqueued to either queue as the number of `<raise>` and `<send>` elements with respective targets within $\mathcal{X}(\widetilde{t})$, the number of `<uninvoke>` elements (see section 5.3.5) and the number of composite states in the optimal transition's exit set. The number can be adjusted somewhat if we account for alternative paths through the executable content caused by `<if>` / `<else>` blocks (without evaluating their condition).

Events will be consumed either by merely being in a configuration with no transitions enabled for an incoming event or by taking eventful transitions. The first case will potentially consume an infinite amount of events but is

very hard to determine without actually considering the complete state space just as SPIN does when verifying. The latter will always consume exactly one event.

Now, if we remember the construction for the SCXML state-machine in section 5.3.4, we can determine the maximum length of either queue by finding the sequence of global transitions that will have the most events enqueued. If a sequence of global transitions leads back to a previous global state, we will have to update the subsequent global states accordingly. If we find a cycle while updating, there might be a sequence of events that will enqueue evermore events without exhausting them in between and we can fall back on overproviding on the queues' length.

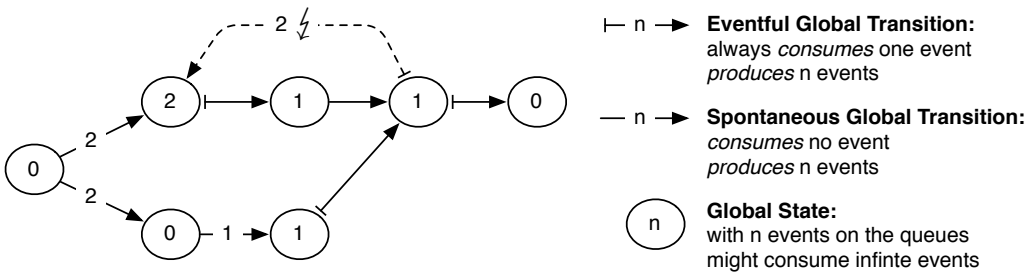


Figure 6.13.: Cycle in an SCXML state-machine that will cause the queues to grow infinitely.

Even if we cannot determine the maximum length of either event queue due to potential cycles that will grow the queues infinitely, the maximum number of events before accounting for cycles still might provide a general order of magnitude when estimating the length. Do note that the presence of such cycles does not imply that the queues will actually grow indefinitely as respective conditions on the transitions might very well prevent such a behavior.

Though, there is a genuine case, wherein an SCXML state-chart will actually cause either event queue to grow indefinitely (e.g. listing 6.18) and no upper bound can be specified. Such state-charts are, indeed, not verifiable with automaton-based model checking.

```
1 <scxml>
2   <state id="s0">
3     <transition target="s0">
4       <raise event="e" />
5     </transition>
6   </state>
7 </scxml>
```

Listing 6.18: SCXML state-chart with indefinitely growing internal queue.

The question is, though, if a state-chart actually requires infinite (or even just ridiculously large) queues, can it still perform a useful function in any regard other than as a theoretical consideration. Such a state-chart would just spend a lot of time processing events it enqueued itself and not be able to process external events as its actual, observable function. As such, it might be argued to be defective at least when employed to describe a dialog model for interaction.

6.4.11 Miscellaneous

There are a few more noteworthy features and solutions to problems encountered when modeling an SCXML state-machine interpreter in PROMELA. Even though they are not as central as those introduced in the sections above, they are still required to give a complete description of the transformation and allow traceability.

6.4.11.1 Atomicity and Interleaving

If multiple processes in a PROMELA are running concurrently with the highest priority among the processes and two or more are about to process executable (non-blocked) statements, the SPIN interpreter will branch out and create multiple transitions in the underlying Kripke structure. Due to the *perfect synchrony hypothesis*, implicitly mandated by SCXML, the statements during two macro-steps of different state-machines do not need to be interleaved, only the order and type of events to be processed by the state-machines. Therefore, we could wrap everything within a macro-step in an `atomic` block to prevent such interleaving. But as these blocks will lose their atomicity if a blocking statement is encountered, we even employ deterministic `d_step` blocks, wherein it is a semantical error to encounter a blocking statement, guaranteeing that each macro-step is processed uninterleaved by statements from another process.

However, there are some syntactical problems with PROMELA that prevent us from applying this concept thoroughly: It is not allowed to jump with the control flow out or into a `d_step` block. As we implemented all executable content within a micro-step as a sequence of respective PROMELA statements at a consecutively enumerated jump label, we cannot prevent interleaving of other processes in between micro-steps. While this is no problem semantically, it does increase the underlying Kripke structure and thus, the state-space during an exhaustive search when verifying LTL expressions.

We currently have no satisfying solution to this problem as embedding each macro-step in a complete `d_step` without any jumping for the control flow would entail considerable code duplication. One approach might be to notate executable content from a micro-step of a global transition not as a sequence of statements at a jump label, but as an inline function, but we have not attempted to do so.

6.4.11.2 The `In` Predicate

During runtime of the PROMELA program, the current configuration is always available in the boolean array `_x.states`. While we enumerated all global states for a numeric representation in the `_state` variable, we also enumerated all original states and maintain their activation status while executing a micro-step.

This enables us to offer the `In` predicate by testing the activation status of any given original state identifier via `_x.states[ORIG_ID]`. Offering the `In` predicate directly as a boolean function with the state name as a formal argument is not available in the PROMELA syntax as function calls are not really supported and inline functions will merely replace their invocation by their function body after replacing formal arguments and do not support return values.

6.4.11.3 Additional PROMELA Features via XML Comments

Despite all the automatic transformations from SCXML state-charts to PROMELA programs, it might sometimes be desirable to embed arbitrary PROMELA code and other language features specific to the verification process into the transformed program. Just as with the XML comments available to specify events from external entities when describing our approach to close the system in section 6.4.9, we support two more classes of features in these comments:

Embedding arbitrary PROMELA code:

Any XML comment containing a line starting with `promela-inline` can be followed by arbitrary PROMELA code. These comments are valid as executable content in `<onentry>`, `<onexit>` and `<transition>` elements and will be copied verbatim at the corresponding positions in the transformed program. Such code has no semantics when interpreted as part of the PROMELA data-model within the SCXML interpreter, but is available when the resulting PROMELA program is verified by the SPIN model checker. This enables an author to employ all the additional language elements from the PROMELA language, which are not supported by the data-model implementation:

```
1  <!-- promela-inline:
2    printf("Any sequence of promela statements is valid here");
3    ...
4  -->
```

LTL expressions for verification:

If such an XML comment contains a line starting with `promela-ltl`, the rest of the comment is assumed to contain a named LTL expression and will be appended to the eventual PROMELA program. This is useful to notate the temporal properties to be verified within the SCXML document as it allows to add such expressions whenever an author wants to assert an additional temporal property:

```
1  <!-- promela-ltl:
2    ltl property1 {
3      eventually (s == s1)
4    }
5  -->
```

Special PROMELA labels:

Another class of XML comments is available to insert specific PROMELA labels into the transformed program. These labels are available to check for liveliness properties of a system and to identify valid end states. There are three of these labels with specific semantics (from the PROMELA language reference[4]):

[4] `http://spinroot.com/spin/Man/Intro.html` (accessed November, 4th 2025)

- **Progress labels**: *[These] state the requirement that the labeled global state must be visited infinitely often in any infinite system execution*
- **Accept labels**: *The Spin generated verifiers can prove either the absence or presence of infinite runs that traverse at least one accept state in the global system state space infinitely often. The mechanism can be used, for instance, to prove LTL liveness properties.*
- **End labels**: *mark a control state that is acceptable as a valid termination point for all processes of the corresponding proctype*

The transformation will honor XML comments containing a line starting with `promela-progress`, `promela-accept` and `promela-end` respectively and generate respective PROMELA labels. Unfortunately, in the current form, this feature is not too useful as executable content is always wrapped in a `d_step` block, preventing e.g. the semantic of end-labels as control flow will never block and is never interleaved. It might be worth investigating whether a more elaborate approach, e.g. with annotating states and transitions and not only executable content would be more useful. For the moment, the major formalism for verification are the LTL expressions.

6.4.11.4 System Variables

We already implied the representation of the `_event` variable either as a single integer encoding the enumerated global states, or as a compound, non-recursive structure with a type automatically derived via syntactical analysis. The other system variables as introduced in section 4.1.2 are represented in the PROMELA program as follows:

- `_sessionid`: The session identifier is merely a free integer as an enumerated string, prepended by a unique prefix in the case of multiple state-charts in a PROMELA program:

```
1  #define INV_E25D0__SESSIONID 9 /* INV_e25d0__sessionid */
```

- `_name`: Just as with the session identifier, the name of the state-chart is also encoded as an enumerated string and available as a macro with a respective name:

```
1  #define INV_E25D0__NAME 10     /* INV_e25d0__name */
```

- `_ioprocessors`: Information about available I/O processors is contained in the compound variable `_ioprocessors`. However, as we are required to close the system, only information about the SCXML I/O processor to communicate with other state-charts is of any use. E.g. `test500` and `test501` from the SCXML IRP do employ their own SCXML I/O processor's location to send events to themselves. While this is of questionable usefulness in a real state-chart, we nevertheless represent the required `_ioprocessors` variable as:

```
1  typedef _ioprocessors_scxml_t {
2    int location;
3  };
4  ...
5  typedef _ioprocessors_t {
6    _ioprocessors_scxml_t scxml;
7  };
8  hidden _ioprocessors_t _ioprocessors;
```

- `_x`: The compound variable `_x` is mandated as the root element for any platform specific extensions in an SCXML interpreter. We already used this compound to represent the configuration of the original state-chart at any point during interpretation (see section 6.4.11.2).

6.4.11.5 Automatic Events

We introduced a set of events, a compliant SCXML interpreter will raise on various occasions in section 4.1.5. Of those, only the `done.state.[ID]` events signifying the exiting of a composite state and the `done.invoke.[invoke.ID]` to signal the termination of an invoked entity are available in the transformed PROMELA programs. We introduced an approach to model `<donedata>` for final states as simply raising a complex event to the internal queue when discussing required extensions to an interpreter in section 5.3.5 and the approach is readily transformable onto PROMELA.

The class of automatic events that are indeed inexpressible are all the `error` events from table 4.4:

error.communication
These events are to be raised whenever an event as a message to an external entity is undispatchable. As we had to close the system in section 6.4.9, there are no external entities for which delivery could fail for any reason.

error.execution
Events with a name of `error.execution` are raised, e.g. when a data-model specific statement contains a syntactical error. But PROMELA programs containing syntactical errors would already be rejected by the SPIN interpreter prior to any semantic analysis. Another instance described by the standard is the reception of an inexpressible event structure from an external system, something which is, again inapplicable in a closed system. Finally dispatching a message via an unavailable I/O processor would also raise such an error and is, also, inapplicable in a closed system.

error.platform
The SCXML recommendation mandates no occasion when an `error.platform` event is to be raised. It is rather a reserved event name for any platform specific error conditions. E.g. our implementation will raise `error.platform` when the given XML document contains no `<scxml>` element or the creation of the SCXML DOM generally fails for any reason. No such error condition can occur in a PROMELA program.

6.4.12 Verifying Application Specific Instantiations

In section 4.2, we introduced five approaches for the application-specific instantiation of SCXML state-charts to connect their behavior to an actual problem domain.

In general, we can already put all the state-chart's properties modeled in PROMELA in temporal relations. That is, we can postulate properties about sequences of configurations, about data-model valuation or events with all their attached data and any combination of these. Now, if we want to make application specific instantiations available for temporal verification, we just need to model them in PROMELA in order to argue about them in LTL specifications.

Interpreter Callbacks

We introduced a series of interpreter callbacks where an application can register its functionality. The various callbacks were rather fine granular and provided the possibility to make many phases of the within macro- and micro-steps observable to an application. When attempting to formally verify temporal properties, an important issue with regard to granularity is the fact that we grouped all executable content within a micro-step as a deterministic `d_step`. This prevents interleaving from other processes and ensures that the complete micro-step is processed uninterleaved by other processes. In particular, this also prevents the interleaving with LTL specifications, which are modeled as specific concurrent processes when verified with SPIN. As such, we would have to unwrap the micro-step from the `d_step` block, potentially causing an even larger state-space explosion. Otherwise, we can formulate LTL expressions to argue about temporal relations of all modeled properties, but only in between micro-steps.

New I/O Processor Types

Application specific instantiation via a new I/O processor would deliver events send from within the state-chart with respective type onto the application, where reception of events would trigger respective functionality. With a closed system, there is nowhere to send these events. One approach could be to enqueue events targeted to such an I/O processor onto a PROMELA message channel, which is emptied in between macro- or micro-steps. This would allow to argue about the presence of certain events in this very message channel when specific preconditions were given. However, our transformation does currently not support such an approach.

New Invoker Types

If the specific instances within an application are to be controlled, an approach wherein new respective invoker types are introduced and instantiated is suitable. Currently, we will only support invokers of type `scxml` as nested state-charts and all other types only as event sources via the notation in the special XML comments introduced in section 6.4.9. If temporal relations about events sent to such an invoker are to be verified, the transformation could employ an approach similar as described for custom I/O processors, wherein these events are enqueued in a transient PROMELA message channel, which is empties in between macro- or micro-steps.

Data-Model Extensions

Exposing specific application-related objects in the state-chart directly is actually discouraged as the sole source of changes to the data-model ought to be events as implied in the SCXML recommendation. However, it is

allowed to trigger functionality from the application via functions made available in the data-model's language and, indeed, this is the approach taken e.g. with the Nvidia Shield SCXML dialog model we will see in section 7.1.

If we are to make the calling of these functions available to LTL expressions, we can either encode their formal arguments as a PROMELA type and enqueue a respective message in a transient message channel for every calling within a micro-step or, as we will do for the Nvidia Shield example, merely encode the fact that they were called within a micro-step as a respective boolean value which is reset in between micro-steps. For the Nvidia Shield dialog model, this is perfectly sufficient as none of these functions actually takes any arguments from the state-chart's data-model, nor is there a micro-step in which a function is called more than once. However, for a more general approach the transient message channels would ultimately be more apt.

Custom Executable Content

Just as with data-model extensions in the form of exposing application-specific functions, we can think of custom executable content in the form of special `<ns:custom>` elements as the invocation of a function and apply the very same approach as above.

Not all of these approaches are actually implemented in our transformation, but there does not seem to be a principal problem inherent to the overall approach when making these application specific instantiations available for LTL expressions.

6.4.13 Evaluating the Resulting PROMELA Programs

With the XSLT transformation of the 181 data-model agnostic SCXML tests for the PROMELA data-model and the subsequent transformation onto actual PROMELA programs described above, we can formally verify whether the PROMELA programs will pass the tests. All non-manual SCXML tests are considered passed if the interpreter assumes a final configuration with only the state `pass` active. As such, we can verify all tests with the following LTL expression:

```
#define s1 1        /* from "active:{pass}" */
unsigned s : 2;     /* current state */
...
ltl { eventually (s == s1) }
```

The transformation of state-charts to state-machines causes the state `pass` to be encoded as `active:{pass}` along with history assignments and already entered states as part of the identifier. The tests not eventually assuming state `pass` are listed in table 6.6 along with the functional requirement of SCXML they tested.

6.5 Missing Language Features

It is unfortunate that the tested requirement does not, in most cases, reflect the cause for failing the test. E.g. `test329` tests that the various system variables are not modifiable. Failing the test does not imply that one might accidentally alter the system variables in the resulting PROMELA program, but they are modeled as constant macros and assigning a value to them is already a syntactical error. The actual reasons for failing formal verification of the tests transformed to PROMELA programs are given in table 6.7. The following list gives a detailed explanation as to why the various features required or implied by the failed tests are inexpressible, or what would be required to support them:

Errors from the platform at runtime:

The SCXML standard specifies various occasions, when an error event is delivered via the interpreter's internal queue. Error events indicate syntactical errors in data-model expressions, missing / invalid attributes at various XML elements or timeouts when contacting remote entities. It is not possible to model this behavior in general, as the SPIN interpreter will outright refuse to process a PROMELA program with syntax errors. Most of the respective tests would pass, if the author of the SCXML document would have been more careful, not to introduce syntactical errors. The only useful feature that is actually non-expressible with the automated transformation are communication errors with external entities. Then again it is up to the modeling of the external entity as either a nested SCXML state-machine or a random event source in PROMELA to exhibit this behavior.

Class	Sec.	#Tests
Core Constructs		**38 / 40**
Events	3.12	2 / 4
`test401`: *Errors are delivered via internal queue*		
`test402`: *Errors events are treated as regular events*		
Executable Content		**9 / 13**
Foreach	4.6	4 / 7
`test152`: *Raise error for invalid array*		
`test156`: *Error in iteration breaks loop*		
`test525`: *Iterate on shallow copy of array*		
Evaluation	4.9	1 / 2
`test159`: *Break evaluation on error*		
Data-Model and Data Manipulation		**26 / 50**
Data	5.3	5 / 7
`test277`: *Error for invalid value expression*		
`test280`: *Declare vs. assign with late binding*		
Assign	5.4	2 / 4
`test286`: *error for invalid location*		
`test487`: *error for invalid assignment*		
Donedata	5.5	0 / 1
`test294`: *Event's 'donedata' from content or params*		
Content	5.6	2 / 3
`test528`: *'error.execution' for illegal expression*		
Param	5.7	0 / 3
`test298`: *Error for invalid 'location' expression*		
`test343`: *Invalid 'expr' causes empty data*		
`test488`: *Error for invalid 'expr' expression*		
Expressions	5.9	1 / 8
`test307`: *Variable declared in script element*		
`test309`: *Uninterpretable boolean expression is false*		
`test311`: *Error for invalid location expression*		
`test312`: *Error for invalid value expression*		
`test313`: *Error for ill-formed expression*		
`test314`: *Error raised at correct time*		
`test344`: *Error for invalid transition condition*		

Class	Sec.	#Tests
Data-Model and Data Manipulation (contd.)		
System Variables	5.10	12 / 20
`test322`: *Error when assigning to '_sessionid'*		
`test324`: *Error when assigning to '_name'*		
`test325`: *'_ioprocessors' is bound at startup*		
`test326`: *'_ioprocessors' is bound until termination*		
`test329`: *System variables are non-modifiable*		
`test331`: *An event's type is set correctly*		
`test332`: *An event's send id is set correctly*		
`test346`: *Error when assigning to any system variable*		
External Communications		**42 / 51**
Send	6.2	14 / 19
`test178`: *Duplicate variables per param and namelist*		
`test194`: *Error for invalid target*		
`test199`: *Error for invalid type*		
`test521`: *Error for undispatchable event*		
`test553`: *Error for illegal identifier in namelist*		
Invoke	6.4	25 / 29
`test216`: *Support 'srcexpr' for invoke*		
`test224`: *Generate idlocation as 'stateid.platformid'*		
`test530`: *Content 'src' is evaluated at runtime*		
`test554`: *Error for illegal identifier in namelist*		
Data-Models		**0 / 51**
Specific to XPath, ECMAScript, NULL → inapplicable		
Event I/O Processors		**17 / 28**
SCXML Event I/O Processor	C.1	14 / 16
`test354`: *'event.data' via namelist, param or content*		
`test496`: *Error for invalid target*		
Basic HTTP Event I/O Processor	C.2	3 / 12
`test509`: *Accept HTTP POST requests*		
`test513`: *HTTP 200 reply for well-formed request*		
`test518`: *Namelist entries as POST parameters*		
`test519`: *Param elements as POST parameters*		
`test520`: *Content element becomes request body*		
`test531`: *Param '_scxmleventname' is used as event name*		
`test532`: *HTTP type as event name if none specified*		
`test534`: *Event at send becomes '_scxmleventname'*		
`test577`: *Error for missing target*		
Total		**132 / 182**

Table 6.6.: List of SCXML IRP tests with PROMELA data-model failing formal verification with SPIN.

String operations:

Given the central role of strings in general, this class of tests is surprisingly small. The introduction of string literals as enumerated integers and the respective identity operator sufficed for a large class of tests that are now being passed. More elaborate operations on strings, such as concatenation, substring extraction, or finding one string in another are currently not supported and it is not obvious how one would realize these in PROMELA with only integer arithmetics available.

Encoding atomic string literals as prime numbers and concatenated strings as their product will allow to support a `contains` operation but prevents the `identity` operation as the order of factors is lost. Expressing concatenated strings as sorted arrays of atomic string literals and pre-calculating all the relevant relations into a set of $M_{pred}(\vec{s_1}, \vec{s_2}) \rightarrow \{\texttt{true}, \texttt{false}, n \in \mathbb{N}\}$ matrices might be a feasible solution to support the more elaborate string operations in special cases. Though, given the Turing completeness of SCXML, we can, in the general case, not know the longest concatenation sequence and over-providing strings as arrays with a ridiculous size will have a detrimental effect on memory requirements and runtime when running the model-checker. For special cases, we can detect the maximum number of atomic string literals in a concatenated string and calculate the length of $\vec{s_i}$. While this would, most likely help to pass more tests, its applicability in more elaborate state-charts is at least questionable.

Cause	#	Tests
Relies on errors raised by platform	27	152, 156, 159, 194, 199, 277, 286, 298, 311, 312, 313, 314, 331, 332, 343, 344, 346, 401, 402, 487, 488, 496, 521, 528, 553, 554, 577
String operations required	5	224, 518, 519, 520, 534
Assumes open HTTP socket	4	509, 513, 531, 532
Inexpressible event structure	3	178, 294, 354
Assigning to system variables	3	322, 324, 329
Compounds treated as atoms	2	325, 326
Late data binding or implicit variables	1	280, 307
Dynamic URL for nested machine	1	216
Syntax error evaluates to `false`	1	309
Shallow copies in foreach	1	525
XML DOM node in variable	1	530

Table 6.7.: Reasons for failing formal verification for PROMELA programs transformed from SCXML tests with PROMELA data-model.

Open HTTP socket:

The tests for the `basichttp` I/O processor require the SCXML runtime to accept HTTP requests at `_ioprocessors.basichttp.location` and make assumptions about the structure of the delivered events. While this is possible for the PROMELA data-model employed as part of SCXML documents, there is no corresponding language support in PROMELA when interpreted by SPIN.

Inexpressible event structure:

All data-models specified or even required by the SCXML standard employ dynamic typing. The PROMELA language is statically typed which is a problem with some tests. When the transformation detects the necessity for complex events as opposed to simple integers, a nested type definition with the event's implied fields is automatically derived from all PROMELA expressions in the document. For some tests, this type definition is impossible to specify, e.g. accessing both `_event.data` and `_event.data.Var1` as atoms in different events requires the `data` field to be both, an atom and a compound with a field `Var1` which is not expressible as a single type definition. One solution might be to employ different event type definitions for different classes of events, but the pragmatic solution for now is to require a single consistent type for all events.

Assigning to system variables:

This class of failed tests is due to our modeling of system variables as enumerated integers via macros. The SCXML specification mandates for a platform to raise an `error.execution` whereas the transformed PROMELA programs generate a syntactical error and SPIN refuses interpretation. Assigning e.g. to the predefined system variable `_sessionid` will cause the generation of the following PROMELA snippet:

```
1 <scxml datamodel="promela" name="machineName">
2   <datamodel>
3     <data id="Var1" type="string" expr="_sessionid"/>
4   </datamodel>
5   <state id="s0">
6     <onentry>
7       <assign location="_sessionid" expr="'otherName'"/>
8     </onentry>
9 ...
```

```
1  /* string literals */
2  #define _NAME 2          /* _name */
3  #define _SESSIONID 1     /* _sessionid */
4  #define MACHINENAME 1    /* machineName */
5  #define OTHERNAME 3      /* otherName */
6  ...
7  atomic {
8  /* transition to state s0 */
9    _SESSIONID = OTHERNAME;
10   goto microStep;
11 }
12 ...
```

Listing 6.19: Assigning to system variables is already a syntactical error.

After the pre-processor resolved macros, line 9 in the right column of figure 6.19 reads as `1 = 3`, which causes SPIN to refuse the PROMELA program. While the tests formally register as failed, the intention, to prevent a state-machine from assigning to system variables, is preserved.

Treating compounds as atoms:

In PROMELA, the boolean value of a compound is undefined and to use a non-atomic field in a boolean expression is a syntactical error. Similarly, it is not possible to assign one compound directly to another compound, even if their types are identical. In both instances, the transformation could resolve the issue by defining the boolean value of a compound as the `or`'ed boolean value of its constituting atoms and rectify compound assignment into assignments of the individual, constituting atoms. This is not done in the current implementation.

Late data binding and dynamic / implicit variable declarations:

The SCXML specification assumes that variables can be introduced at any time

- to the data-model via a `<data>` element nested in a state (with late data binding enabled),
- from the data-model by simply declaring the variable dynamically in a `<script>` element and
- implicitly, by using a new identifier as the `item` or `index` attribute of an `<foreach>` element.

This is generally not expressible with the PROMELA language. Variables in PROMELA are either declared globally or bound to the scope of a process. Checking at runtime whether a variable is declared is not possible and accessing an undeclared variable in any expression is a syntactical error. For dynamical and implicit variable declarations, the code analysis will identify missing declarations via syntactical analysis of all PROMELA code fragments and introduce hidden variables to the global scope.

```
<scxml datamodel="promela">
  <datamodel>
    <data id="Var1" type="int[2]">
      Var1[0] = 1;
      Var1[1] = 2;
    </data>
  </datamodel>
  <state id="s0">
    <onentry>
      <foreach array="Var1" item="Var2" index="Var3" />
      <script>int Var4 = 5</script>
    </onentry>
  ...
```

```
...
/* \gls{data-model} variables per <data> */
int Var1[2];
/* Implicitly or dynamically declared variables are
   hidden and the code analysis makes an effort to
   use minimal bit-width */
hidden int Var2;
hidden unsigned Var3 : 2;
hidden unsigned Var4 : 3;
...
init {
  Var1[0] = 1;
  Var1[1] = 2;
...
```

Listing 6.20: Implicitly and dynamically declared variables are introduced as hidden variables in the global scope.

Nested `<data>` elements with a late data binding are refused by the transformation. The strong implication regarding the semantics of a "yet undeclared variable" whose *declaredness* can be tested is inexpressible in the PROMELA language. As this notion is not as explicit with dynamic and implicit declarations, we choose to introduce them automatically. This is not a dire implication as late data binding can always be expressed by making the "declaredness of a variable" explicit in an accompanying boolean variable per identifier. Then again, the whole feature is somewhat questionable and can easily be avoided.

Dynamic source URLs for nested machines:

The source of a nested state-chart, or any external resource for that matter, has to be known when transforming the state-chart onto an PROMELA program. During verification or simulation, there is no language support to load external resources nor to perform any transformation on their DOM in the case of XML documents. As such, it is not possible to support the `srcexpr` attribute for invokers in general. However, if the set of possible URLs is finite and known at transformation time, all the potential state-charts can be transformed into the resulting PROMELA program and the respective machine started in an `if` block, conditionalized by the content of the `srcexpr` value at runtime. The gain in expressiveness somewhat pales in comparison to the additional effort, though.

Syntax error in boolean expression evaluates to `false`:

In `test309`, it is assumed that a syntax error in a boolean expression is evaluated as `false`. As with the problems of errors raised for syntactical errors, the SPIN interpreter will just refuse to interpret PROMELA programs with syntax errors. And, again, this class of errors is easily avoided by using syntactically correct expressions.

Shallow copies in foreach:

When iterating an array structure via the `<foreach>` element, the standard mandates for an interpreter to

create a shallow copy prior to iterating. This prevents alterations to the array within an iteration to cause undesired side effects. In `test525`, this is validated by expanding the array in an iteration. Neither expanding an array, nor the shallow copy are implemented in the current transformation. The first is impossible to do in PROMELA as arrays may not shrink or grow after they have been declared, the latter is expressible by duplicating each array declaration for which an iteration is employed and use the duplicate for the shallow copy prior to iterating.

XML DOM node in variable:

In `test530` occasion, the content for a nested invoker is given via the `<content>`'s `expr` attribute and initialized to a DOM subtree. It is, in the general case, not possible to resolve the content of the given expression at transformation time and substitute it accordingly when processing the `<content>` element. In our original SCXML interpreter, it is up to the data-model to support XML DOM nodes for variables. While it is possible to model a DOM-like API in PROMELA, the effort required is immense.

There may be other reasons for failing formal verification of the tests which only become visible once one of the classes of reasons for failing above is resolved. For example, `test509` fails formal verification as it requires an open HTTP socket, here this error hides another issue of `test509` requiring substring searching on the event's received data. Concluding the discussion about the transformation of SCXML documents with the PROMELA data-model onto actual PROMELA programs as they can be interpreted by the SPIN model-checker, figure 6.14 summarizes this step in the scope of the overall approach.

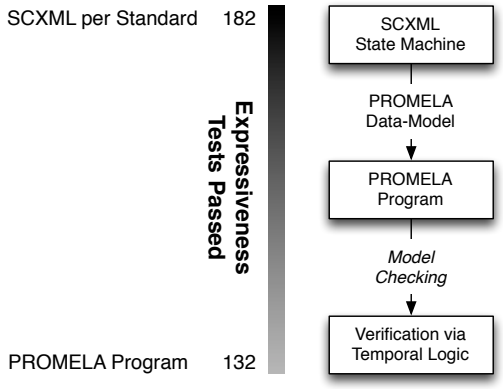


Figure 6.14.: SCXML tests passed when formally verified as PROMELA programs via SPIN.

Part III.
Applications and Conclusion

7 Applications

In the previous parts of this thesis, we detailed our approach to verify multimodal dialogs given as State Chart eXtensible Markup Language (SCXML) documents via Linear Temporal Logic (LTL) as the main contribution of this thesis and went through great length to motivate its relevance for general interaction in pervasive environments. This last part will complete the overall narrative by 1. applying the approach of formal verification to a non-trivial, real-world SCXML dialog model and 2. describe our SCXML interpreter platform along with a few selected applications and relevant tools we developed over the years before we 3. finally conclude the thesis along with some critical remarks and an outlook to follow-up work.

We start by transforming the complete SCXML dialog model of the Nvidia Shield handheld gaming console onto PRocess MEta LAnguage (PROMELA). This serves as a concluding evaluation of the applicability of our core contribution to a description of interaction employed in an actual consumer product. We will identify a couple of short-comings of the original transformation and present two adaptations to improve its performance considerably.

Subsequently introducing our interpreter platform, along with the various Modality Component (MC) and a selection of user interfaces and developer tools we implemented might help to clarify any remaining questions with regard to modeling interaction with SCXML. While not necessarily a scientific contribution, the platform is something we developed over the course of three years and was essential in enabling this contribution and the corresponding evaluation.

7.1 Evaluation with the Nvidia Shield Gaming Console

In this chapter we will transform a real-world SCXML document employed, in a more recent version, by the Nvidia Corporation to control the user interface of their Shield handheld gaming console [KN14a] to a PROMELA program. The document was made available by Nvidia and permission to publish was granted. The complete original document is listed in the appendix (see A.4). It is interpreted on the handheld gaming console by a custom SCXML interpreter and implicitly uses Lua[1] as the language for its embedded data-model. While the 231 World Wide Web Consortium (W3C) tests for SCXML employed when the transformation was introduced in section 5.3 did establish the subset of SCXML that is accessible to model-checking, this document gives an impression about the performance of the transformation and whether it is applicable to real-world interaction descriptions.

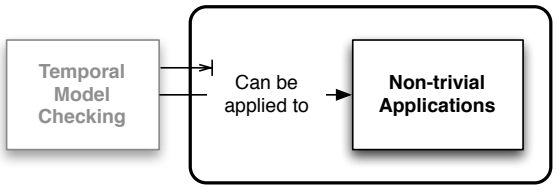


Figure 7.1.: This chapter tackles the proposition "*The approach is applicable for non-trivial applications.*" from the main argument of this thesis (cf. figure 1.1).

The document features `58` states, one of them `parallel` and five `history` pseudo-states of which one is `deep`. There are `126` transitions in the document and the upper bound $|\widetilde{\mathcal{S}}|_u$ for the number of global states per formula 5.29 is `77,785` with `464` distinct legal active configurations, the rest due to `168` different history assignments.

When trying to apply the transformation from SCXML state-charts to state-machines described in section 5.3 for the Nvidia Shield SCMXL document, there are a number of issues enumerated below and detailed in the following sections.

1. The transformation to state-machines is not defined for the employed Lua data-model.
2. The state-chart relies on external events to drive the state transitions, these have to be modeled.
3. The time required to identify the potentially optimal transition sets in $\widetilde{\mathcal{T}}(i)$ for the (at worst) `77,785` global states.
4. The sheer size of the resulting PROMELA program and the entailing performance of subsequent steps when every global state is written explicitly.

The first issue is specific to the given SCXML document and an author aiming for formal verifiability would most likely already start by using the PROMELA data-model (or a data-model with a suitable transformation onto

[1] `http://www.lua.org`

PROMELA). The second point is expected to be rather common when verifying state-charts: self-contained state-charts, driving their own transitions via events raised internally and not relying on external events are really only suited for tests of the platform, but not for actual descriptions of interaction or any state-chart that needs to react to external events.

The last two points became evident with the application of the transformation to the Nvidia Shield SCXML document. While the formalisms and the construction described for the transformation made sense didactically and were presumably more comprehensible, its direct application raised performance issues described and solved below. These solutions are applicable to any transformation onto PROMELA and were verified with the W3C SCXML tests.

7.1.1 Transforming the Data-Model from Lua to PROMELA

The SCXML document for the Nvidia Shield handheld console implicitly employs a data-model with the syntax and semantics of Lua, wherein the transformation from section 5.3 is only defined for the PROMELA data-model. As such, we have to transform all data-model specific declarations, expressions and statements onto PROMELA. Most of the semantics with regard to controlling the actual user interface are realized by the platform via entering and exiting states [KN14a]. As such, language features of data-model are only used sparingly and most often, there is a direct correspondence to PROMELA expressions:

1. **Implicit Lua Data-model:**
   The SCXML interpreter of the Nvidia Shield console will implicitly use the Lua data-model. We have to set the data-model to `promela` explicitly via the `datamodel` attribute of the topmost `<scxml>` element.

2. **Boolean Operators:**
   In Lua, boolean operators are identified by their literal names `and`, `or`, and `not`. They have a direct equivalent in PROMELA as `&&`, `||`, and `!`.

3. **String Equality:**
   String comparisons in Lua are performed with the `~=` operator, wherein our extension for strings in PROMELA employs the `==` operator as defined for integers.

4. **Function Calls:**
   This is currently still an issue. Not only are there no actual function calls in PROMELA syntactically, the employed functions are also specific to Nvidia's SCXML interpreter (their Lua data-model to be specific). The solution, at the moment, is to handle these function calls as assignment to boolean variables with the same name and reset their value prior to each micro-step. This will allow to verify that these functions were called (their respective boolean value will be true), but requires to edit the PROMELA files manually. However, the very same approach could well be automated.

7.1.2 Faster Identification of the Potentially Optimal Transition Sets

To create the SCXML state-machine from the state-chart, the algorithm in 2, requires to establish $\widetilde{\mathcal{T}}(i)$ as the sorted set of potentially concurrently enabled transitions that may form an optimal transition set. $\widetilde{\mathcal{T}}(i)$ is a subset of the power-set $\mathcal{P}(T(i))$ of all the transitions that are direct descendants of the state-charts active states. This set has to be established for every possible global state. We already know the number of global states to be `77,785` at worst and can calculate the distribution of $|T(i)|$ for every distinct legal active configuration (see figure 7.2).

We can see, for example, that there are 32 configurations in the SCXML document with 18 transitions potentially enabled. The majority of configurations features around 10-15 potentially enabled transitions. With this distribution we can give an upper bound to the number of transition sets we will have to consider to establish $\widetilde{\mathcal{T}}(i)$ for every global state:

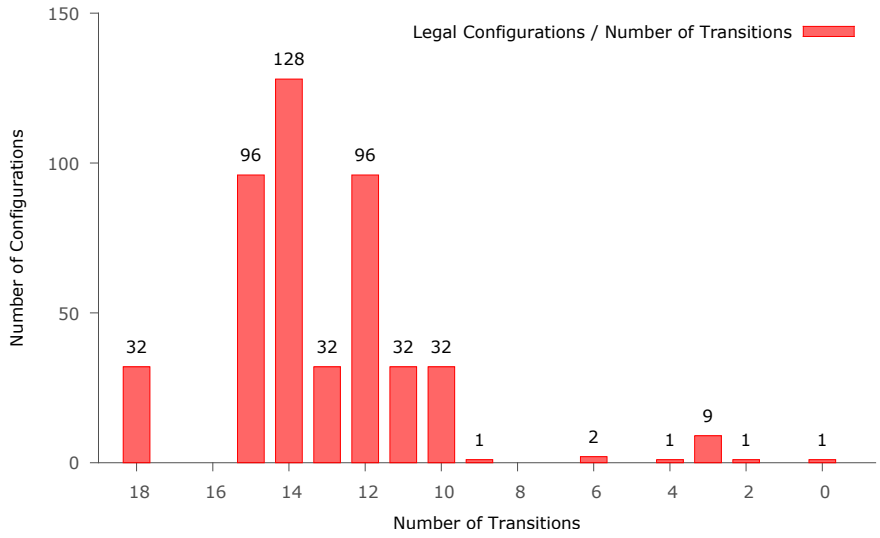


Figure 7.2.: Number of potentially enabled transitions per legal configuration in the Nvidia Shield SCXML document.

$$\sum_{i=1}^{77{,}785} |\mathcal{P}(T(i))| := \tag{7.1}$$

$$\begin{aligned}
& & 32 * 2^{18} & = 8,388,608 \\
& + & 96 * 2^{15} & = 3,145,728 \\
& + & 128 * 2^{14} & = 2,097,152 \\
& + & 32 * 2^{13} & = 262,144 \\
& + & 96 * 2^{12} & = 393,216 \\
& + & 32 * 2^{11} & = 65,536 \\
& + & 32 * 2^{10} & = 32,768 \\
& + & 1 * 2^{9} & = 512 \\
& + & 2 * 2^{6} & = 128 \\
& + & 1 * 2^{4} & = 16 \\
& + & 9 * 2^{3} & = 72 \\
& + & 1 * 2^{2} & = 4 \\
& + & 1 * 2^{0} & = 1 \\
\hline
& & & = 14,385,885 \\
& * & 168 & = 2,416,828,680
\end{aligned}$$

Every one of the `2,416,828,680` transition sets will have to be checked for the invalidation criteria in equations 5.8 - 5.12 and subsequently sorted per global state. If we assume that we can check 500,000 transition sets per second for their validity to form a potential optimal enabled set, we would need around 80 minutes just to identify valid transition sets. There are two observations that allow to reduce this number dramatically:

Observation 1: The set and ordering of global transitions only depends on the interpreter's active configuration:

$$\widetilde{s}_a(i) = \widetilde{s}_a(j) \Rightarrow \widetilde{\mathcal{T}}(i) = \widetilde{\mathcal{T}}(j) \tag{7.2}$$

This means, foremost, that we can skip the calculation of $\widetilde{\mathcal{T}}(i)$ for global states that differ only in their history assignments $s_h(i)$ or the set of states with nested `<data>` elements already visited in $s_d(i)$. This allows us to cache and reuse $\widetilde{\mathcal{T}}(i)$ for global states with the same active configuration, already dropping the multiplication by `168` from equation 7.1.

It is important to note, that the same does not hold true for the executable content in $\mathcal{X}(\widetilde{t})$ as, e.g., transitions in history elements and associated executable content is only processed on the initial entry.

Observation 2: Starting with the power-set $\mathcal{P}(T(i))$ and subsequently invalidating subsets is far from optimal. We can exploit properties from definition 12 for an *optimal transition set* to create a more sane starting set to reduce.

The fundamental property to exploit is the fact that every atomic state in a state-chart can only ever enable a single transition (see definition 10). Therefore, transition sets containing transitions with nested or identical source states can never form an optimal transition set as either the more deeply nested or the first transition in document order would take precedence. We can use this property for a reduction in transition sets to be considered. The basic idea is to (i) start with the sets of transitions from the most deeply nested states (or their first parent with actual transitions) in a given active configuration $s_a(i)$, (ii) establish every combination of picking one or no transition from each each set, (iii) create follow-up transition sets by traversing the nested states with transitions towards the root. We will illustrate the approach with the example state hierarchy in figure 7.3.

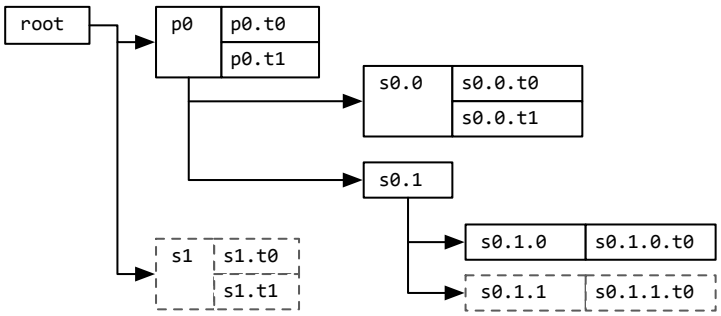


Figure 7.3.: Example state hierarchy to illustrate optimized transition set selection.

For an active configuration containing `p0, s0.0, s0.1` and `s0.1.0`, the improved algorithm will first identify the most deeply nested states containing transition elements. In this case `s0.0` and `s0.1.0`. Their transitions form the first set of transition sets to combine. From each of those state's transitions, we will pick one or none for the combined set. If an optimal transition set contains one transition from either set, it cannot contain transitions from `p0` as each atomic state con only activate a single transition. Consequentially, no two transitions from the same atomic state be contained in an optimal transition set either. Leading to the first set of candidates for potentially optimal transition sets:

```
{},
{s0.0.t0},
{s0.0.t1},
{s0.0.t0, s0.1.0.t0},
{s0.0.t1, s0.1.0.t0}
```

Subsequently, we move up in the state hierarchy towards the root and create new set of transition sets to combine. The next state with transitions is the parallel state `p0`. For a transition from `p0` to be part of an optimal transition set, no transitions form its child states may be contained. As such the next set of transition sets to combine contains only the transitions from `p0`, leading to the next set of candidates for potentially optimal transition sets as:

```
{},
{p0.t0},
{p0.t1}
```

Joining both sets leads to only *six* potentially optimal transition sets as opposed to $2^5 = 32$ sets. It is plain to see that this optimized selection for a starting set to reduce and sort as $\widetilde{\mathcal{T}}(i)$ is way smaller while still containing all potential optimal transition sets. Employing only this optimization for the Nvidia Shield SCXML document will drop the number of transition sets that have to be considered by a factor of ∼`100` to `24,093,552`.

Both optimizations can be employed at the same time and will reduce the number of potentially optimal transition sets to be considered to establish $\widetilde{\mathcal{T}}(i)$ by a factor of ∼`16,852` from `2,416,828,680` to `143,414` for the Nvidia Shield SCXML document.

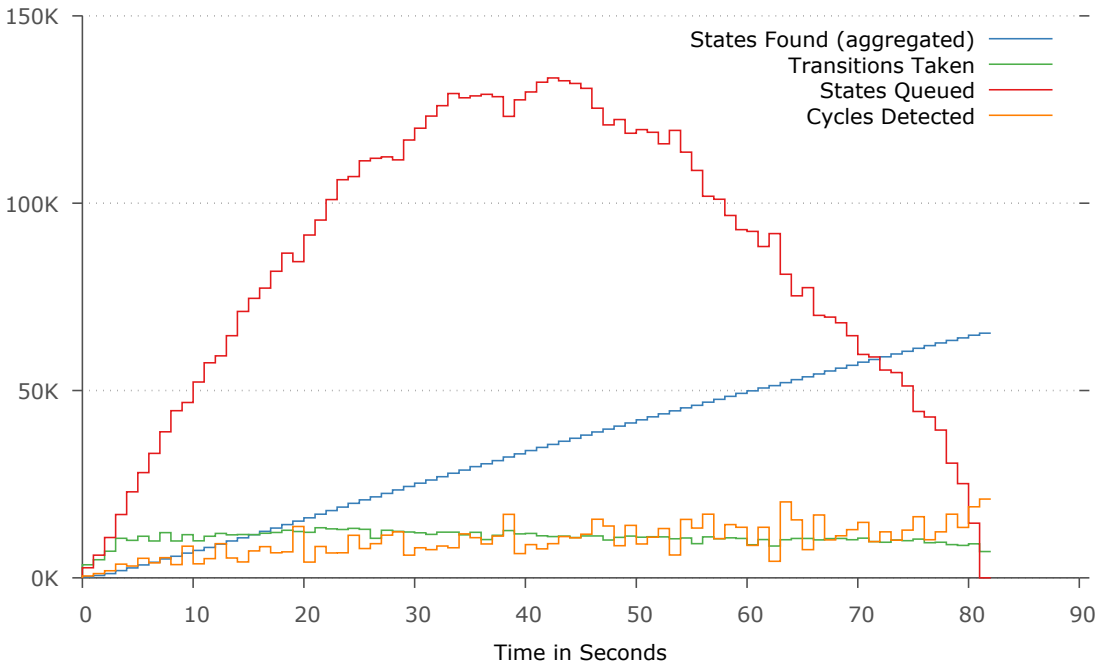


Figure 7.4.: Analysis of the transformation of the Nvidia Shield SCXML document onto a state-machine.

Figure 7.4 illustrates the development of the most important metrics when establishing the set of all global states $\widetilde{\mathcal{S}}$ and the related global transitions $\widetilde{t}_{i,j}$ for every global state $\widetilde{s}(i) \in \widetilde{\mathcal{S}}$ as part of the transformation. We can see that the algorithm took a bit more than 80 seconds to traverse the reachable global state space.

Following the approach in algorithm 2 and starting with the initial configuration, global states reached via potentially optimal transition sets as their global transitions are enqueued in a FIFO and dequeued until all global states have been found. The number of global states enqueued in any given second is plotted as the red line. This number is, at times, larger than the upper bound of global states as cycles are only detected when the global state is dequeued. The number of these cycles detected per second, wherein a global transition leads back to a global state already in $\widetilde{\mathcal{S}}$ is plotted as the orange line. Whenever a global state is dequeued, its global transitions are identified, taken and their global target state enqueued. This number of global transitions processed every second is plotted as the green line.

Finally, and most importantly, the blue line in figure 7.4 show the development of $|\widetilde{\mathcal{S}}|$ as the total number of global states already found. It is rather peculiar that this number increases virtually linearly throughout the complete global state traversal. If we disregard the first and last measurement, ~800 global states are found per second on average, with a standard deviation of only ~116. While this behavior is difficult to generalize to any SCXML document, it would provide a reliable indication about the maximum remaining time of the transformation along with the upper bound $|\widetilde{\mathcal{S}}|_u$ from equation 5.29.

7.1.3 Reducing the Size of the PROMELA Program

The final state-machine for the Nvidia Shield SCXML document has 65,297 global states connected by 878,723 global transitions. If we assume (i) a size for a global transition's executable content representation in PROMELA to be around 300 bytes and (ii) the respective conditional when dispatching global transitions per state to be around 200 bytes (compare listing 6.2), we end up with a PROMELA program of ~440 MB (after pre-processing to replace macro names). Initial attempts to employ SPIN to create C source files for the actual model-checker with these input files were unsuccessful and exhausted available memory.

The creation of global states for the state-machine suffers from a potentially double exponential explosion of states from the original state-chart. The first being caused by flattening the state hierarchy into a state-machine, not unlike the increase in number of states when creating a DFA from an NFA. The second exponential explosion is due to the number of global states required to encode history assignments. This second explosion can be avoided by making the history assignments explicit again.

History assignments influence global transitions by potentially changing their target configuration and the executable content to be executed. They will, however, leave the same global state under the same conditions with

regard to their guard condition and the event name that enabled them. Therefore, if we model history assignments explicitly in PROMELA via an array of booleans for every child or descendant state (depending on the history's `type`), we can use this array to conditionalize a global transition's executable content and target state. This allows us to drop all global states that differ only in their history assignments for a reduction by a factor of `168` to `464` in the Nvidia Shield case. To this effect, we will build two graphs of global states in algorithm 2 concurrently: (i) the graph of global states as introduced in section 5.3 and (ii) the graph of active global states, wherein two nodes are equivalent if they only differ in their history assignments.

Whenever we dequeue a global state $\widetilde{s}(j)$ in algorithm 2, whose active state configuration $\widetilde{s}_a(j)$ has already been visited in $\widetilde{s}(i)$, we copy the original global transition set $\widetilde{\mathcal{T}}(i)$ as the *global base transition set* into $\widetilde{\mathcal{T}}(j)$ as its *global history transition set* and establish a reference. This will cause all global base transitions, leaving the first global state with a given active configuration to know each of their variants caused by different history assignments.

Now, we can write the PROMELA state-machine, using only the global states with distinct active configurations and select executable content and global target states for the global transitions depending on the assignment of the history arrays of boolean. The following listing illustrates the approach:

```
1  <scxml datamodel="promela">
2    <state id="s0">
3      <transition priority="2" target="s1.h0"/>
4    </state>
5
6    <state id="s1">
7      <onentry><log label="Entering s1"/></onentry>
8      <history id="s1.h0">
9        <transition target="s1.s0">
10         <log label="History Transition s1.h0"/>
11       </transition>
12     </history>
13     <state id="s1.s0">
14       <onentry><log label="Entering s1.s0"/></onentry>
15       <transition priority="1" target="s1.s1"/>
16     </state>
17     <state id="s1.s1">
18       <onentry><log label="Entering s1.s1"/></onentry>
19       <transition priority="0" target="s0"/>
20     </state>
21   </state>
22 </scxml>
```

Spontaneously enter the compound state `s1` via its history `s1.h0`.

Transition is only taken on initial entry with empty history assignments for `s1.h0`.

Spontaneous transition to `s1.s1` and exiting compound state for `s0` to reenter via history `s1.h0`. Subsequent entries will skip `s1.s0` as history assignment is set to `s1.s1`.

Listing 7.1: State-chart with executable content and target depending on history.

The state-machine will start in `s0` and immediately transition to the history pseudo-state `s1.h0`. With no prior history assignment for `s1.h0`, the history's transition to `s1.s0` is taken and the respective executable content processed. In `s1.s0`, a spontaneous transition to `s1.s1` is taken, where the compound state is then exited for `s0` again. Upon exiting the compound state, the history pseudo-state `s1.h0` is assigned the history containing `s1.s1`, causing subsequent entries via `s1.h0` to enter `s1.s1` directly, skipping the executable content in the history's transition and for entering `s1.s0`.

The state-machine exemplifies how history assignments can alter executable content and the target state. With the parallel construction of the graph of active global states described above, we can think of the executable content and target per global transition to depend on the relevant history assignments.

Source	History	Executable Content	Target
`s0`	`{}`	l_7, l_{10}, l_{13}	`s1, s1.s0`
	`s1.h0:{s1.s1}`	l_7, l_{17}	`s1, s1.s1`

Table 7.1.: Executable content and target for transition from `s0` depending on history assignments.

Now, we want the most compact representation of blocks of statements representing executable content, conditionalized by the history assignments to account for every variation. An optimal solution to this problem is difficult to attain, maybe even NP-hard as it seems to contain the *longest common subsequence problem* for M sequences [Mai78] (M being the number of variations due to history assignments). Therefore, another approach with a more verbose solution but better runtime characteristics is employed:

We will only iterate the executable content for the very first variation of a global transition that originates in an active global state as the *base variation*. All other variations of this global transition due to different history assignments are expressed and conditionalized relative to this base variation. To this effect, we categorize all executable content for all variations in a sorted set $\widetilde{\mathcal{X}}(\tilde{t})$ containing three classes of sets, namely:

- $\widetilde{\mathcal{X}}(H)_{but}$ containing executable content to be processed for every variation of the global transition excluding those with history assignments given in H,
- $\widetilde{\mathcal{X}}(H)_{only}$ containing executable content specific to the history assignments given in H and
- $\widetilde{\mathcal{X}}_{every}$ with executable content to be processed by all variations.

We start with the first element of executable content in the base transition. If this element is common to all variations, we establish a set $\widetilde{\mathcal{X}}_{every}$ with the respective executable content and insert it as the first item in $\widetilde{\mathcal{X}}(\tilde{t})$. If there is a transition variation for which the element differs and the element is not contained as a subsequent item in the variation, we will establish a set $\widetilde{\mathcal{X}}(H)_{but}$ with the variation's history assignments and the element form the base transition. If the element is contained later in the list of executable content for the variation, we establish a set $\widetilde{\mathcal{X}}(H)_{only}$ containing all executable content up to the point of the element from the base variation. We continue this process until all elements of executable content from the base variation are exhausted and append all remaining elements from the other variations as individual $\widetilde{\mathcal{X}}(H)_{only}$ sets to $\widetilde{\mathcal{X}}(\tilde{t})$. For the example in listing 7.1, the construction is as follows:

Variation	**Executable Content** $\mathcal{X}(\widetilde{t})$	**Conditionalized Executable Content** $\widetilde{\mathcal{X}}(\tilde{t})$		
		$\widetilde{\mathcal{X}}_{every}$	$\widetilde{\mathcal{X}}(H)_{but}$	$\widetilde{\mathcal{X}}(H)_{only}$
`base`	$\underline{l_7}$, l_{10}, l_{13}			
`s1.h0:{s1.s1}`	$\underline{l_7}$, l_{17}			
Add l_7 for every variation		l_7		
`base`	$\underline{l_{10}}$, l_{13}			
`s1.h0:{s1.s1}`	$\underline{l_{17}}$			
Add l_{10} for all but `s1.h0:{s1.s1}`			`s1.h0:{s1.s1}`: l_{10}	
`base`	$\underline{l_{13}}$			
`s1.h0:{s1.s1}`	$\underline{l_{17}}$			
Add l_{13} for all but `s1.h0:{s1.s1}`			`s1.h0:{s1.s1}`: l_{13}	
`base`				
`s1.h0:{s1.s1}`	$\underline{l_{17}}$			
Base transition exhausted, add rest as history specific				`s1.h0:{s1.s1}`: l_{17}

Table 7.2.: Executable content and target for transition from `s0` depending on history assignments.

```
1  ...
2  /* history assignments for s1.h0
3     0: s1.s0
4     1: s1.s1
5  */
6  bool _hist_s1_h0[2];
7  ...
8  t1: /* from state: s0
9   ----- on event: SPONTANEOUS --
10        to state: s1, s1.s0 with no history
11                  s1, s1.s1 with history:{s1.h0:{s1.s1}} */
12 d_step {
13   printf("Entering s1");                          | Executable content l7 common to all variations.
14   if
15   :: (_hist_s1_h0[1]) -> skip;                    | Skip following block if history s1.h0 contains s1.s1
16   :: else -> {
17     printf("History Transition s1.h0");           | Executable content l10, l13 for all transition variations
18     printf("Entering s1.s0");                     | but s1.h0:{s1.s1}.
19   }
20   fi
21   if
22   :: (_hist_s1_h0[1]) -> {                        | Executable content l17 only processed when s1.h0
23     printf("Entering s1.s1");                     | contains s1.s1, skip block otherwise
24   }
25   :: else -> skip;
26   fi;
27   if
28   :: (_hist_s1_h0[1]) -> s = s2; /* to s1, s1.s1 */   | Select target state depending on history assignments,
29   :: else             -> s = s1; /* to s1, s1.s0 */   | defaults to target of base transition.
30   fi;
31 }
32 goto microStep;
33 ...
```

Listing 7.2: Executable content and target state conditionalized by history assignments.

By conditionalizing executable content in $\widetilde{\mathcal{X}}(\tilde{t})$ with regard to the history assignments, we can now generate a more compact sequence of statements to account for the various variations of global transitions. Elements contained in a $\widetilde{\mathcal{X}}_{every}$ set are written without any conditional, elements from a $\widetilde{\mathcal{X}}(H)_{but}$ are excluded via a conditional that skips a following sequence of statements for the given history assignments and elements in an $\widetilde{\mathcal{X}}(H)_{only}$ set are only processed for the given history assignment. Listing 7.2 shows an excerpt of the transformed state-chart from listing 7.1. Displayed is the global transition from `s0` to the history pseudo-state `s1.h0` containing the original transition with priority 2 (line 3 in listing 7.1). One optimization is also shown, wherein consecutive elements of $\widetilde{\mathcal{X}}(\tilde{t})$ with equal classes and conditionalized on the same history assignments are aggregated into a single conditionalized block (line 17-18). After all executable content is conditionalized appropriately, the transition will finally select the target state and continue with the next micro-step.

Encoding the history assignments not in the state-machine's states but as described results in a PROMELA program with a size of only ~`2.5` MB after preprocessing.

7.1.4 Validating Temporal System Properties

With the SCXML document as a PROMELA program, we can finally verify temporal properties for the system described. The properties are given as named LTL expressions and appended to the PROMELA program, e.g.:

```
...
ltl name1 {
  always(eventually(s == s1))        |   The system will, ever again, enter state s1.
}
```

Listing 7.3: Any number of named LTL expression can be appended to the PROMELA program.

The overall process to arrive at an executable binary for a verification is depicted in figure 7.5 and consists of four individual steps (duration are for the Nvidia Shield SCXML document with an Intel Core i7-5557U CPU at 3.1Ghz):

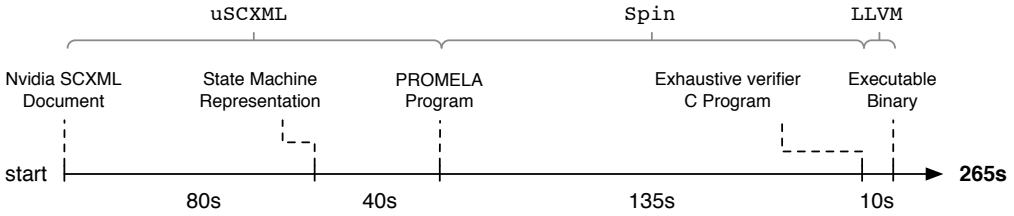


Figure 7.5.: Timeline of the transformation steps from the Nvidia Shield SCXML document onto an executable binary for an exhaustive verification.

1. The SCXML document is transformed, according to the construction described in section 5.3 onto an SCXML state-machine. With the adaptations to the construction outlined in section 7.1.2 above, this step takes around 80 seconds. The state-machine is never explicitly written as an SCXML document, but remains in memory only for the next step.

2. The state-machine representation is transformed into a proper PROMELA program, using the construction described in section 6.4 and, again, the adaptations described in section 7.1.3 above. This step takes around 40 seconds and results in an actual PROMELA program, which is written to the file system.

   If no LTL expressions to verify were given in XML comments within the original SCXML document (compare section 6.4.11.3), the author is expected to append them before progressing with the next step (listing 7.3).

3. Now, we can employ the SPIN interpreter to create a C program from the system's description in PROMELA for an exhaustive search of the system's state space, which takes around 135 seconds.

4. Subsequently, the C program created by the SPIN interpreter is compiled via LLVM[2] into an executable binary that, if executed, performs an exhaustive search of the system ultimately described in the original SCXML document.

[2] `http://llvm.org` (accessed October 28th, 2015)

The resulting C program has a size of ~2 MB and compilation time is very sensitive to the employed compilation parameters. E.g. instructing the compiler to optimize the program (`-On`) will considerably increase compilation time. The duration of 10s depicted in figure 7.5 was achieved by omitting all such optimizations.

The executable binary created at the end of this process contains all named LTL expressions given in the XML comments or appended to the intermediate PROMELA program. As such, this process is only performed once per SCXML document and the specific LTL expression to verify can be specified by name when executing the binary. This allows to parallelize the verification of each individual LTL expression once this executable binary exists by merely running the program on multiple computers (figure 7.6).

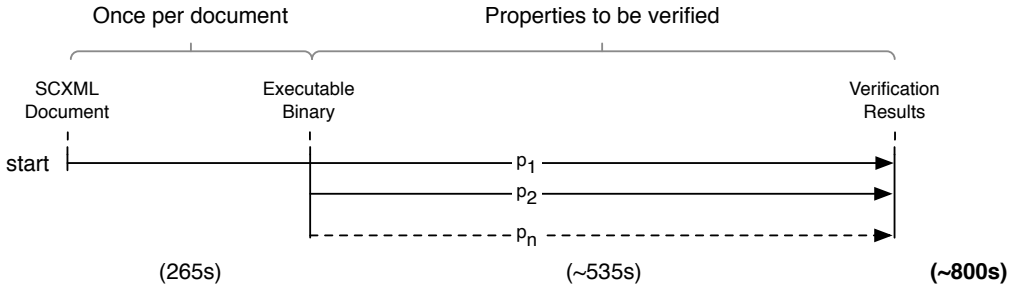


Figure 7.6.: Timeline of the complete process to arrive at verification results; an arbitrary number of system properties can be verified independently once the exhaustive verifier is compiled (Time in brackets for Nvidia Shield SCXML document).

The duration of ~535 seconds in figure 7.6 is measured for a verification without any LTL expression. Omitting such an expression during an exhaustive search will, nevertheless, cause the binary to traverse the complete state space (e.g. for PROMELA `assert` statements) and, as such, constitutes a lower bound for any successful verification. If, however, a temporal property given in an LTL expression is invalidated during verification, the process will stop when detecting the first invalid execution trace and print the sequence of events that caused the temporal property to fail. As such, the invalidation of a temporal property can be considerably faster, depending on the point in time during the traversal of the state space when the invalid execution trace is detected.

7.2 The uSCXML Interpreter Platform

The `uSCXML` interpreter is a standard-compliant implementation of SCXML written, predominantly, in C++98[3] and served as the platform to implement and evaluate all respective aspects of this thesis. It is available under a free open-source license and actually deployed, e.g. as part of a commercial system for ambient assisted living and smart homes [SWRAS15]. Even though the implementation itself is no scientific contribution, its central role in enabling the actual contribution of this thesis warrants an introduction.

The development of `uSCXML` started in September, 2012 with the SCXML specification still in working draft status and its results of the SCXML tests were officially submitted as part of the Implementation Report Plan (IRP) in June, 2014 - ultimately passing all tests for the ECMAScript data-model. At the time of this writing, `cloc`[4] reports its distribution of employed languages and lines of code, after ignoring generated files, as listed in table 7.3. It will compile for and run on Windows, Linux, MacOSX and iOS, both as 32 and 64 bit versions each. A port for the Android mobile platform was attempted in the early stages of development but postponed due to problems with compiling some dependent libraries, though, there is nothing inherent to the code-base preventing such an attempt. Language bindings for most of the functionality of `uSCXML` are available for Java, C#, Python, Perl and PHP in descending level of maturity via interface files provided for the `swig`[5] interface generator.

The `uSCXML` platform extends the SCXML recommendation via a wide range of (i) invokers, (ii) custom executable content XML elements, (iii) I/O processors and (iv) data-models as embedded scripting languagesfor application-specific instantiation of state-charts (compare section 4.2). The following sections will give an overview of each class and briefly discuss their role to provide interaction in general and an integration of MCs from the W3C Multimodal Architecture and Interfaces (W3C MMI architecture) recommendation in particular (compare section 3.1.3). Here, it is important to note that SCXML as a recommendation has no notion of an MC, it is only the W3C MMI architecture

[3] ISO/IEC 14882:1998

[4] `http://cloc.sourceforge.net` (accessed September 16th, 2015)

[5] `http://www.swig.org` (accessed September 16th, 2015)

Language	Files	Blank	Comment	Code
C++	131	7830	5591	37980
C/C++ Header	122	2947	3725	10745
XML	80	337	424	4435
Javascript	6	525	288	3560
HTML	7	508	75	3387
CMake	12	411	338	2353
Java	23	354	176	1694
C	3	84	265	604
C#	8	80	51	519
Visual Basic	2	24	29	211
Protocol Buffers	12	26	0	187
Objective C++	2	36	38	160
CSS	1	0	1	156
lex	1	32	3	82
Bourne Again Shell	2	11	11	66
PHP	1	11	8	66
MSBuild script	1	0	7	59
Prolog	1	5	13	20
Perl	1	4	13	18
YAML	1	1	0	15
JSON	1	0	0	10
SUM:	418	13226	11056	66327

Table 7.3.: General language statistics of `uSCXML` as reported by `cloc`.

that identifies SCXML as a markup language applicable to realize the responsibilities of an Interaction Manager (IM), which necessarily implies the responsibility to instantiate and control MCs. However, the concrete representation of a MC within SCXML is undefined and we will discuss the applicability and possible variations of the different extension mechanisms in this regard.

For the `uSCXML` interpreter, all extensions can also be provided from within the runtimes of target language bindings, which allows for a very flexible and expressive approach to application-specific instantiation of an SCXML document. E.g. in a deployment of `uSCXML` for multimodal dialogs in a smart-home environment, the existing Java code-base is integrated via a special I/O processor provided from within the Java runtime.

Not all of the extensions are approachable with formal verification as some are unusable with the `promela` data-model, e.g. due to their extensive use of DOM operations or the requirement to work with binary data.

7.2.1 Data-Models

In SCXML terminology, data-models are scripting languages embedded in a SCXML document and available e.g. to guard transitions via boolean expressions. Within the W3C SCXML recommendation, only the trivial `null` data-model is mandatory and one normative description for the `ecmascript` data-model is specified. Earlier versions of the SCXML standard also contained a normative description of an `xpath` data-model, which was later removed due to insufficient implementation reports. Many current SCXML interpreters will already provide additional data-models, more suited for their targeted platforms. E.g. the SCXML interpreter from Nvidia [KN14a] features a Lua data-model for a reduced memory footprint on their handheld gaming console.

The SCXML interpreter in our `uSCXML` platform currently features six different data-models (see comparison in table 7.4), selectable by providing a respective `datamodel` attribute in the topmost `<scxml>` element of a document. The set of applicable W3C IRP tests passed per data-model (compare section 5.1) is given in brackets:

- The **NULL** data-model (passes 1 of 1 tests)

  The null data-model is not the complete absence of functionality as it is required (as all others) to provide the `In('stateID')` predicate to check whether the given state is in the currently active configuration. The correct implementation of this predicate is the only functionality tested by the applicable IRP test.

- The **ECMAScript** data-model (passes 202 of 202 tests)

  The ECMAScript data-model is the most actively used, most mature data-model and features the rich expressiveness of the ECMAScript language along with some very useful extensions which we will discuss in more detail below. It is fully standards compliant and passes all tests.

- The **XPath** data-model (passes 107 of 211 tests)

  The SCXML IRP still provides some tests specific to the XPath data-model as there used to be a normative description within the specification. While it is still supported and occasionally used, it is a rather minimal implementation that still fails 104 of the 211 tests.

- The **Lua** data-model (passes 156 of 201 tests)

  Lua is an interesting scripting language to embed as its memory footprint and binary size are remarkably small while its syntax and semantics are very similar to ECMAScript. It is intended to run on rather constrained devices with a binary size of just 300kB for a static library containing the complete interpreter.

  There are no IRP tests provided for the Lua data-model, but Gavin Kirstner of Nvidia did transform some of the tests in a manual approach for the Lua data-model. They have no official status but can still provide a measure for the maturity of an implementation. For those, our implementation still fails 45 of 201 tests.

- The **Prolog** data-model (no IRP tests are transformed)

  To our knowledge, `uSCXML` is the only SCXML interpreter to offer a Prolog data-model and we already introduced it as a possible approach to extend SCXML for rule-based dialog modeling in section 4.5.1. Our implementation merely wraps an SWI Prolog[6] interpreter as a very mature, very established implementation of Prolog. This allows us to benefit from all modules and extensions already written for the SWI platform, e.g. JSON and XML parsers and the plethora of other libraries.

  We have yet to write a complete XSLT transformation description for a Prolog specific instantiation of the IRP tests.

- The **PROMELA** data-model (passes 171 of 182 tests)

  This data-model was discussed at length in section 6.3 as it is instrumental to the central contribution of this thesis. In the context of the other data-models, it is arguably the least expressive with the exception of the null data-model but still passes 171 of 182 applicable IRP tests (compare section 6.3.4).

In violation to the spirit of the SCXML recommendation, our platform allows application developers to register custom `DataModelExtension`s as callback functions which are available to expose variables of the embedding system directly in the data-model. This allows for a tight and convenient integration with existing application-specific data by simply exposing it in the data-model, but violates the implied assumption that all state changes (even those of the data-model) originate in events.

Name	Compound Events	Exposed SCXML DOM	Binary Data	Per Document Storage
`null`	No	No	No	No
`ecmascript`	Yes	Yes	Yes	Yes
`xpath`	Yes	No	No	No
`lua`	Yes	No	No	No
`prolog`	Yes	No	Yes	No
`promela`	Yes	No	No	No

Table 7.4.: Comparison of the data-model features implemented in `uSCXML`.

In principal, it is possible to integrate MCs from the W3C MMI architecture via a representation in the various data-models, e.g. by exposing them as objects. However, this is awkward to do with data-models that lack sufficient expressiveness and, again, violates the assumption that all state changes originate in events.

7.2.1.1 The ECMAScript Data-Model

In this section, we will discuss the ECMAScript data-model in some more detail as it is the most most mature, most feature-rich option provided in `uSCXML`. There are actually three variations for the ECMAScript data-model implemented:

[6] `http://www.swi-prolog.org` (accessed October 19th, 2015)

1. An implementation via **Google's V8** engine[7].

   The V8 implementation is used within the Chrome browser and offers good platform independence at excellent speed with some heightened developer effort to embed in a C++ library.

2. An implementation via **Apple's JavaScriptCore**[8].

   JavaScriptCore is the implementation used in the WebKit family of Hypertext Markup Language (HTML) browsers and also offers good platform independence at reasonable speed with moderate developer effort for embedding.

3. A proof-of-concept implementation using **Mozilla's Rhino** runtime in Java[9].

   This is not actually an implementation that is selected per default on any platform but merely served to drive the API requirements for data-models behind the various language bindings for `uSCXML` (here Java).

Currently, the JavaScriptCore implementation is the default on all Mac OS X and iOS platforms as well as Raspberry Pi with V8 being the default implementation on all Windows and Linux variants. Both these variations are fully standards compliant and offer the exact same extensions:

- *Complete SCXML DOM available via a DOM Core Level 2 API and XPath 1.0:*

  When using the `ecmascript` data-model, the complete SCXML DOM is available in the `document` variable and offers the popular DOM Core Level 2 API along with XPath 1.0 known from HTML browsers to adapt the SCXML DOM at runtime. This, essentially, enables a technique one could call *dynamic SCXML*, as an analogy to dynamic HTML (DHTML) which is at the core of every modern, interactive HTML document. With DHTML, the document can be adapted in response to various events, e.g. user input or replies for asynchronous `XMLHTTPRequests` and its graphical representation is updated accordingly. The same approach is available for SCXML documents on our platform.

  Junger elaborated on the principal approach [Jun14] in the context of his JSSCXML[10] platform running in the ECMAScript environment of an HTML browser. It is unclear if this technique will eventually be beneficial, for the moment it remains an academic exercise as more experience is needed to develop idioms and best practices.

- *Binary Data handling conforming to the TypedArray specification:*

  ECMAScript, as a language specification, does not provide any affordances to work with binary data. In the context of WebGL, the TypedArray API[11] was developed to remedy this short-coming by offering `DataViews` as interpretations of binary data contained in `ArrayBuffer` instances. Our platform does offer this API in the ECMAScript data-model, which is useful to coordinate invokers and other extensions that work with binary data, e.g. attachments to an email or multi-media content. While such binary content could also always be represented via an URL as a handle, having access to the binary representation allows for more flexibility and avoids life-cycle specific issues for binary data only referred to by URLs.

- *Storage of local data conforming to the WebStorage specification:*

  In the context of HTML5, several APIs for client-side storage were proposed, e.g. Web Storage, Web SQL, Indexed Database API or the FileSystem API to replace the older cookie-based approaches. The ECMAScript data-model in `uSCXML` implements the Web Storage API[12] as a pragmatic approach to persist arbitrary key/value pairs per document on a host system.

The bindings for these extensions are actually generated from a formal description of their API given in Web Interface Description Language (Web IDL) via a Perl script adapted from the Chrome browser distribution. This ensures that, e.g. the DOM Core Level 2 API implementation for V8 and JavaScriptCore features the exact same methods and signatures as any compliant implementation in a HTML browser.

[7] `https://code.google.com/p/V8/` (accessed October 20th, 2015)
[8] `http://trac.webkit.org/wiki/JavaScriptCore` (accessed October 20th, 2015)
[9] `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino` (accessed October 20th, 2015)
[10] `http://jsscxml.org` (accessed October 20th, 2015)
[11] `https://www.khronos.org/registry/typedarray/specs/latest/` (accessed October 20th, 2015)
[12] `https://html.spec.whatwg.org/multipage/webstorage.html` (accessed October 20th, 2015)

7.2.2 Invokers

In SCXML, invokers are software components available to control the life-cycle of remote entities and access their functionality from within a SCXML document. In the context of multimodal interaction adhering to the W3C MMI architecture recommendation, invokers are a candidate to realize the MCs, with the SCXML document constituting a description of an IM (compare section 3.1.3). The only invoker mandated by the SCXML specification is the `scxml` invoker to instantiate and control nested SCXML state-charts.

Our platform implements a wide range of invokers, some developed to realize requirements that arose in the context of projects, others for their perceived usefulness. The following list gives a brief overview of the invokers available in `uSCXML` before we detail a selection to illustrate different approaches for integration.

- Invokers for Markup Language Interpreters
  1. The **SCXML** Invoker (`type="scxml"`)
     - *Integration of nested state-charts*

     The SCXML invoker in `uSCXML` is merely a standard-compliant invoker for nested SCXML state-charts. It will instantiate a new interpreter, setup communication and initialize the parameters passed by the parent interpreter. The standard mandates for individual instances to be completely isolated, with only events available for synchronization. From within an invoked SCXML runtime, the invoking interpreter can be addressed via the special `_parent` target in a `<send>` element.
  2. The **VoiceXML** Invoker (`type="voicexml"`)
     - *Integration of voice user interfaces*

     Our Voice eXtensible Markup Language (VoiceXML) invoker allows for a tight integration of VoiceXML markup with SCXML via a fine-grained event interface to control and get notifications from a VoiceXML interpreter platform. We will discuss this invoker in more detail below.
  3. The **XHTML** Invoker (`type="xhtml"`)
     - *Integration of graphical user interfaces*

     The Extensible HyperText Markup Language (XHTML) invoker is available to instantiate and control HTML browsers on the host system. With HTML being the undisputed standard for visual content and interaction in the World Wide Web, integration with this technology is of elevated importance, which warrants the more detailed discussion found below. It is required to represent any HTML content as its proper XML dialect XHTML to parse and embed fragments in SCXML documents.
- Invokers for Multimedia Content
  1. The **3D Scenegraph** Invoker (`type="scenegraph"`)
     - *Render and control three-dimensional scenes*

     This invoker wraps the functionality found in the `OpenSceneGraph`[13] project and is available to control and display three-dimensional scenes. It is noteworthy for its unique approach to integration: The invoker is provided with a document conforming to a custom XML dialect to describe a three-dimensional scene. This document's object model is shared by the invoker and the SCXML runtime. Manipulations of the scene are subsequently performed via DOM Core Level 2 API operations from within the SCXML interpreter runtime.

     Ultimately, the custom markup ought to be replaced by an established XML standard for three-dimensional graphics, e.g. X3D[14], but no suitable implementations were available when we initially evaluated the different approaches.
  2. The **3D Model Conversion** Invoker (`type="model-convert"`)
     - *Convert three-dimensional models into various formats*

[13] `http://www.openscenegraph.org` (accessed October 20th, 2015)

[14] `http://www.web3d.org/x3d/what-x3d` (accessed October 20th, 2015)

This invoker is available to convert various three-dimensional formats into other three-dimensional and even two-dimensional formats, e.g. as screenshots of a scene in a given pose. It will take a three-dimensional model or a complete scene in a file specified via events and asynchronously process it into a specified output format, which is written to a file or delivered as an event with binary data attached.

3. The **Movie** Invoker (`type="movie"`)

   – Compose multiple images into a movie

   The movie invoker wraps the `FFmpeg` library[15] to render movies from a series of screenshots attached to events sent to an instance of this invoker. It integrates with the model conversion invoker above to create movies from a series of screenshots, e.g. a 3D model with a rotating pose.

4. The **Spatial Audio** Invoker (`type="spatial-audio"`)

   – *Rendering of audio samples at a perceived location*

   This invoker provides the functionality of the OpenAL library[16] for rendering spatial audio. Currently, it will not support streaming of audio but only playback of static audio files. However, all functionality with regard to the spatial placement of audio sources and listeners is available.

- Invokers for Messaging

1. The **Instant Messaging** Invoker (`type="instant-messaging"`)

   – *Sending and receiving of instant messages*
   – *Presence and status information*

   The instant messaging invoker provides the means to send and receive messages and manage contacts via a wide range of protocols. It exposes the functionality of `libpurple`[17] which provides access to the instant messaging services of e.g. AIM, Jabber, MSN, Yahoo! and Facebook.

2. The **SMTP** Invoker (`type="smtp"`)

   – *Composing and sending of emails*

   The SMTP invoker provides the means to compose and send emails. It is a rather versatile invoker that supports multipart messages with arbitrary attachments. It uses the SMTP implementation found in `libcurl`[18].

3. The **IMAP** Invoker (`type="imap"`)

   – *Receiving of emails*
   – *Mailbox management*

   Just as the SMTP invoker is available to compose and send emails, the IMAP invoker allows to check for and receive emails. It does expose all the functionality defined in the respective RFC3501 via the implementation found in `libcurl`.

- Other Invokers

1. The **Directory Monitor** Invoker (`type="directory-monitor"`)

   – *Notifications for file system modifications*

   This invoker is available to monitor a directory structure for changes to its contents. It is invoked with the name of a directory on the local file system and will raise events whenever a file system operation for one of the files or directories contained occurred.

2. The **Publish/Subscribe** Invoker (`type="umundo"`)

   – *Communication with other entities on channels as opposed to addresses*

[15] `https://www.ffmpeg.org` (accessed October 20th, 2015)
[16] `https://www.openal.org` (accessed October 20th, 2015)
[17] `https://developer.pidgin.im` (accessed October 18th, 2015)
[18] `http://curl.haxx.se/libcurl/` (accessed October 19th, 2015)

This invoker exposes the functionality of our `uMundo`[19] distributed publish/subscribe middleware for SCXML documents. Invoking an instance expects a channel name as a parameter and will subscribe to the given channel. Sending events to an instance will serialize all of the events' data and publish a message to all subscribed entities within the same multicast domain. This might be other SCXML runtimes or just about any application that employs the `uMundo` platform.

It is a rather pragmatic approach to realize distributed user interfaces, employing Multicast DNS for discovery of other nodes and ZeroMQ[20] to abstract from raw socket programming. There is no hierarchical structure of nodes or more elaborated routing, but it proofed to be very suited for experimentation and user interface demonstrations.

3. The **iCalendar** Invoker (`type="icalendar"`)

   – *Integration with calendaring software for events*

   The iCalendar invoker is available to read and process calendar data in a RFC5545 (iCalendar) compliant format. It is essentially the implementation of `libical`[21] exposed as an invoker in the `uSCXML` platform. It will read a compliant file from a URL or process respective text data supplied within an SCXML document and raises the specified events at the according time.

4. The **Expect** Invoker (`type="expect"`)

   – *Generic integration for all command-line programs*

   The Expect invoker provides the functionality of the NIST's Expect library[22] for the `uSCXML` platform. It allows to automate any program that can be started and controlled on a textual console by emulating a user typing commands and matching the program's output via regular expressions.

5. The **Hearbeat** Invoker (`type="heartbeat"`)

   – *Periodically raise events on the external queue*

   The heartbeat invoker is merely a convenience invoker as its functionality is already available with SCXML native language features. If a periodic event source is needed, one can always model a cyclic `<transition>` in a `<parallel>` child state that will enqueue a delayed event. However, setting up such a periodic event source is rather verbose and non-obvious when compared to the invocation of an explicit heartbeat invoker at some parent state.

There are different approaches to integrate the functionality of external systems as invokers in an SCXML document. We can differentiate between 1. the initialization of the external system and 2. the synchronization of state between the system and the SCXML runtime. Depending on the invoked functionality and the approach taken for integration, the amount of information required for each aspect differs considerably (see table 7.5). Even within any given invoker, one can oftentimes trade semantic richness of one aspect for the other.

An important criterion when weighing the semantic richness of the initialization against the richness of subsequent events is the question about the granularity of control and integration that is required with an invoked component. From a puristic point of view, with SCXML as a notation for state transition systems, one might argue that an invoked component should not have any state at all, or at least that its state ought to be transparent to the SCXML document. This allows for the most fine-grained integration at the cost of expensive communication in terms of required band-width. However, in some circumstances, large aspects of the invoked component's state are uninteresting for integration into a multimodal application. If, for example, we invoke an HTML browser to merely show a picture, we would not need to have access to its DOM nor be aware of a pointing device's position on the browser's window. On the other hand, there might be situations were such information is indeed useful, e.g. to augment the interaction with such an image via other modalities.

In the following, we will detail three invokers in more detail for their different approaches at integration:

- The **VoiceXML invoker** allows to invoke a respective interpreter with a complete or partial VoiceXML document. It is noteworthy for its tight integration, despite being an external interpreter. This is possible as the specific implementation we employ implements the W3C MMI architecture MC interface with a rich set of events being communicated.

[19] `https://github.com/tklab-tud/umundo` (accessed 25th October, 2015)
[20] `http://zeromq.org` (accessed 25th October, 2015)
[21] `https://github.com/libical/libical` (accessed October 18th, 2015)
[22] `http://expect.nist.gov/` (accessed October 18th, 2015)

Initialization	Synchronization
Rich	
A complete XML DOM or, depending on the invoker, a compliant document is provided either in the invocation's `<content>` element or via the `src` attribute to setup the bulk of the invoker's initial state.	Events passed between the SCXML interpreter and the invoked component contain rich semantics and, potentially, change the complete state of the invoked component.
Light	
The invoker's state as setup by an initial document is very rudimentary and only prepares the component to accept subsequent events.	Information passed between the SCXML interpreter and the external system are only in the form of lightweight notifications and control messages.

Table 7.5.: Design dimensions with regard to the integration of invokers.

- The **XHTML invoker** provides three different approaches for invocation and control: (1) *invoke-and-forget* with a URL, (2) *invoke-and-forget* with dynamic XHTML content and (3) instantiation with a XHTML template for bi-directional communication with the SCXML runtime. Many of the design decision regarding this invoker were influenced by the fact that all modern HTML interpreters represent considerable technological investments. Here, integration ought to be tight but we want to treat such an HTML interpreter as unmodifiable because maintaining a branch of a major HTML browser is very expensive. Therefore, we want to work within their technological confines without any adaptations or extensions.
- The **3D Scenegraph invoker** showcases yet another approach for a very close integration. The three-dimensional scene is modeled in a custom XML dialect, exposed to the data-model and manipulated via a DOM Core Level 2 API, not unlike manipulating HTML via ECMAscript in the approach taken for dynamic HTML in browsers. This approach illustrates what an integration of a XHTML invoker might look like if we were willing to adapt a HTML browser implementation's source code.

7.2.2.1 The VoiceXML Invoker

Integration of VoiceXML interpreters with SCXML is of particular importance as SCXML is, in parts, an answer to the postponed ambitions for a third iteration of the VoiceXML recommendation. After the specification of VoiceXML 2.1, the W3C recommendation came under pressure, especially for its rather static approach to spoken dialog modeling. Herein, a static Form Interpretation Algorithm (FIA) would just prompt for fields of the current form in sequence, after an initial *mixed initiative* prompt allowed to fill many fields with a single utterance. In VoiceXML 3, an attempt was made to isolate the various responsibilities into modules and profiles. However, the approach proofed too ambitious for a W3C specification and the standard has seen no progress since December, 2010. Nevertheless, VoiceXML as a language for modeling Spoken Dialog System (SDS) was a success, most notably due to its deployments for Interactive Voice Response (IVR) systems in telephone call-centers. Thus, it is desirable to retain much of the VoiceXML 2.1 language features and embed them in a more flexible approach to dialog modeling, where SCXML as a modality agnostic dialog modeling language finds its role.

#	Invocation Options	Brief Description
1. Invocation with a VoiceXML document or fragment thereof		

	`<invoke src="..." type="voicexml">`
	`  <param name="target" expr="..." />`
	`  <content>`
	`  <vxml: ...>`
	`  </content>`
	`</invoke>`
		Invoking a VoiceXML interpreter with a complete document or a fragment will just instantiate a respective session with an interpreter at the given `target` URL. Currently, only the JVoiceXML interpreter[23] can be integrated with this approach as it is (to our knowledge) the only one to implement the MC interface from the W3C MMI architecture specification.

Table 7.6.: Invocation options for the `voicexml` invoker.

Our various approaches to adapt VoiceXML for the special requirements of pervasive environments can be found in various workshop contributions, conference proceedings and even journals [RSW12, SWRMua13] and eventually a book chapter [SWR15]. Ultimately, we settled upon the approach to embed VoiceXML 2.1 in our `uSCXML` platform as

an invoker that accepts VoiceXML markup or *meaningful fragments* thereof via its `<content>` element or per URL in the `<invoke>`'s `src` attribute. This allows a dialog author to either reuse complete VoiceXML documents or, more selectively, embed spoken utterances and automatic speech recognition for individual fields in a SCXML dialog model. The tables 7.6 - 7.8 detail the options for invoking a session with a VoiceXML interpreter platform and list the events accepted and raised.

The integration of the `voicexml` invoker does assume the remote interpreter platform to implement MC interface specified in the W3C MMI architecture (compare section 3.1.3). As such, an invocation will start by automatically issuing a `NewContextRequest` followed by a `StartRequest` with the document specified in the `<invoke>` element's `<content>` or `src` attribute as soon as the context is established. Processing of the VoiceXML document will begin immediately while the invoker remains available to accept the events listed in table 7.7. The W3C MMI architecture does not mandate any detailed structure nor does it define a set of application-specific events apart from the general life-cycle events. Here, the general `ExtensionNotification` message is intended as a container to wrap application-specific data exchanged during a session.

#	Events Accepted	Brief Description

1.	*Notify VoiceXML of the deprecation of a field*
	`<send target="#_vxml"`
	`event="vxml.input.[field]">`
	`<param name="field" expr="..." />`
	`</send>`
		A field, due to be queried for by the VoiceXML FIA, was supplied by other means, e.g. via another modality component.

2.	*Pulling the value of intermediate fields prior to completion.*
	`<send target="#_vxml"`
	`event="vxml.data.<identifier>" />`
		Query the VoiceXML interpreter for an arbitrary identifier in its ECMAScript runtime.

Table 7.7.: Events accepted by a running `voicexml` invoker.

There are only two classes of events that the remote VoiceXML session will accept:

1. A notification for the deprecation of a field given in a VoiceXML form. This is required to notify the remote FIA, that it is not necessary to prompt for the field any longer. This is the case if the information to be gathered in such a field has been provided via other means (e.g. via another modality) or, in the case of a rule-based approach, if it can be resolved from information already established.

2. Querying for the value of any given identifier in the VoiceXML interpreter's ECMAScript runtime. This is not as central as the ability to deprecate fields but still convenient. However, it is not intended to poll for individual form fields until they become available as these will be raised as dedicated events (see below).

It is noteworthy, that the processing of these events is still asynchronous, i.e., querying for an identifier will merely request the VoiceXML interpreter to deliver a respective event.

#	Events Raised	Brief Description
1.	*Status notifications*	
	`vxml.input.start`	The speech recognizer became active.
	`vxml.input.end`	The speech recognizer was deactivated.
	`vxml.input.speech.start`	The user started to speak (including barge-in).
	`vxml.record.start`	The VoiceXML interpreter started to record audio data.
	`vxml.record.end`	The recording of audio data stopped.
	`error.vxml.*`	One of the VoiceXML error events was raised in the VoiceXML interpreter.
	`vxml.output.start`	The rendering of a system prompt (e.g. via text-to-speech) was started.
	`vxml.output.end`	The rendering of a system prompt ended.
	`vxml.output.emptyqueue`	All the system's prompts were played.
2.	*Data input*	
	`vxml.input.<fieldid>`	A form field was successfully filled in.
	`vxml.input.nomatch`	The user's utterance did not match any active grammar.
	`vxml.input.noinput`	The user remained silent despite an open prompt.
	`vxml.input.help`	The user requested help.

Table 7.8.: Events raised by the `voicexml` invoker.

The set of events raised by the VoiceXML invoker is more exhaustive to provide a close integration with the SCXML interpreter (see table 7.8). Virtually all changes to the VoiceXML interpreter's state are communicated back to the SCXML interpreter.

When compared to other invokers for external systems, the `voicexml` is unique as we were willing and able to extend an established VoiceXML platform to implement the MC interface of the W3C MMI architecture. This allowed for a very fine-grained and largely automated integration into the notion of passing events as is emphasized with SCXML dialog models.

7.2.2.2 The XHTML Invoker

The approach taken for the integration of HTML browsers via the `xhtml` invoker is very different from the `voicexml` invoker. Here, our utmost design goal was to avoid any adaptations of the established HTML interpreter platforms in order to minimize maintenance cost and offer a seamless integration with the huge deployment base. As such, we had to work within the confines of the respective platforms and realize integration with the language features afforded by *modern* HTML. I.e. we could not adapt an implementation to conform to the MC interface with a set of application-specific events in `ExtensionNotification`s.

Nevertheless, HTML is indisputably the most important markup language to describe graphical interfaces in the World Wide Web. Its role is so prominent that some other SCXML implementations, indeed, employ the ECMAScript runtime of a HTML browser as a *host environment* and implement a SCXML interpreter within. But if we are to realize multimodal applications in pervasive environments, we should not start our ambitions with a greatly exaggerated role for graphical interaction. On that note, there is also an interesting trend to augment HTML with evermore functionality originally associated with other modalities. E.g. the Web Speech API[24] intends to make automatic speech recognition and speech synthesis available in HTML documents.

Regardless of whether it makes sense to embed SCXML as a modality agnostic dialog modeling language in a necessarily modality specific host environment, the importance of HTML as a markup language for graphical interfaces is undisputed. If SCXML is ever to succeed in any meaningful manner, it will have to provide a convincing integration. In the approach we took for integrating HTML documents via the `xhtml` invoker, there are three different options available (see also table 7.9):

#	Invocation Options	Brief Description
1.	*Invocation of a HTML browser with a given URL* `<invoke type="xhtml" src="..." />`	Providing a URL in the `src` attribute of the `<invoke>` element will simply instantiate the host system's default HTML browser with the document at the given URL.

2.	*Invocation of a HTML browser with dynamic inline content*
	`<invoke type="xhtml">`
	`  <content>`
	`  <any />`
	`  </content>`
	`</invoke>`
		If the `<content>` element contains any children, the SCXML interpreter will instantiate an HTTP server listening at the platform's `basichttp` port with a URL path containing the invocation's identifier. Subsequently, the system's default HTML browser is opened with this URL and the content contained will be delivered for the browsers request. A special `${expr}` syntax is available for all child nodes in `<content>` for evaluating expressions on the data-model. These are substituted during invocation as string literals or as their JSON representation.

3.	*Dynamic content and bi-directional communication with a HTML browser* `<invoke type="xhtml" />`	If the `<invoke>` element contains neither inline content nor a `src` attribute, the SCXML interpreter will also instantiate a HTTP server and start a browser but only deliver a template containing an HTML document with an empty `<body>` and some ECMAScript to establish a bi-directional communication via long-polling `XMLHTTPRequests`. Subsequently, content send to this invoker is dynamically added to the remote XML Document Object Model (DOM) with respect to the content's `type` and `xpath` attributes. Here, the `xpath` attribute identifies a node in the remote HTML browser's DOM and the `type` attribute identifies the operation (see below).

Table 7.9.: Options for invoking external HTML browsers

[24] `https://dvcs.w3.org/hg/speech-api/raw-file/tip/speechapi.html` (accessed October 20th, 2015)

1. An obvious but very limited approach is to merely start an HTML interpreter with a given URL. This will just open the host system's default HTML browser with no possibility to dynamically influence the HTML content that is displayed or to control the browser at all. This approach might be suited to display an HTML document with some information in the World Wide Web but is otherwise inapplicable to integrate with the SCXML runtime. There is not even an application for the `<param>` element or the `namelist` attribute to parametrize such an invocation with data-model variables.

2. An approach that allows for some amount of integration is to provide the `xhtml` invoker with dynamic HTML. The SCXML standard does provide the means to pass XML DOM nodes via the `<content>` element's `expr` attribute, but this requires considerable support from the data-model (e.g. a DOM Core Level 2 API as is found in our ECMAScript data-model). An alternative is to pass dynamic HTML as a simple, dynamically assembled string, but this is very tiresome for an author to do and difficult to maintain and adapt. Here, our `xhtml` invoker will allow to provide HTML in the invocation's `<content>` element with special `${expr}` tokens that are substituted by their evaluation in the data-model. This allows to insert dynamic text, JSON structures or even complete DOM sub-structures at any place in the HTML content if supported by the data-model. In both cases, the `uSCXML` runtime will register a specific path with an embedded HTTP server and start a HTML browser with the respective URL. This will cause the HTML browser to send a HTTP `GET` request into the `uSCXML` runtime, which is answered by the dynamic HTML provided with the `<invoke>` element. While this approach allows to invoke an HTML browser with dynamic HTML, it still provides no means to return any events back into the SCXML interpreter's runtime.

3. The third approach for integrating a HTML interpreter is the most versatile and does allow for bi-directional communication and dynamic adaptations to the DOM of the HTML interpreter throughout its lifetime. If the `xhtml` invoker is started without any content, again, a path with the embedded HTTP server is registered and a HTML browser started with the respective URL, but in this case, the initial reply will contain a template with an empty HTML `<body>` and some ECMAScript to establish bi-directional communication via long-polling `XMLHTTPRequests`.

   Subsequently, the `xhtml` invoker instance is available to `<send>` DOM nodes or complete DOM trees to the remote HTML browser to modify its local DOM. To this effect, we allow for two additional attributes with the `<content>` element in a `<send>` request, namely `type` and `xpath`. The `xpath` attribute is assumed to contain a XPath expression to identify a node in the remote HTML browser's DOM and defaults to `/html/body`. The `type` attribute specifies the operation to perform with the DOM contained in the `<content>` element in relation to the node identified by the XPath expression. The following types are supported:

   - A value of `firstchild` or `lastchild` will insert the DOM contained in the `<content>` element as the first or last child element in the node identified by the `xpath` expression respectively.
   - A type value of `previoussibling` or `nextsibling` will insert the DOM from the `<content>` element right before or after the node from the `xpath` expression.
   - The values of `replace` or `delete` have their obvious semantics, with the latter ignoring any DOM given in a local `<content>` element and merely deleting the node identified by the `xpath` expression.
   - Specifying a type of `replacechildren` will insert the DOM nodes from the `<content>` element in the remote node after any existing child nodes were removed.
   - Finally, the `addattribute` type is available to add an attribute to a remote DOM node. Here, the content is expected to be a literal string.

   The initial HTML template will also declare and initialize an ECMAScript object named `_parent` in the HTML browser, which manages all communication with the parent SCXML session and is available to return events. Listing 7.4 does provide an example for a dynamic adaptation and the returning of form data back to the SCXML interpreter. It does assume an active `xhtml` invoker running with an invocation identifier of `xhtmlid` and will cause the remote HTML browser to display a form containing two fields, its values returned to the parent SCXML session when submitted.

```
1  <send target="#_xhtmlid">
2    <content type="replacechildren" xpath="/html/body">
3      <html:p>Enter some details to continue!</html:p>
4        <html:form id="form1" onsubmit="
5              _parent.send('form.submitted',
6                           [document.forms['form1'].elements[0].value,
7                            document.forms['form1'].elements[1].value]);
8              return false;">
9          First name1: <html:input type="text" name="firstname" /><html:br />
10         Last name1: <html:input type="text" name="lastname" />
```

```
11          <html:input type="submit" value="Submit" />
12        </html:form>
13    </content>
14  </send>
```

Listing 7.4: Replacing the HTML <body> element's children with a form in a xhtml invoker.

Using this approach, an author can also add an HTML <script> element to the remote DOM containing ECMAScript to register callbacks for arbitrary events from the SCXML interpreter. This allows for just about any type of event to be send to the xhtml invoker (table 7.10) and processed in the HTML browser's ECMAScript runtime.

Likewise, the _parent.send method in invoked HTML browser's ECMAScript runtime will accept arbitrary event names as its first argument and any non-cyclic data-structure as the second event name, which will be serialized and delivered to the scxml interpreter's external queue (table 7.11).

#	Events Accepted	Brief Description

1.	*Adapting the remote DOM*
	`<send target="#_invokeid">`
	`  <content type="replacechildren"`
	`  xpath="/html/body">`
	`  <html:p>Thank you!</html:p>`
	`  </content>`
	`</send>`
		Sending an event without a name will assume the presence of a <content> element with a type and a xpath attribute. Such events are available to modify the remote HTML browser's <dom>.

2.	*Arbitrary events with optional data*
	`<send target="#_invokeid"`
	`  event="..."`
	`  namelist="...">`
	`  <param name="..." expr="..." />`
	`  <content>`
	`  <any />`
	`  </content>`
	`</invoke>`
		The remote HTML browser can be prepared by adding <script> elements to its DOM to accept and process arbitrary events. Such an event will merely be serialized into a JSON structure by the xhtml invoker and passed into a respective ECMAScript callback function registered with the _parent object in the remote HTML browser's ECMAScript runtime.

Table 7.10.: Events accepted by the xhtml invoker with the dynamic DOM approach.

#	Events Raised	Brief Description
1.	*Anything send via* _parent.send(name, data) *from the HTML browser* `_parent.send('event.name', data);`	The _parent.send() method established by the initial HTML template served for the first HTTP request from an invoked HTML browser allows to return arbitrary events back to the SCXML interpreter.

Table 7.11.: Events raised by the xhtml invoker in the dynamic DOM approach.

This last approach for integrating a xhtml invoker allows for considerable flexibility with asynchronous adaptations to the remote HTML browser's DOM and, essentially, enables a set of techniques popularized as DHTML.

If one were willing to adapt the code-base of an existing HTML browser, an approach more natural to developers already familiar with the techniques of dynamic HTML might be to integrate an HTML browser via the DOM Event I/O processor[25]. In this approach, the SCXML runtime would synchronize directly with an HTML document via DOM events[26] passed between the two documents. The dom I/O processor used to be contained as a normative description in the SCXML standard up until its recommendation candidate status, but was moved into a working group note due to missing implementation reports. It is undefined how issues of life-cycle management and initialization are to be resolved, the working group note only remarks that "an example [...] would be a document containing both SCXML and HTML markup" with no clarification about the structure of such a shared document and that "the SCXML processor must support sending DOM events to any node in the document". In our approach above, it is already possible to selectively synchronize via DOM events, as is exemplified by the onsubmit DOM event in listing 7.4. It would be an option to use this I/O processor to communicate with the DOM specified for the xhtml invoker via the

[25] http://www.w3.org/TR/scxml-dom-iop/ (accessed 26th October, 2015)

[26] http://www.w3.org/TR/DOM-Level-3-Events/ (accessed 26th October, 2015)

<content> element with the second approach described above. However, it requires a very tight integration between the runtimes of the SCXML interpreter and the HTML browser, which might not be possible without embedding in the same process.

7.2.2.3 The 3D Scenegraph Invoker

The scenegraph invoker allows to display three-dimensional scenes in multiple windows, optionally divided into viewports, on the host system. Essentially, it wraps the scenegraph implementation from OpenSceneGraph project for uSCXML. It is special among the invokers for its approach to integrate the scenegraph in the SCXML document via a shared DOM. The invoker is necessarily instantiated with a XML document containing custom markup to describe a set of scenes (see listing 7.5). Subsequently, an author would employ the W3C DOM Core API available in the ecmascript data-model to adapt the various scenes.

```
1  <invoke type="scenegraph">
2    <content>
3      <scenegraph:display>
4        <scenegraph:viewport>
5          <scenegraph:rotation>
6            <scenegraph:node src="scenegraph/model.wrl" />
7          </scenegraph:rotation>
8        </scenegraph:viewport>
9      </scenegraph:display>
10   </content>
11 </invoke>
12 ...
13 <state id="animate">
14   <onentry>
15     <script>
16       var nodeSet = document.evaluate("//scenegraph:rotation").asNodeSet();
17       nodeSet[0].setAttribute("yaw", "yaw" + yaw + "deg");
18       yaw += 0.1;
19     </script>
20   </onentry>
21 </state>
```

Listing 7.5: Example XML excerpt for displaying a rotating VRML file in a three-dimensional scene.

The invoker will not accept nor raise any events. Ultimately, it is very much desirable to have the invoker raise events for the various user interface events that occur whenever a user interacts with the scene.

#	Invocation Options	Brief Description

1.	*Invocation with a custom XML document*
	`<invoke type="scenegraph">`
	`<content src="...">`
	`<scenegraph: ...>`
	`</content>`
	`</invoke>`
		Invocation will necessarily require a XML document conforming to a custom dialect for describing three dimensional scenes which is subsequently modified via a DOM Core Level 2 API.

	Events Accepted	**None**
	Events Raised	**None**

Table 7.12.: Integration of the scenegraph invoker.

This approach at integration is similar to the discussed variation of the xhtml invoker with the dom I/O processor. In its current form, it was, foremost, a technical prototype to assess the viability of a shared DOM integration for an invoker. Here, we have to conclude that it requires a tight coupling between the SCXML runtime and the interpreter for the respective XML dialect. Together with the requirement for a DOM Core Level 2 or similar API available in the data-model, it requires considerably more development effort for integration when compared to a simple *"invoke with domain specific document and synchronize via events"* approach.

7.2.3 Custom Executable Content

The SCXML recommendation allows for the introduction of platform-specific executable content and our uSCXML platform does provide four such custom elements.

- The <fetch> Element

This element allows for the asynchronous retrieval of arbitrary data referenced per URL in its mandatory `src` attribute. It also requires a `callback` attribute to identify the name of the event that is to be raised on the interpreter's external queue once the data is available. The functionality of this element is comparable with the `XMLHTTPRequest` object found in modern HTML browsers and it provides the means for an SCXML document to *pull content into* the running session (compare section 4.6.2).

The functionality of this element can, in different variations, found in many more SCXML implementations, evidencing its usefulness. E.g. an implementation compliant with the `uSCXML` `<fetch>` element is also found in JSSCXML.

- The `<respond>` Element

  The SCXML recommendation does not describe any mechanism to reply to external HTTP requests. There is a `basichttp` I/O processor, but it will only receive HTTP encoded events and reply with `200/OK` or `500/Server Error` with no means to specify actual payload data.

  The `<respond>` element remedies this short-coming. In conjunction with the `http` I/O processor, this element allows for an SCXML document to reply to an HTTP request with arbitrary, dynamic data. The `http` I/O processor will, for every incoming HTTP request, raise a respective event on the external queue and the `<respond>` element can refer to the event's `origin` field to assemble and send a reply.

  Providing this functionality is important, as it provides the means to *pull content from* the running session (again, compare section 4.6.2).

- The `<file>` Element

  The functionality of the `<file>` element is comparable to the `<fetch>` element, but it also allows to write or append arbitrary data to a file on the local file system.

- The `<postpone>` Element

  This element allows to postpone an event until a certain condition, given in its `cond` attribute is met. If an event is postponed, the interpreter will check its condition at the end of a macro-step and reissue the event to the front of the external event queue once the condition evaluates to `true`.

  There are some situations where such functionality is beneficial, e.g. when the reply to an HTTP request needs to be postponed until some other processing is completed. However, its semantics proofed to be complicated as all effects performed by processing the event until postponing would be performed twice: Once until the `<postpone>` element is encountered and again when the event is reissued. This includes any eventual configuration changes to the interpreter, which are very complicated to *undo*.

Extending a platform via custom executable content might be another candidate to integrate MCs from the W3C MMI architecture. One could imagine, e.g. to provide a subset of VoiceXML as custom executable content and render a spoken prompt for any occurrence of a `<vxml:prompt>` element. This would allow for a very convenient and *natural* approach to multimodal dialogs as one would just mix XML namespaces. However, there would be no means to manage the life-cycle of an MC integrated in this manner and coordinating e.g. two distinct instances of an MC would be awkward (though still possible with distinct namespaces e.g. `<vxml1:prompt>` and `<vxml2:prompt>`).

7.2.4 Additional I/O Processors

Extending an SCXML interpreter platform with a custom I/O processor will, essentially, extend the set of legal values for the `<send>` element's `type` attribute. This class of extensions is limited in its usefulness because, just as with custom elements, an I/O processor's state is global per SCXML session. As such, it is useful for stateless input/output connections, such as HTTP but difficult to apply when multiple stateful connections need to be maintained and managed. It is for this reason that, e.g., the `umundo` publish/subscribe platform is integrated as an invoker and not an I/O processor as the subscription to channels can be expressed as invocations.

- **HTTP**

  The standard only provides a normative description for a `basichttp` I/O processor for which the `uSCXML` platform does provide a compliant implementation. However, the `basichttp` I/O processor is rather limited in its applicability: It is available to enqueue events with associated data on the interpreter's external queue, but will not allow to reply with anything other than `200/OK` for well-formed requests and `500/Server Error` for invalid requests.

The `http` I/O processor alleviates this short-coming. In conjunction with the `<respond>` element described above, it will allow to formulate replies containing arbitrary data, e.g. XHTML, JSON, plain text or even binary content with a given mime type. Every incoming HTTP requests on the I/O processor's URL is represented as an event `http.get` or `http.post` with all its headers and payload contained in the event's compound data structure. The event's `origin` field will contain a unique identified for the `<respond>` element to address in the reply. In out implementation, the I/O processor's URL takes the form of `http://<hostname>:<port>/[document name|session identifier|]` with the port specified when instantiating an embedded interpreter or given on the command-line with the `uscxml-browser` tool (see below). The path component of the URL constitutes of the SCXML session identifier from `_sessionid` defined per standard or, alternatively, the `name` of the SCXML document given in its topmost `<scxml>` element. For multiple instances with the same document name, a consecutive number is appended. This allows an external component to determine the URLs of all instances running a given SCXML document.

One problem with this I/O processor is the fact that every request received will necessarily require the processing of a corresponding `<respond>` element to formulate a reply. Otherwise the connection is kept lingering until the connecting client decides to close it due to a time-out. Here, it is the responsibility of the SCXML author to guarantee that every HTTP request received via the `http` I/O processor is eventually matched by a `<respond>` element.

Long-Polling HTTP

The problem with lingering connections can actually be very useful to realize a simple *server-push* idiom for clients that do not allow for incoming connections (e.g. the majority of HTML runtimes). Here, the client will maintain an asynchronous connection to the `http` invoker that is only replied to when the server wants information pushed into the client. It is this approach that allowed for the tight integration with the XHTML invoker.

- **WebSockets**

  The `websocket` I/O processor provides bi-directional communication with clients that implement the respective standard[27], most notably many modern HTML browsers. It is an alternative to long-polling HTTP requests and employs a much more stream-lined protocol with a dramatically reduced overhead per message (2 bytes with WebSockets versus ~800 bytes with a typical HTTP request[28]). This is especially of relevance if many small messages need to be transferred as is oftentimes the case when synchronizing user interaction events between an MC and its IM.

7.2.5 Integrating Modality Components in State Chart XML

The SCXML recommendation never refers to its suggested role as a IM in the W3C MMI architecture for the Multimodal Dialog System (MDS) proposed by the W3C Multimodal Interaction Working Group (W3C MMI WG). It is clear, however, that such a role was intended as e.g. Jim Barnett, Michael Bodell and T.V. Raman, are editors or authors of both, the W3C MMI architecture and the SCXML recommendation. We had the chance to speak with Jim Barnett about the integration of MCs into SCXML at our EICS SCXML workshop in Rome, 2014. He would confirm that it was indeed the extension mechanism of invokers in SCXML that was intended to realize the integration of MCs but admitted that more experience was needed to develop concrete manifestations of the approach.

To this effect, the previous sections already introduced the various extensions of the `uscxml` platform provides as (i) invokers, (ii) custom executable content, (iii) I/O processors and (iv) data-modelsand discussed their suitability to integrate a MC from the W3C MMI architecture. The tables 7.13 - 7.16 summarize the advantages and disadvantages of the different approaches. Here, we will not necessarily concern ourselves with a literal implementation of the MC interface, but a rather liberal interpretation wherein the same effect, to instantiate and control external components, can be achieved. Each approach is classified with regard to five general characteristics, namely:

1. Whether instantiation and communication is **asynchronous**. If the SCXML interpreter can progress without waiting for the employed approach to succeed or fail, we will grant asynchronicity.

2. Whether or not eventual **markup can remain intact**. This is important to reuse existing descriptions or to delegate the creation of such documents to respective experts. E.g. with HTML, it is generally desirable to have a web-designer author the document with familiar tools. This is very much dependent on the invoked functionality and a rather ambiguous classification.

[27] `http://www.w3.org/TR/websockets/` (accessed October 18th, 2015)

[28] `https://www.websocket.org/quantum.html` (accessed accessed October 18th, 2015)

3. Whether all **changes to the state** of the SCXML interpreter and the invoked component **originate in events**. This is generally a desirable property, as it ties into the core concepts of SCXML and guarantees that such a component can be distributed and synchronized via message containing these events.

4. Whether the approach will allow for **multiple instances** of the same component.

5. Whether the **life-cycle of instances of a component can be controlled**. That is, whether instances can be created and destroyed while transitioning through the SCXML state-chart.

Generally, it is desirable for an approach to support all five characteristics. Another important consideration is whether any given approach at integration is agnostic of the employed data-model, and the requirements for language feature it implies. Among the different approaches, only the third variation for an integration via custom invokers (see table 7.14) does exhibit all these desirable characteristics and we choose this approach to integrate the overwhelming majority of external systems in the `uSCXML` platform. However, there are a few remaining short-comings that, ever again, lead us to explore the other means to integration found in tables 7.13 - 7.16.

Usually, when attempting to integrate any given external system, we started by pondering about the minimal state that is required to instantiate such a system into a useful configuration. This initial state would then be supplied with the `<invoke>` element either as a set of `<param>` elements or via a `<content>` element. We hardly ever employ the `namelist` attribute to parametrize the instantiation via data-model expression, as the `<param>` element offers a superset of its functionality with the benefit of explicit names of parameters for intelligibility. Another consideration for the specification of this initial state and also subsequent events is the set of language features required by the data-model, e.g. only the ECMAScript data-model supports a representation of an XML DOM and we, oftentimes, defaulted to simple structured data to ease the requirements.

Whatever approach is selected for the integration of any given external system, ultimately, the issue of *standardizability* has to be considered as well: The implied pretense of SCXML in the context of the W3C MMI architecture is to standardize dialog management. As such, any given integration is only really useful, if it can be thoroughly standardized and eventually implemented in different interpreters. Here, it is desirable to rely upon as many established standards as is applicable.

7.2.6 Tools and Development Support

The functionality of the **uSCXML** platform is, foremost, contained in the C++ `libuscxml` library. This library contains our SCXML interpreter and a plugin interface for the extensions discussed above. The library and the plugin interface is also available via the language bindings in various target languages (e.g. Java and C#). There are, however, also two important command-line tools distributed with **uSCXML**:

- `uscxml-browser`:

  This binary provides the means to instantiate SCXML interpreters for a set of documents given as parameters on the command-line. In essence, it is merely a small wrapper around `libuscxml` and associated functionality, such as the live-debugger (see below).

- `uscxml-transform`:

  This binary complements the browser and provides the means to transform a given SCXML document for direct embedding in a target language. At the moment, only the `promela` and `scxml.fsm` targets are supported, realizing a transformation onto the PROMELA language and onto SCXML state-machines as described in the core contribution of this thesis.

  Ultimately, this binary ought to provide transformations of a given SCXML document onto other target languages as well. This allows for an alternative to embedding `libuscxml` as the state-chart would be described directly in the target language. Obviously, the functionality of the various extensions would be inaccessible as they are, with the exception of the ECMAScript Rhino data-model, written in C/C++. Here, it would be the responsibility of an application developer to provide the generated state-chart with respective callbacks.

7.2.6.1 On-line Debugging

The `uscxml-browser` tool also allows to start a debugging environment for SCXML documents. The debugger will register an URL with the HTTP server contained in the `uscxml-browser` binary and is subsequently available to (i) start new SCXML interpreters or (ii) attach to running instances, (iii) register breakpoints, (iv) inspect the document's configuration and data-model, (v) as well as evaluate data-model expressions.

We do provide one example client for this HTTP-based debugging interface as an HTML front-end which employs long-polling HTTP requests for communication (see figures 7.7, 7.8). The debugger is detailed in [RSWS14] and it provides most of the functionality developers come to expect from other, more established debuggers.

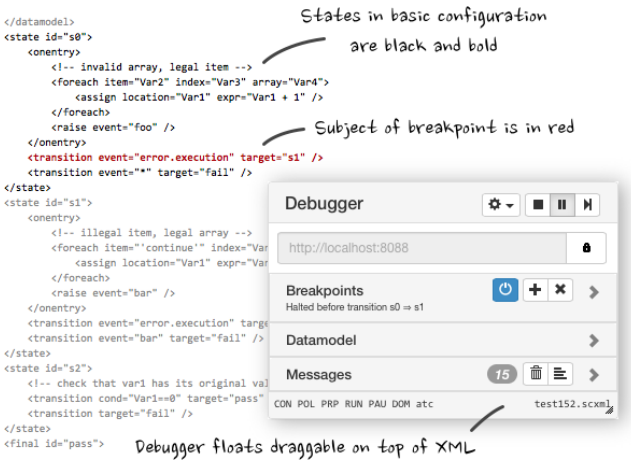


Figure 7.7.: Debugging interface on top of SCXML document in a HTML browser (from [RSWS14]).

Figure 7.8.: Elements of the debugger interface.

7.2.7 Example Applications in uSCXML

In the following sections, we will briefly showcase two applications that we realized with the `uSCXML` platform. Both were developed as part of the EU-funded SmartVortex project[29], with the multimodal map application published in [RSW13].

7.2.7.1 Multimodal Map

The multimodal map allows to display notifications for geo-referenced sensor data in a HTML browser showing a map of the world. Whenever such a notification arrives, its position on the map is marked and a brief clicking sound in relation to its geo-referenced coordinates and the user's current viewport is rendered via spatial audio. This provides a user with an additional, spatial awareness as to where the sensor data event occurred and allows to group such events mentally by their general area. Originally, we also had the notification's message spoken via text-to-speech, but this would become very confusing when events arrived in short order.

The application employs the `umundo` publish/subscribe invoker to receive notifications published from a data-stream management system as well as the `xhtml` and `spatial-audio` invokers to render the map with the markers and the audio notifications respectively.

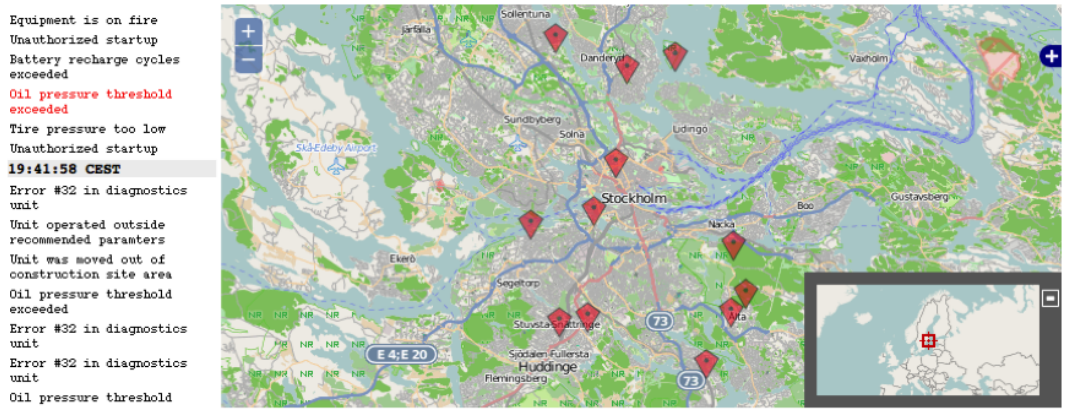


Figure 7.9.: Screenshot of the HTML interface with the map of the world and markers for the geo-referenced events.

7.2.7.2 FEM Model Visualization

This application was developed for an industrial partner within the SmartVortex consortium to visualize the progress of a finite element method process when minimizing a specified workpiece, e.g. with regard to overall mass. It will wait for one iteration of the process to produce a VRML file containing the voxels removed from the previous iteration and is available via a HTML frontend. Here, it allows to render these sequences as (i) individual images of the 3D model in a pose specified from within the HTML browser and (ii) a movie of the overall sequence with individual key poses to interpolate e.g. a rotating model which is progressively refined by the process. Ultimately, the rendering of individual images in a specified pose was substituted by a WebGL model rendered in the HTML browser, which allowed a more responsive exploration of a single iteration.

The respective SCXML document employs the `directory-monitor` invoker to watch a given directory for VRML models written by the FEM solver at the end of one iteration. Whenever a new file is found, it is transformed with the `model-convert` invoker into a binary representation of the three-dimensional model to speed up subsequent access. The `http` I/O processor is employed to wait for HTTP requests from an instance of the HTML frontend running. In this specific application, the frontend itself is not invoked but integrated into a web application server. Whenever a request for a given iteration with a specified pose is received, the `model-convert` invoker is used again to generate an image which is returned via the `<respond>` element.

The HTML frontend would allow to save individual images as "key poses" and eventually request a movie of all iterations with their pose interpolated. To this effect, the SCXML document would employ the `model-convert` and `movie` invokers to create e.g. a MPEG-4 encoded file which is returned, again, with the `<respond>` element.

[29] `http://cordis.europa.eu/project/rcn/96769_en.html` (accessed October 24th, 2015)

Figure 7.10.: Model for a workpiece displayed via WebGL in a HTML browser.

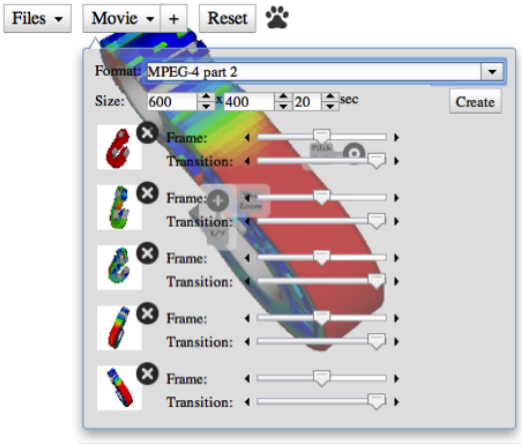


Figure 7.11.: Interface to provide key poses when exporting a sequence as a movie.

Advantages	Disadvantages	*Asynchronous*	*Markup Intact*	*State Changes from Events*	*Multiple Instances*	*Life-Cycle Management*	Example
`Data-Model`							
1. Represent modality component as a class		No	No	No	Yes	Yes	
• Approach very similar to more traditional programming. • Intuitive for developers to use. • Life-cycle management identical to that of objects.	• Implementation has to be provided per data-model. • Does not fit into overall concepts of SCXML. • Concept of classes and objects not available with every data-model.						`none`

Table 7.13.: Options to integrate a Modality Component via the data-model.

Advantages	Disadvantages	*Asynchronous*	*Markup Intact*	*State Changes from Events*	*Multiple Instances*	*Life-Cycle Management*	Example
`Invokers`							
1. Invoke-and-forget with URL or fixed data		Yes	Yes	No	Yes	No	
• Available with most external systems. • Minimal developer effort.	• No synchronization via events. • No feedback from invoked component.						`xhtml` (1st approach)
2. Invoke-and-forget with dynamic data		Yes	(Yes)[a]	No	Yes	No	
• Still pretty universally available with external systems. • Allows to parametrize the initial state via the data-model.	• No synchronization via events. • No feedback from invoked component.						`xhtml` (2nd approach)
3. Invocation and subsequent events		Yes	(Yes)[b]	Yes	Yes	Yes	
• Very little development effort. • Few requirements for data-model features. • Potentially very fine granular control of invoked system's state.	• Available functionality not obvious. • Requires access to an API for the invoked system. • Oftentimes degenerates into tightly coupled systems in the same runtime.						`xhtml` (3rd approach), `scxml`, `voicexml`, `movie`, `model-convert`, `smtp`, `spatial-audio`, `expect`, `instant-messaging`, `imap`, `heartbeat`, `ical-endar`, `directory-monitor`,
4. Shared DOM		No	Yes	(No)[c]	Yes	Yes	
• Established approach known from DHTML.	• Difficult to implement and maintain. • Very sensitive to API changes in the invoked system.						`scenegraph`

[a] Markup can remain intact when using the `${expr} syntax introduced above`.
[b] Initial DOM can be provided intact; usually only fragments in subsequent events.
[c] The `dom` I/O processor could be used to substitute modifications via a DOM Core Level 2 API changes by events.

Table 7.14.: Options to integrate a Modality Component via a custom invoker.

Advantages	Disadvantages	*Asynchronous*	*Markup Intact*	*State Changes from Events*	*Multiple Instances*	*Life-Cycle Management*	Example
`Executable Content`							
1. Embed Invoker's XML namespace		No	(No)[a]	No	No	No	

• No special considerations for different data-models.
• Can be very natural depending in the functionality, e.g.:
`<onentry>`
`  <vxml:prompt>I am Dave!</vxml:prompt>`
`</onentry>`
	• Life-cycle corresponds to SCXML document.
	• No parameters for instantiation.
	• Only a single instance of the integrated component.
	• No means to synchronize.
							`none`

2. Embed Invoker's XML via multiple namespaces		No	(No)	No	Yes	No	

• No special considerations for different data-models.
• Can still be very natural.
`<onentry>`
`  <vxml1:prompt>I am Dave!</vxml1:prompt>`
`  <vxml2:prompt>And I am Jane!</vxml2:prompt>`
`</onentry>`
	• All the disadvantages of the first approach, but multiple instances can be supported.						`none`

[a] Only the individual fragments would remain intact.

Table 7.15.: Options to integrate a Modality Component in via executable content.

Advantages	Disadvantages	*Asynchronous*	*Markup Intact*	*State Changes from Events*	*Multiple Instances*	*Life-Cycle Management*	Example
`I/O Processor`							
1. Represent modality component via an I/O processor		Yes	Yes	Yes	No	No	
• No special considerations per data-model.	• Only a single instance can be provided. • Life-cycle corresponds to SCXML document. • No means to parametrize initialization.						`none`

Table 7.16.: Options to integrate a Modality Component as a custom I/O processor.

8 Conclusion

This thesis presented a transformation for a large subset of Harel state-charts with the State Chart eXtensible Markup Language (SCXML) syntax and semantics onto PRocess MEta LAnguage (PROMELA) programs for the SPIN model-checker as the core contribution. This enabled the formal verification of temporal constraints and properties given in Linear Temporal Logic (LTL) for the systems described. This contribution derives a special relevance among the related approaches from the recommendation of the W3C Multimodal Interaction Working Group (W3C MMI WG) to realize the responsibilities for multimodal dialog management with SCXML. The applicability of (i) SCXML to express multimodal dialogs and (ii) our transformation onto PROMELA was evidenced, most notably, via the SCXML dialog model for the user interface of the Nvidia Shield handheld gaming console.

In order to align this main contribution with the pretense and expectations raised by the thesis' title, we argued for the necessity to show the following chain of propositions developed in the problem statement:

1. *Interaction in pervasive environments necessitates multiple devices and modalities.*

   This was largely treated as self-evident: If we are to realize the vision of Mark Weiser with regard to "disappearing technologies", a computer interface will have to leverage all the interaction cues a human user provides, i.e. interpret all the information conveyed via the available modalities. This necessitates the coordination of a wide range of sensors / actuators and devices, both for establishing the implicit or explicit interaction intent and delivering the system's response.

   We did provide an in-depth discussion of the various definitions for the term *modality* proposed over the years and finally settled upon a definition in reference to the concept of *perceptual mode* as identified in perceptual psychology. Furthermore, we also traced the term *dialog* through more than 50 years of Human-Computer Interaction (HCI) research and did show it to be an established notion of interaction in general. With the advent of User-Interface Management System (UIMS) and Multimodal Dialog System (MDS) at the latest, dialog management became a dedicated responsibility for an interactive system, giving rise to considerations with regard to suitable dialog management techniques as conceptualizations to express dialog models. In conjunction, this set the scope of our ambitions to identify a formally verifiable dialog management technique suited for multimodal interfaces in pervasive environments and aligned the contribution with the thesis' title.

2. *Multimodal interaction in pervasive environments can be expressed in a dialog model on a suitable platform.*

   We did show that there are (multimodal) dialog systems suited to be deployed in pervasive environments as largely independent, distributed platforms to realize interactive applications. A wealth of reference models was introduced with progressively converging responsibilities with regard to (i) input fusion, (ii) dialog management and (iii) output fission, motivating the necessity for a standardized approach. This necessity which was eventually answered by the W3C MMI WG via a set of World Wide Web Consortium (W3C) recommendations, most notably the W3C Multimodal Architecture and Interfaces (W3C MMI architecture) and the related W3C Multimodal Interaction Framework (W3C MMI framework) specification.

   We introduced a series of dialog management techniques and compared them with regard to their relative expressiveness and overall conception to SCXML as the proposal of the W3C MMI WG for a dialog management technique with their standardized reference model.

   Subsequently, we detailed the syntax and semantics of SCXML as defined in the respective W3C recommendation and argued for its applicability as a dialog management technique for distributed interaction in pervasive environments. Here, we also introduced the Prolog data-model as an embedded scripting languages for SCXML to support grounding and reasoning and extend the applicability of SCXML towards rule-based approaches. Even if we initially failed to convincingly show the immediate applicability of the W3C MMI architecture to provide runtime support for dialog models in pervasive environments, all that is needed is the existence of any such system and the applicability of SCXML to express respective dialog models. With SCXML supporting potentially arbitrary endpoints to deliver and accept events (i.e. publish / subscribe systems) and making no assumptions about the structure and content of the information contained in an event, we postulated this as given.

3. *These dialog models can be made accessible to the formalisms of temporal logic.*

   The argument to verify this proposition constituted the core contribution of this thesis. Starting with SCXML as a W3C recommendation for state-charts, we developed a data-model agnostic transformation onto SCXML

state-machines as an intermediate representation for the dialog model. The general approach was comparable to the transformation from Non-Deterministic Finite Automaton (NFA)s to Deterministic Finite Automaton (DFA)s, wherein flattened *global states* are connected via potential *optimal transition sets* merged into *global transitions*. This initial transformation preserved all semantics of the original state-chart (if we allowed for three slight extensions of an interpreter).

Semantic equivalence of the state-machine and state-chart representation was shown via the 232 tests for functional requirements, accompanying SCXML as part of its Implementation Report Plan (IRP) suggested by the W3C process. It is noteworthy, that these SCXML state-machines, even without any data-model, were not equivalent to DFAs in the Chomsky hierarchy as we were able to embed both, (i) a Push-Down Automaton (PDA) with its stack via nested state-chart invocations and (ii) even a Deterministic Queue Automaton (DQA) as a Turing machine equivalence with its queue via the conceptually infinite event queues in SCXML, thus showing Turing completeness. This representation was complemented by a closed formula for the upper-bound of states in the state-machine representation evaluated for its *tightness*.

The intermediate state-machine representation was subsequently transformed, onto PROMELA programs as input files for the Spin model checker. Here, PROMELA served as a convenience transformation target with the SPIN model-checker implicitly transforming these programs onto a DFA (actually onto a Kripke structure). The equivalence to Turing machines and the implied problem of the Halting problem (i.e. `eventually(state == final)`) was resolved by limiting the event queues, the embedding of a PDA was prevented by prohibiting recursive invocations. In conjunction, both limitations were required to establish finite enumerability of an SCXML document's overall state and thus equivalence to a DFA.

In order for this transformation onto PROMELA programs to be applicable to a non-trivial amount of SCXML IRP tests, we had to define a data-model with a syntax and semantic that could be subjected to the transformation. Here, we pragmatically settled on providing a subset of the PROMELA language itself. This allowed us to transform 132 of 182 applicable tests and to formally verify that they passed via LTL expressions (i.e. `eventually(state == pass)`), showing the expressiveness retained in the transformation function's domain. The majority of expressiveness lost (27 of 50 tests) was due to unsupportable error semantics of a compliant SCXML interpreter, e.g. in case of syntax errors in data-model expressions or communication timeouts.

4. *The approach is applicable for non-trivial applications.*

   While the SCXML IRP tests already established functional applicability of the approach as the subset of SCXML verifiable, they are deliberately small and concise. The transformation would have remained a rather academic exercise without an application to a real-world dialog model of a more realistic size. Here, we were fortunate to have access to an early version of the SCXML dialog model for the Nvidia Shield handheld gaming device. While this dialog model does not employ all semantic features of SCXML, it is used in an actual consumer product and was written without the original authors being aware the approach detailed in this thesis. By transforming its dialog model and performing an exhaustive search of its state space we were able to show actual applicability for a non-trivial system.

   When attempting to apply our approach for formal verification on the Nvidia Shield's SCXML dialog model, two deficiencies became apparent: (1) The identification of global transitions as potential optimal transition sets per global state by starting with the power-set of all active transitions and subsequently removing invalid sets proofed to be very expensive with more than 2.4 billion sets to consider and (2) the size of the resulting PROMELA program with ~440MB turned out to be very costly for SPIN to transform into a binary for an exhaustive search. The first problem was solved by (i) exploiting the transition selection rules of SCXML and (ii) the realization that only the state-chart's active configuration needs to be considered when identifying the global transitions per global state. This lead to a more sane starting set of potential optimal transition sets to consider, dropping the total amount by a factor of ~17.000 in the Nvidia Shield case. The second problem was approached by removing the encoding of history assignments from global states and making these explicit in PROMELA via boolean arrays, dropping the size of the resulting PROMELA program by a factor of ~170.

   With both of these adaptations to the original transformation, we were able to complete the process starting from the SCXML dialog model to the result of an exhaustive search of its state-space in ~13min. Only approximately one third of the time was required to arrive at the binary for the exhaustive search, which can then be run in parallel for every LTL expression to be verified.

8.1 Critical Reflection

Even though I am convinced of the overall novelty, relevance and evaluation of the contribution described in this thesis, there are a few missing pieces and general remarks I like to make in the interest of scientific rigor:

- *There is no concluding argument for the DFA equivalence of the bound SCXML subset.*

  When we started our work to transform SCXML onto PROMELA, our original intuition lead us to believe that SCXML would already by DFA equivalent. This intuition was based on the assumption that state-charts were just *syntactic sugar* for finite state-machines. It was only when we ran into the problem of determining the upper bound for SCXML's internal and external queue in PROMELA that we realized its actual computational model as Turing complete. Subsequently, we assumed that limiting the length of those queues to an arbitrary but fixed upper bound would restore DFA equivalence. The realization that one can also embed a PDA via recursive invocations came only when working through the survey of Green [Gre86], wherein he argues for PDA equivalence of recursive transition networks.

  As such, there is no concluding evidence that the subset of SCXML with bound event queues and non-recursive invocations is DFA equivalent. We did argue that the remaining semantic retains finite enumerability, but there might still be language features which falsify this assumption. All we can say is that every SCXML document transformed onto PROMELA is necessarily DFA equivalent, but there might still be SCXML documents in the bound subset which are not transformable.

- *We did not show real-world applicability of all SCXML language features that were transformable.*

  While we did show usefulness and applicability of the approach by transforming an actual dialog model of a real-world consumer product, only a subset of transformable SCXML language features were employed therein. For all other transformable language features, we only did show that they can be expressed in PROMELA, not their real-world applicability. For instance, using the `idlocation` attribute with the `<send>` element requires a compliant interpreter to automatically assign a unique identifier at the given location. This will cause a considerable increase of the state space for an exhaustive search as each event sent will be considered different.

- *Real-world system is not necessarily an application in pervasive environments.*

  Even though the transformation and exhaustive search of the state-space of the SCXML dialog model for the Nvidia Shield handheld gaming console is a proof for the relevance of the approach in a real-world application, it is not exactly a good example for multimodal interaction in pervasive environments. Then again, formally verifying distributed, concurrent processes is one of the core competencies of the SPIN model checker and modeling distributed, modality-agnostic interaction a core competency of SCXML. Therefore, I remain confident in claiming applicability for multimodal dialogs in pervasive environments as the pretense raised by this thesis' title.

- *Choice of SPIN as a model-checker is unmotivated.*

  Formally, we would have needed to evaluate the various model-checkers for suitability to verify distributed dialog models prior to settling on one implementation, and the unmotivated choice of SPIN without a more elaborate discussion of comparative studies or respective original research is certainly a weakness of this thesis. In all honesty, the choice for SPIN was made due to prior experiences with the tool and the assumption that an implementation as acclaimed as SPIN (e.g. it won the ACM Software System Award in 2001) would not reveal major deficiencies, with us already having invested a considerate amount of work. Ultimately, a transformation of SCXML onto input languages of other model-checkers is definitely desirable and, here, our semantically equivalent intermediate state-machine representation could considerably ease the requirements with regard to the expressiveness these languages would have to offer.

- *State space and runtime for exhaustive verification with SPIN could be reduced considerably.*

  Even though we choose SPIN for its assumed maturity, there are some situations were we left its "comfort zone" and, indeed, even some syntactic short-comings that prevent a smaller state space for the exhaustive search during verification:

  1. In PROMELA, sequences of statements that are meant to be processed uninterleaved by statements from concurrent processes can be enclosed in `atomic` blocks. A more restrictive alternative are `d_step` blocks wherein a blocking statement would even constitute a syntactic error. As the *perfect synchrony hypothesis* holds for SCXML, all events are processed instantaneously and the semantic will not allow for any state changes in between events. As we also removed all blocking statements in the PROMELA data-model, it is these `d_step` blocks that can be used to wrap all statements in between event dequeueing. However, PROMELA will not allow to use the `goto` statement to redirect control flow into and from within these blocks. This is a problem as we constructed our PROMELA state-machine to separate event dispatching and transition selection from the actual executable content per global transition and use the `goto` statement to jump to the respective executable content and back to the next micro-step

when processing is completed. Here, we are required to break the `d_step` block when redirecting control flow, causing SPIN to "branch out" for every other process with non-blocking statements. It would be possible to interleave the executable content with event dispatching and transition selection into one giant `if`-cascade, but readability of the PROMELA program would suffer.

2. An indication that our usage of SPIN pushed its limits is the fact that the length of individual `d_step` blocks is constrained at compile time to some arbitrary number. We had to quadruple this number prior to compiling SPIN to be able to process the Nvidia Shield's SCXML dialog model.
3. Yet another unfortunate deficiency of SPIN is the fact that, even though global variables can be declared `hidden` to disregard their value when determining the equivalence of two states during exhaustive verification, no such feature exists for members of a compound variable. This proofed to be a problem as we initially employed the *sorted insert* (`!!`) operator to enqueue delayed events, which required a sequence identifier as the second field to prevent sorting by the subsequent fields in the event compound (such as the event's name). This caused SPIN to regard every such event as unique, increasing the state space by 2^{32} bits, one for every possible value of the events' sequence number.

- *A more concise narrative could have been chosen.*

  The core contribution of this thesis could have been motivated solely via SCXML as a standardized statechart syntax and semantic with no reference to interaction at all. However, many of our previous publications indicated the chosen narrative and indeed much of the relevance of the approach only becomes apparent in the scope of interaction in pervasive environments: The implied aspiration of the W3C MMI WG is to reproduce the overwhelming success of Hypertext Markup Language (HTML) for interaction, not limited to graphical user interfaces, but distributed interaction via multiple modalities in general, as it is required for pervasive environments.

8.2 Outlook

There is a wide range of possibilities for subsequent activities connecting to the contribution of this thesis, both scientific and applied:

- **Improving verification performance**

  One immediate area of work is to improve upon the runtime and memory requirements for the actual verification with SPIN. While we made every attempt to keep the characteristics of both within reasonable limits, there are still many design decisions in the actual transformation from SCXML onto PROMELA that warrant a more thorough evaluation. Here, the ultimate goal is to minimize the implicit Kripke structure established by SPIN to one that will only branch out to account for different sequences of events, thereby exploiting the perfect synchrony hypothesis. Furthermore, problems with the modeling of some SCXML language features in PROMELA (most notably delayed events) may even justify adaptations to the SPIN source code to avoid unnecessary computations during verification.

- **Other model-checkers**

  A related issue is the transformation onto input languages of other model-checkers and a comparative evaluation of their runtime and memory requirements. Here, the intermediate state-machine representation can provide a common basis to simplify the process. Targeting different model-checkers will also enable other classes of temporal logic (e.g. Computation Tree Logic (CTL)) to become available for verification.

- **Resource constrained devices**

  One very nice consequence of limiting the expressiveness of SCXML to DFA equivalence is the resulting simplification for implementations on resource constrained devices and the potential for optimizations on all platforms. E.g., with the complete state being finite enumerable, we can pre-allocate all required memory on the system's stack to avoid costly dynamic allocation at runtime and benefit from CPU cache locality. DFA equivalence is also relevant when ultimately providing application-specific integrated circuits or even only field-programmable gate arrays to realize the system as dedicated hardware elements as it simplifies the respective implementation in a suitable hardware description language considerably.

- **Verifying rule-based dialog management techniques**

  Finally, another extension of the transformation's domain onto rule-based dialog management techniques might be applicable. Herein, the implementation of SLD-resolution for grounding and reasoning via Horn-clauses

would need to be "unrolled" into (possibly huge) and-or trees of predicates, essentially performing resolution during verification. To this effect, we already implemented the Prolog data-model and one would need to provide a more restricted subset to retain DFA equivalence.

As the final conclusion, I would like to argue that this thesis does provide a contribution at a very important *junction* of technologies. Currently, there are multiple competing solutions to realize interaction in pervasive environments, evidenced by the various products and platforms presented by different commercial consortia, e.g. on IFA 2015. Here, SCXML has the potential to abstract from the specific platforms and provide a canonical dialog management technique with a number of desirable properties. Its flexibility and expressiveness with regard to the various data-models as embedded scripting languages, the possibilities for static analysis and even formal verification enabled by this thesis, as well as its recommendation status from the W3C as a renowned institution for standardization might indeed suffice to establish SCXML as the "HTML of distributed, multimodal interaction".

Part IV.
Appendix

A Program Listings

A.1 Complete Transformation Example

The following listings detail a complete transformation of an SCXML document containing most language features to an SCXML state-machine and ultimately a PROMELA program. The original SCXML document as the basis for the transformation is given in listing A.1. It contains all elements from the SCXML recommendation and will simply transition through a completely predetermined series of configurations until it reaches the final state `pass`.

As with the examples in section 5.3.6, the XML in the subsequent listings is annotated with some attributes without semantic in SCXML to ease understanding of the transformation. With the original document in listing A.1, this is only the `priority` attribute for `<transition>` elements in proper states as the priority of a transition when identifying the optimal transition set (cf. equation 5.17), which is also used as an identifier for such a transition.

```
<scxml datamodel="promela">
  <datamodel>
    <data id="foreachArray1" type="int[3]">[1,2,3]</data>
    <data id="parallelVar1" type="int" expr="0"/>
    <data id="ifVar1">{ foo: 1, bar: 'baz' }</data>
    <data id="counter">{ itemSum: 0, indexSum: 0 }</data>
    <data id="sendVar1" type="int" expr="4"/>
    <data id="histVar1" type="int">0</data>
    <data id="finalizeVar1" type="int">0</data>
  </datamodel>

  <state id="s0">
    <history id="s0.h0" type="deep" />
    <parallel id="p0">
      <state id="p0.s0">
        <state id="p0.s0.s0">
          <onentry>
            <if cond="ifVar1.foo == 3">
              <log label="if choosen" />
              <log label="ifVar1.bar is" expr="ifVar1.bar" />
              <foreach array="foreachArray1" item="foreachItem1" index="foreachIndex1">
                <script>
                  counter.indexSum = counter.indexSum + foreachIndex1;
                  counter.itemSum = counter.itemSum + foreachItem1;
                </script>
                <log label="foreach counter.indexSum is" expr="counter.indexSum" />
                <log label="foreach counter.itemSum is" expr="counter.itemSum" />
              </foreach>
              <raise event="if.choosen" />
            <elseif cond="ifVar1.bar == 'baz'" />
              <log label="elseif choosen" />
              <log label="ifVar1.bar is" expr="ifVar1.bar" />
              <assign location="ifVar1.foo" expr="3" />
              <send event="elseif.choosen" namelist="sendVar1">
                <param name="foo" expr="sendVar1 + 16" />
                <param name="bar" expr="'a string literal'" />
              </send>
            <else />
              <log label="else choosen" />
              <log label="ifVar1.foo is" expr="ifVar1.foo" />
              <log label="ifVar1.bar is" expr="ifVar1.bar" />
              <raise event="else.choosen" />
            </if>
            <script>parallelVar1++</script>
          </onentry>
          <transition priority="11" event="else.choosen" target="p0">
            <assign location="ifVar1.bar" expr="'baz'" />
          </transition>
          <transition cond="_event.data.foo == 20 &amp;&amp;
                            _event.data.sendVar1 == 4 &amp;&amp;
                            _event.data.bar == 'a string literal'"
                      priority="10" event="elseif.choosen" target="p0"/>
        </state>
        <state id="p0.s0.s1">
          <onentry>
            <if cond="_x.states['p0'] &amp;&amp; histVar1 == 1">
              <raise event="to.s2" />
            <else />
              <raise event="to.s1" />
            </if>
          </onentry>
          <transition priority="9" event="to.s2" target="s2" />
```

```
63            <transition priority="8" event="to.s1" target="s1" />
64          </state>
65        </state>
66        <state id="p0.s1">
67          <onexit>
68            <script>parallelVar1++</script>
69          </onexit>
70        </state>
71        <transition priority="7" event="if.choosen" target="p0.s0.s1"
72                    cond="counter.itemSum == 6 &amp;amp;&amp;amp; counter.indexSum == 3" />
73      </parallel>
74    </state>
75
76    <state id="s1">
77      <invoke type="scxml" autoforward="true">
78        <content>
79          <scxml datamodel="promela">
80            <state id="waitForEvent">
81              <transition event="trigger.child">
82                <send target="#_parent" event="back.to.history" />
83              </transition>
84            </state>
85          </scxml>
86        </content>
87        <finalize>
88          <script>finalizeVar1++;</script>
89        </finalize>
90      </invoke>
91      <onentry>
92        <send event="trigger.child" delay="1000" />
93      </onentry>
94      <transition priority="6" event="back.to.history" target="s0.h0"
95                  cond="finalizeVar1 == 1">
96        <assign location="histVar1" expr="4-3" />
97      </transition>
98    </state>
99
100   <state id="s2" initial="s2.s0">
101     <onentry>
102       <send event="cancel.delayed" delay="3000" sendid="cancel.delayed" />
103       <cancel sendid="cancel.delayed" />
104     </onentry>
105     <transition priority="4" event="done.state.s2" target="s3.h0"
106                 cond="_event.data.Var1 == 'foo'" >
107       <assign location="histVar1" expr="8" />
108     </transition>
109     <transition priority="3" event="done.state.s2" target="fail" />
110     <transition priority="2" event="cancel.delayed" target="fail" />
111     <transition priority="1" target="pass" cond="histVar1 == 8" />
112     <state id="s2.s0">
113       <transition priority="5" target="s2.s1"/>
114     </state>
115     <final id="s2.s1">
116       <donedata>
117         <param name="Var1" expr="'foo'"/>
118       </donedata>
119     </final>
120   </state>
121
122   <state id="s3">
123     <history id="s3.h0" type="shallow">
124       <transition target="s3.s1">
125         <log label="history transition" />
126         <assign location="histVar1" expr="4" />
127       </transition>
128     </history>
129     <state id="s3.s1">
130       <transition priority="0" target="s2" />
131     </state>
132   </state>
133
134   <final id="pass">
135     <onentry>
136       <log label="Outcome" expr="'pass'"/>
137     </onentry>
138   </final>
139   <final id="fail">
140     <onentry>
141       <log label="Outcome" expr="'fail'"/>
142     </onentry>
143   </final>
144 </scxml>
```

Listing A.1: Example SCXML state-chart document with most language features.

Table A.1 lists the global state table as introduced in section 5.3.6.1 for the state-chart in listing A.1. We can see that the transformation will generate 15 global states $\widetilde{s}(0) \ldots \widetilde{s}(14)$ and 29 global transitions (last column) for the SCXML state-machine representation.

Step	Global State / Optimal Transition Sets	Invalidation Criteria / Executable Content	Target
$\widetilde{s}(0)$	$\widetilde{s}_a(0)$: $\emptyset$ $\widetilde{s}_d(0)$: $\emptyset$ $\widetilde{s}_h(0)$: $\emptyset$	$\{t_{initial}\}$	
$\widetilde{\mathcal{T}}(0)$	$\{t_{initial}\}$	$\mathcal{X} := (l_{18-44})$	$\widetilde{s}(1)$
$\widetilde{s}(1)$	$\widetilde{s}_a(1)$: `{s0, p0, p0.s0, p0.s0.s0, p0.s1}` $\widetilde{s}_d(1)$: $\emptyset$ $\widetilde{s}_h(1)$: $\emptyset$	$\{t_7, t_{11}, t_{10}\}$	
$\widetilde{\mathcal{T}}(1)$	~~$\{t_{11}, t_{10}, t_7\}$~~ ~~$\{t_{11}, t_{10}\}$~~ ~~$\{t_{11}, t_7\}$~~ $\{t_{11}\}$ ~~$\{t_{10}, t_7\}$~~ $\{t_{10}\}$ $\{t_7\}$	*Inv₃: Same source state* *Inv₃: Same source state* *Inv₄: Nested transitions* $\mathcal{X} := (l_{68}, l_{47}, l_{18-44})$ *Inv₄: Nested transitions* $\mathcal{X} := (l_{68}, l_{18-44})$ $\mathcal{X} := (l_{68}, l_{56-60})$	 $\widetilde{s}(1)$ $\widetilde{s}(1)$ $\widetilde{s}(2)$
$\widetilde{s}(2)$	$\widetilde{s}_a(2)$: `{s0, p0, p0.s0, p0.s0.s1, p0.s1}` $\widetilde{s}_d(2)$: $\emptyset$ $\widetilde{s}_h(2)$: $\emptyset$	$\{t_9, t_8, t_7\}$	
$\widetilde{\mathcal{T}}(2)$	~~$\{t_9, t_8, t_7\}$~~ ~~$\{t_9, t_8\}$~~ ~~$\{t_9, t_7\}$~~ $\{t_9\}$ ~~$\{t_8, t_7\}$~~ $\{t_8\}$ $\{t_7\}$	*Inv₃: Same source state* *Inv₃: Same source state* *Inv₄: Nested transitions* $\mathcal{X} := (l_{68}, l_{102-103})$ *Inv₄: Nested transitions* $\mathcal{X} := (l_{68}, l_{92})$ $\mathcal{X} := (l_{68}, l_{56-60})$	 $\widetilde{s}(3)$ $\widetilde{s}(4)$ $\widetilde{s}(2)$
$\widetilde{s}(3)$	$\widetilde{s}_a(3)$: `{s2, s2.s0}` $\widetilde{s}_d(3)$: $\emptyset$ $\widetilde{s}_h(3)$: `{s0.h0:{p0.s0.s1, p0.s1}}`	$\{t_5, t_4, t_3, t_2, t_1\}$	
$\widetilde{\mathcal{T}}(3)$	~~$\{t_5, t_4, t_3, t_2, t_1\}$~~ ~~$\{t_5, t_4, t_3, t_2\}$~~ ~~$\{t_5, t_4, t_3, t_1\}$~~ ~~$\{t_5, t_4, t_3\}$~~ ~~$\{t_5, t_4, t_2, t_1\}$~~ ~~$\{t_5, t_4, t_2\}$~~ ~~$\{t_5, t_4, t_1\}$~~ ~~$\{t_5, t_4\}$~~ ~~$\{t_5, t_3, t_2, t_1\}$~~ ~~$\{t_5, t_3, t_2\}$~~ ~~$\{t_5, t_3, t_1\}$~~ ~~$\{t_5, t_3\}$~~ ~~$\{t_5, t_2, t_1\}$~~ ~~$\{t_5, t_2\}$~~ ~~$\{t_5, t_1\}$~~ $\{t_5\}$ ~~$\{t_4, t_3, t_2, t_1\}$~~ ~~$\{t_4, t_3, t_2\}$~~ ~~$\{t_4, t_3, t_1\}$~~ ~~$\{t_4, t_3\}$~~ ~~$\{t_4, t_2, t_1\}$~~ ~~$\{t_4, t_2\}$~~ ~~$\{t_4, t_1\}$~~ $\{t_4\}$ ~~$\{t_3, t_2, t_1\}$~~ ~~$\{t_3, t_2\}$~~ ~~$\{t_3, t_1\}$~~ $\{t_3\}$ ~~$\{t_2, t_1\}$~~ $\{t_2\}$	*Inv₃: Same source state* *Inv₃: Same source state* *Inv₃: Same source state* *Inv₃: Same source state* *Inv₃: Same source state* *Inv₃: Same source state* *Inv₃: Same source state* *Inv₄: Nested transitions* *Inv₃: Same source state* *Inv₃: Same source state* *Inv₃: Same source state* *Inv₄: Nested transitions* *Inv₃: Same source state* *Inv₄: Nested transitions* *Inv₄: Nested transitions* $\emptyset$ *Inv₃: Same source state* *Inv₃: Same source state* *Inv₃: Same source state* *Inv₃: Same source state* *Inv₃: Same source state* *Inv₃: Same source state* *Inv₃: Same source state* $\mathcal{X} := (l_{107}, l_{125-126})$ *Inv₃: Same source state* *Inv₃: Same source state* *Inv₃: Same source state* $\mathcal{X} := (l_{141})$ *Inv₃: Same source state* $\mathcal{X} := (l_{141})$	 $\widetilde{s}(5)$ $\widetilde{s}(6)$ $\widetilde{s}(7)$ $\widetilde{s}(7)$

	~~$\{t_1\}$~~	*Opt_1: Earlier unconditional match*	
$\widetilde{s}(4)$	$\widetilde{s}_a(4)$: `{s1}` $\widetilde{s}_d(4)$: $\emptyset$ $\widetilde{s}_h(4)$: `{s0.h0:{p0.s0.s1, p0.s1}}`	$\{t_6\}$	
$\widetilde{\mathcal{T}}(4)$	$\{t_6\}$	$\mathcal{X} := (l_{96}, l_{56-60})$	$\widetilde{s}(8)$
$\widetilde{s}(5)$	$\widetilde{s}_a(5)$: `{s2, s2.s1}` $\widetilde{s}_d(5)$: $\emptyset$ $\widetilde{s}_h(5)$: `{s0.h0:{p0.s0.s1, p0.s1}}`	$\{t_4, t_3, t_2, t_1\}$	
$\widetilde{\mathcal{T}}(5)$	~~$\{t_4, t_3, t_2, t_1\}$~~ ~~$\{t_4, t_3, t_2\}$~~ ~~$\{t_4, t_3, t_1\}$~~ ~~$\{t_4, t_3\}$~~ ~~$\{t_4, t_2, t_1\}$~~ ~~$\{t_4, t_2\}$~~ ~~$\{t_4, t_1\}$~~ $\{t_4\}$ ~~$\{t_3, t_2, t_1\}$~~ ~~$\{t_3, t_2\}$~~ ~~$\{t_3, t_1\}$~~ $\{t_3\}$ ~~$\{t_2, t_1\}$~~ $\{t_2\}$ $\{t_1\}$	*Inv_3: Same source state* *Inv_3: Same source state* *Inv_3: Same source state* *Inv_3: Same source state* *Inv_3: Same source state* *Inv_3: Same source state* *Inv_3: Same source state* $\mathcal{X} := (l_{107}, l_{125-126})$ *Inv_3: Same source state* *Inv_3: Same source state* *Inv_3: Same source state* $\mathcal{X} := (l_{141})$ *Inv_3: Same source state* $\mathcal{X} := (l_{141})$ $\mathcal{X} := (l_{136})$	 $\widetilde{s}(6)$ $\widetilde{s}(7)$ $\widetilde{s}(7)$ $\widetilde{s}(9)$
$\widetilde{s}(6)$	$\widetilde{s}_a(6)$: `{s3, s3.s1}` $\widetilde{s}_d(6)$: $\emptyset$ $\widetilde{s}_h(6)$: `{s0.h0:{p0.s0.s1, p0.s1}}`	$\{t_0\}$	
$\widetilde{\mathcal{T}}(6)$	$\{t_0\}$	$\mathcal{X} := (l_{102-103})$	$\widetilde{s}(10)$
$\widetilde{s}(7)$	$\widetilde{s}_a(7)$: `{fail}` $\widetilde{s}_d(7)$: $\emptyset$ $\widetilde{s}_h(7)$: `{s0.h0:{p0.s0.s1, p0.s1}}`	$\emptyset$	
$\widetilde{s}(8)$	$\widetilde{s}_a(8)$: `{s0, p0, p0.s0, p0.s0.s1, p0.s1}` $\widetilde{s}_d(8)$: $\emptyset$ $\widetilde{s}_h(8)$: `{s0.h0:{p0.s0.s1, p0.s1}}`	$\{t_9, t_8, t_7\}$	
$\widetilde{\mathcal{T}}(8)$	~~$\{t_9, t_8, t_7\}$~~ ~~$\{t_9, t_8\}$~~ ~~$\{t_9, t_7\}$~~ $\{t_9\}$ ~~$\{t_8, t_7\}$~~ $\{t_8\}$ $\{t_7\}$	*Inv_3: Same source state* *Inv_3: Same source state* *Inv_4: Nested transitions* $\mathcal{X} := (l_{68}, l_{102-103})$ *Inv_4: Nested transitions* $\mathcal{X} := (l_{68}, l_{92})$ $\mathcal{X} := (l_{68}, l_{56-60})$	 $\widetilde{s}(3)$ $\widetilde{s}(4)$ $\widetilde{s}(8)$
$\widetilde{s}(9)$	$\widetilde{s}_a(9)$: `{pass}` $\widetilde{s}_d(9)$: $\emptyset$ $\widetilde{s}_h(9)$: `{s0.h0:{p0.s0.s1, p0.s1}}`	$\emptyset$	
$\widetilde{s}(10)$	$\widetilde{s}_a(10)$: `{s2, s2.s0}` $\widetilde{s}_d(10)$: $\emptyset$ $\widetilde{s}_h(10)$: `{s0.h0:{p0.s0.s1, p0.s1}, s3.h0:{s3.s1}}`	$\{t_5, t_4, t_3, t_2, t_1\}$	
$\widetilde{\mathcal{T}}(10)$	~~$\{t_5, t_4, t_3, t_2, t_1\}$~~ ~~$\{t_5, t_4, t_3, t_2\}$~~ ~~$\{t_5, t_4, t_3, t_1\}$~~ ~~$\{t_5, t_4, t_3\}$~~ ~~$\{t_5, t_4, t_2, t_1\}$~~ ~~$\{t_5, t_4, t_2\}$~~ ~~$\{t_5, t_4, t_1\}$~~ ~~$\{t_5, t_4\}$~~ ~~$\{t_5, t_3, t_2, t_1\}$~~ ~~$\{t_5, t_3, t_2\}$~~ ~~$\{t_5, t_3, t_1\}$~~ ~~$\{t_5, t_3\}$~~ ~~$\{t_5, t_2, t_1\}$~~ ~~$\{t_5, t_2\}$~~ ~~$\{t_5, t_1\}$~~	*Inv_3: Same source state* *Inv_3: Same source state* *Inv_3: Same source state* *Inv_3: Same source state* *Inv_3: Same source state* *Inv_3: Same source state* *Inv_3: Same source state* *Inv_4: Nested transitions* *Inv_3: Same source state* *Inv_3: Same source state* *Inv_3: Same source state* *Inv_4: Nested transitions* *Inv_3: Same source state* *Inv_4: Nested transitions* *Inv_4: Nested transitions*	

	$\{t_5\}$	$\emptyset$	$\widetilde{s}(11)$
	~~$\{t_4, t_3, t_2, t_1\}$~~	Inv_3: *Same source state*	
	~~$\{t_4, t_3, t_2\}$~~	Inv_3: *Same source state*	
	~~$\{t_4, t_3, t_1\}$~~	Inv_3: *Same source state*	
	~~$\{t_4, t_3\}$~~	Inv_3: *Same source state*	
	~~$\{t_4, t_2, t_1\}$~~	Inv_3: *Same source state*	
	~~$\{t_4, t_2\}$~~	Inv_3: *Same source state*	
	~~$\{t_4, t_1\}$~~	Inv_3: *Same source state*	
	$\{t_4\}$	$\mathcal{X} := (l_{107})$	$\widetilde{s}(12)$
	~~$\{t_3, t_2, t_1\}$~~	Inv_3: *Same source state*	
	~~$\{t_3, t_2\}$~~	Inv_3: *Same source state*	
	~~$\{t_3, t_1\}$~~	Inv_3: *Same source state*	
	$\{t_3\}$	$\mathcal{X} := (l_{141})$	$\widetilde{s}(13)$
	~~$\{t_2, t_1\}$~~	Inv_3: *Same source state*	
	$\{t_2\}$	$\mathcal{X} := (l_{141})$	$\widetilde{s}(13)$
	~~$\{t_1\}$~~	Opt_1: *Earlier unconditional match*	
$\widetilde{s}(11)$	$\widetilde{s}_a(11)$: `{s2, s2.s1}` $\widetilde{s}_d(11)$: $\emptyset$ $\widetilde{s}_h(11)$: `{s0.h0:{p0.s0.s1, p0.s1}, s3.h0:{s3.s1}}`	$\{t_4, t_3, t_2, t_1\}$	
$\widetilde{\mathcal{T}}(11)$	~~$\{t_4, t_3, t_2, t_1\}$~~	Inv_3: *Same source state*	
	~~$\{t_4, t_3, t_2\}$~~	Inv_3: *Same source state*	
	~~$\{t_4, t_3, t_1\}$~~	Inv_3: *Same source state*	
	~~$\{t_4, t_3\}$~~	Inv_3: *Same source state*	
	~~$\{t_4, t_2, t_1\}$~~	Inv_3: *Same source state*	
	~~$\{t_4, t_2\}$~~	Inv_3: *Same source state*	
	~~$\{t_4, t_1\}$~~	Inv_3: *Same source state*	
	$\{t_4\}$	$\mathcal{X} := (l_{107})$	$\widetilde{s}(12)$
	~~$\{t_3, t_2, t_1\}$~~	Inv_3: *Same source state*	
	~~$\{t_3, t_2\}$~~	Inv_3: *Same source state*	
	~~$\{t_3, t_1\}$~~	Inv_3: *Same source state*	
	$\{t_3\}$	$\mathcal{X} := (l_{141})$	$\widetilde{s}(13)$
	~~$\{t_2, t_1\}$~~	Inv_3: *Same source state*	
	$\{t_2\}$	$\mathcal{X} := (l_{141})$	$\widetilde{s}(13)$
	$\{t_1\}$	$\mathcal{X} := (l_{136})$	$\widetilde{s}(14)$
$\widetilde{s}(12)$	$\widetilde{s}_a(12)$: `{s3, s3.s1}` $\widetilde{s}_d(12)$: $\emptyset$ $\widetilde{s}_h(12)$: `{s0.h0:{p0.s0.s1, p0.s1}, s3.h0:{s3.s1}}`	$\{t_0\}$	
$\widetilde{\mathcal{T}}(12)$	$\{t_0\}$	$\mathcal{X} := (l_{102-103})$	$\widetilde{s}(10)$
$\widetilde{s}(13)$	$\widetilde{s}_a(13)$: `{fail}` $\widetilde{s}_d(13)$: $\emptyset$ $\widetilde{s}_h(13)$: `{s0.h0:{p0.s0.s1, p0.s1}, s3.h0:{s3.s1}}`	$\emptyset$	
$\widetilde{s}(14)$	$\widetilde{s}_a(14)$: `{pass}` $\widetilde{s}_d(14)$: $\emptyset$ $\widetilde{s}_h(14)$: `{s0.h0:{p0.s0.s1, p0.s1}, s3.h0:{s3.s1}}`	$\emptyset$	

Table A.1.: Global state table for the SCXML state-chart in listing A.1.

Listing A.2 displays the SCXML state-machine obtained from the construction described in section 5.3 for the state-chart in listing A.1. The `step` attribute identifies global states per their order in the breadth-first traversal of reachable configurations, the `member` attribute for global transitions lists their constituting transition elements from the original state-chart. State elements as members of transient state chains (see section 5.3.3) are annotated with a `transient` attribute.

```
<scxml flat="true" datamodel="promela" initial="active:{}">
  <datamodel>
    <data id="foreachArray1" type="int[3]">[1,2,3]</data>
    <data id="parallelVar1" expr="0" type="int" />
    <data id="ifVar1">{ foo: 1, bar: 'baz' }</data>
    <data id="counter">{ itemSum: 0, indexSum: 0 }</data>
    <data id="sendVar1" expr="4" type="int" />
    <data id="histVar1" type="int">0</data>
    <data id="finalizeVar1" type="int">0</data>
  </datamodel>

  <!-- Global State  -->
  <state step="0"
         id="active:{}">
    <transition members="                          "
                target="active:{s0,p0,p0.s0,p0.s0.s0}-via-0" />
  </state>
  <state transient="true"
         id="active:{s0,p0,p0.s0,p0.s0.s0}-via-0">
    <onentry>
      <if cond="ifVar1.foo == 3">
        <log label="if choosen" />
        <log expr="ifVar1.bar" label="ifVar1.bar is" />
        <foreach array="foreachArray1" index="foreachIndex1" item="foreachItem1">
          <script>
            counter.indexSum = counter.indexSum + foreachIndex1;
            counter.itemSum = counter.itemSum + foreachItem1;
          </script>
          <log expr="counter.indexSum" label="foreach counter.indexSum is" />
          <log expr="counter.itemSum" label="foreach counter.itemSum is" />
        </foreach>
        <raise event="if.choosen" />
        <elseif cond="ifVar1.bar == 'baz'" />
        <log label="elseif choosen" />
        <log expr="ifVar1.bar" label="ifVar1.bar is" />
        <assign expr="3" location="ifVar1.foo" />
        <send event="elseif.choosen" namelist="sendVar1">
          <param expr="sendVar1 + 16" name="foo" />
          <param expr="'a string literal'" name="bar" />
        </send>
        <else />
        <log label="else choosen" />
        <log expr="ifVar1.foo" label="ifVar1.foo is" />
        <log expr="ifVar1.bar" label="ifVar1.bar is" />
        <raise event="else.choosen" />
      </if>
      <script>parallelVar1++</script>
    </onentry>
    <transition target="active:{s0,p0,p0.s0,p0.s0.s0,p0.s1}" />
  </state>

  <!-- Global State  -->
  <state step="1" id="active:{s0,p0,p0.s0,p0.s0.s0,p0.s1}">
    <transition members="11                        "
                event="else.choosen"
                target="active:{s0,p0,p0.s0}-via-1" />
    <transition members="   10                     "
                event="elseif.choosen"
                cond="_event.data.foo == 20 &amp;&amp;
                      _event.data.sendVar1 == 4 &amp;&amp;
                      _event.data.bar == 'a string literal'"
                target="active:{s0,p0,p0.s0}-via-4" />
    <transition members="          7               "
                event="if.choosen"
                cond="counter.itemSum == 6 &amp;&amp; counter.indexSum == 3"
                target="active:{s0,p0,p0.s0}-via-6" />
  </state>
  <state transient="true" id="active:{s0,p0,p0.s0}-via-1">
    <onexit>
      <script>parallelVar1++</script>
    </onexit>
    <transition target="active:{s0}-via-2" />
  </state>
  <state transient="true" id="active:{s0}-via-2">
    <onexit>
      <assign expr="'baz'" location="ifVar1.bar" />
    </onexit>
```

```
    <transition target="active:{s0,p0,p0.s0,p0.s0.s0}-via-3" />
  </state>
  <state transient="true" id="active:{s0,p0,p0.s0,p0.s0.s0}-via-3">
    <onentry>
      <if cond="ifVar1.foo == 3">
        <log label="if choosen" />
        <log expr="ifVar1.bar" label="ifVar1.bar is" />
        <foreach
          array="foreachArray1"
          index="foreachIndex1"
          item="foreachItem1">
          <script>
            counter.indexSum = counter.indexSum + foreachIndex1;
            counter.itemSum = counter.itemSum + foreachItem1;
          </script>
          <log expr="counter.indexSum" label="foreach counter.indexSum is" />
          <log expr="counter.itemSum" label="foreach counter.itemSum is" />
        </foreach>
        <raise event="if.choosen" />
        <elseif cond="ifVar1.bar == 'baz'" />
        <log label="elseif choosen" />
        <log expr="ifVar1.bar" label="ifVar1.bar is" />
        <assign expr="3" location="ifVar1.foo" />
        <send event="elseif.choosen" namelist="sendVar1">
          <param expr="sendVar1 + 16" name="foo" />
          <param expr="'a string literal'" name="bar" />
        </send>
        <else />
        <log label="else choosen" />
        <log expr="ifVar1.foo" label="ifVar1.foo is" />
        <log expr="ifVar1.bar" label="ifVar1.bar is" />
        <raise event="else.choosen" />
      </if>
      <script>parallelVar1++</script>
    </onentry>
    <transition target="active:{s0,p0,p0.s0,p0.s0.s0,p0.s1}" />
  </state>
  <state transient="true" id="active:{s0,p0,p0.s0}-via-4">
    <onexit>
      <script>parallelVar1++</script>
    </onexit>
    <transition target="active:{s0,p0,p0.s0,p0.s0.s0}-via-5" />
  </state>
  <state transient="true" id="active:{s0,p0,p0.s0,p0.s0.s0}-via-5">
    <onentry>
      <if cond="ifVar1.foo == 3">
        <log label="if choosen" />
        <log expr="ifVar1.bar" label="ifVar1.bar is" />
        <foreach
          array="foreachArray1"
          index="foreachIndex1"
          item="foreachItem1">
          <script>
            counter.indexSum = counter.indexSum + foreachIndex1;
            counter.itemSum = counter.itemSum + foreachItem1;
          </script>
          <log expr="counter.indexSum" label="foreach counter.indexSum is" />
          <log expr="counter.itemSum" label="foreach counter.itemSum is" />
        </foreach>
        <raise event="if.choosen" />
        <elseif cond="ifVar1.bar == 'baz'" />
        <log label="elseif choosen" />
        <log expr="ifVar1.bar" label="ifVar1.bar is" />
        <assign expr="3" location="ifVar1.foo" />
        <send event="elseif.choosen" namelist="sendVar1">
          <param expr="sendVar1 + 16" name="foo" />
          <param expr="'a string literal'" name="bar" />
        </send>
        <else />
        <log label="else choosen" />
        <log expr="ifVar1.foo" label="ifVar1.foo is" />
        <log expr="ifVar1.bar" label="ifVar1.bar is" />
        <raise event="else.choosen" />
      </if>
      <script>parallelVar1++</script>
    </onentry>
    <transition target="active:{s0,p0,p0.s0,p0.s0.s0,p0.s1}" />
  </state>
  <state transient="true" id="active:{s0,p0,p0.s0}-via-6">
    <onexit>
      <script>parallelVar1++</script>
    </onexit>
    <transition target="active:{s0,p0,p0.s0,p0.s0.s1}-via-7" />
  </state>
```

```
<state transient="true" id="active:{s0,p0,p0.s0,p0.s0.s1}-via-7">
  <onentry>
    <if cond="_x.states['p0'] &amp;&amp; histVar1 == 1">
      <raise event="to.s2" />
      <else />
      <raise event="to.s1" />
    </if>
  </onentry>
  <transition target="active:{s0,p0,p0.s0,p0.s0.s1,p0.s1}" />
</state>

<!-- Global State  -->
<state step="2" id="active:{s0,p0,p0.s0,p0.s0.s1,p0.s1}">
  <transition members="      9                  "
              event="to.s2"
              target="active:{s0,p0,p0.s0}-via-8" />
  <transition members="        8                "
              event="to.s1"
              target="active:{s0,p0,p0.s0}-via-10" />
  <transition members="          7              "
              event="if.choosen"
              cond="counter.itemSum == 6 &amp;&amp; counter.indexSum == 3"
              target="active:{s0,p0,p0.s0}-via-12" />
</state>
<state transient="true" id="active:{s0,p0,p0.s0}-via-8">
  <onexit>
    <script>parallelVar1++</script>
  </onexit>
  <transition target="active:{s2}-via-9" />
</state>
<state transient="true" id="active:{s2}-via-9">
  <onentry>
    <send delay="3000"
          event="cancel.delayed"
          sendid="cancel.delayed" />
    <cancel sendid="cancel.delayed" />
  </onentry>
  <transition target="active:{s2,s2.s0};history:{s0.h0:{p0.s0.s1,p0.s1}}" />
</state>
<state transient="true" id="active:{s0,p0,p0.s0}-via-10">
  <onexit>
    <script>parallelVar1++</script>
  </onexit>
  <transition target="active:{s1}-via-11" />
</state>
<state transient="true" id="active:{s1}-via-11">
  <onentry>
    <send delay="1000" event="trigger.child" />
  </onentry>
  <invoke
    id="03fc22f6-847c-4972-8ebf-31fe0112b0fc"
    parent="s1"
    persist="true"
    autoforward="true"
    type="scxml">
    <content>
      <scxml datamodel="promela">
        <state
          index="8"
          id="waitForEvent">
          <transition event="trigger.child">
            <send event="back.to.history"
                  target="#_parent" />
          </transition>
        </state>
      </scxml>
    </content>
    <finalize>
      <script>finalizeVar1++;</script>
    </finalize>
  </invoke>
  <transition target="active:{s1};history:{s0.h0:{p0.s0.s1,p0.s1}}" />
</state>
<state transient="true" id="active:{s0,p0,p0.s0}-via-12">
  <onexit>
    <script>parallelVar1++</script>
  </onexit>
  <transition target="active:{s0,p0,p0.s0,p0.s0.s1}-via-13" />
</state>
<state transient="true" id="active:{s0,p0,p0.s0,p0.s0.s1}-via-13">
  <onentry>
    <if cond="_x.states['p0'] &amp;&amp; histVar1 == 1">
      <raise event="to.s2" />
      <else />
```

```
246         <raise event="to.s1" />
247       </if>
248     </onentry>
249     <transition target="active:{s0,p0,p0.s0,p0.s0.s1,p0.s1}" />
250   </state>
251
252   <!-- Global State  -->
253   <state step="3" id="active:{s2,s2.s0};history:{s0.h0:{p0.s0.s1,p0.s1}}">
254     <transition members="              5          "
255                 target="active:{s2,s2.s1}-via-14" />
256     <transition members="                 4       "
257                 event="done.state.s2"
258                 cond="_event.data.Var1 == 'foo'"
259                 target="active:{}-via-15" />
260     <transition members="                  3      "
261                 event="done.state.s2"
262                 target="active:{fail}-via-17" />
263     <transition members="                    2    "
264                 event="cancel.delayed"
265                 target="active:{fail}-via-18" />
266   </state>
267   <state transient="true" id="active:{s2,s2.s1}-via-14">
268     <onentry>
269       <raise event="done.state.s2">
270         <param expr="'foo'"
271           name="Var1" />
272       </raise>
273     </onentry>
274     <transition target="active:{s2,s2.s1};history:{s0.h0:{p0.s0.s1,p0.s1}}" />
275   </state>
276   <state transient="true" id="active:{}-via-15">
277     <onexit>
278       <assign expr="8" location="histVar1" />
279     </onexit>
280     <transition target="active:{s3}-via-16" />
281   </state>
282   <state transient="true" id="active:{s3}-via-16">
283     <onexit>
284       <log label="history transition" />
285       <assign expr="4" location="histVar1" />
286     </onexit>
287     <transition target="active:{s3,s3.s1};history:{s0.h0:{p0.s0.s1,p0.s1}}" />
288   </state>
289   <state transient="true" id="active:{fail}-via-17">
290     <onentry>
291       <log expr="'fail'" label="Outcome" />
292     </onentry>
293     <transition target="active:{fail};history:{s0.h0:{p0.s0.s1,p0.s1}}" />
294   </state>
295   <state transient="true" id="active:{fail}-via-18">
296     <onentry>
297       <log expr="'fail'" label="Outcome" />
298     </onentry>
299     <transition target="active:{fail};history:{s0.h0:{p0.s0.s1,p0.s1}}" />
300   </state>
301
302   <!-- Global State  -->
303   <state step="4" id="active:{s1};history:{s0.h0:{p0.s0.s1,p0.s1}}">
304     <transition members="            6            "
305                 event="back.to.history"
306                 cond="finalizeVar1 == 1"
307                 target="active:{}-via-19" />
308   </state>
309   <state transient="true" id="active:{}-via-19">
310     <uninvoke type="scxml" id="03fc22f6-847c-4972-8ebf-31fe0112b0fc" />
311     <onexit>
312       <assign expr="4-3" location="histVar1" />
313     </onexit>
314     <transition target="active:{s0,p0,p0.s0,p0.s0.s1}-via-20" />
315   </state>
316   <state transient="true" id="active:{s0,p0,p0.s0,p0.s0.s1}-via-20">
317     <onentry>
318       <if cond="_x.states['p0'] &amp;amp;&amp;amp; histVar1 == 1">
319         <raise event="to.s2" />
320         <else />
321         <raise event="to.s1" />
322       </if>
323     </onentry>
324     <transition target="active:{s0,p0,p0.s0,p0.s0.s1,p0.s1};history:{s0.h0:{p0.s0.s1,p0.s1}}" />
325   </state>
326
327   <!-- Global State  -->
328   <state step="5" id="active:{s2,s2.s1};history:{s0.h0:{p0.s0.s1,p0.s1}}">
329     <transition members="                4       "
```

```
330                 event="done.state.s2"
331                 cond="_event.data.Var1 == 'foo'"
332                 target="active:{}-via-21" />
333     <transition members="                  3      "
334                 event="done.state.s2"
335                 target="active:{fail}-via-23" />
336     <transition members="                    2    "
337                 event="cancel.delayed"
338                 target="active:{fail}-via-24" />
339     <transition members="                     1  "
340                 cond="histVar1 == 8"
341                 target="active:{pass}-via-25" />
342   </state>
343   <state transient="true" id="active:{}-via-21">
344     <onexit>
345       <assign expr="8" location="histVar1" />
346     </onexit>
347     <transition target="active:{s3}-via-22" />
348   </state>
349   <state transient="true" id="active:{s3}-via-22">
350     <onexit>
351       <log label="history transition" />
352       <assign expr="4" location="histVar1" />
353     </onexit>
354     <transition target="active:{s3,s3.s1};history:{s0.h0:{p0.s0.s1,p0.s1}}" />
355   </state>
356   <state transient="true" id="active:{fail}-via-23">
357     <onentry>
358       <log expr="'fail'" label="Outcome" />
359     </onentry>
360     <transition target="active:{fail};history:{s0.h0:{p0.s0.s1,p0.s1}}" />
361   </state>
362   <state transient="true" id="active:{fail}-via-24">
363     <onentry>
364       <log expr="'fail'" label="Outcome" />
365     </onentry>
366     <transition target="active:{fail};history:{s0.h0:{p0.s0.s1,p0.s1}}" />
367   </state>
368   <state transient="true" id="active:{pass}-via-25">
369     <onentry>
370       <log expr="'pass'" label="Outcome" />
371     </onentry>
372     <transition target="active:{pass};history:{s0.h0:{p0.s0.s1,p0.s1}}" />
373   </state>
374
375   <!-- Global State  -->
376   <state step="6" id="active:{s3,s3.s1};history:{s0.h0:{p0.s0.s1,p0.s1}}">
377     <transition members="                        0"
378                 target="active:{s2}-via-26" />
379   </state>
380   <state transient="true" id="active:{s2}-via-26">
381     <onentry>
382       <send delay="3000"
383                   event="cancel.delayed"
384         sendid="cancel.delayed" />
385       <cancel sendid="cancel.delayed" />
386     </onentry>
387     <transition target="active:{s2,s2.s0};history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}" />
388   </state>
389
390   <!-- Global State  -->
391   <state step="7" id="active:{fail};history:{s0.h0:{p0.s0.s1,p0.s1}}" final="true" />
392
393   <!-- Global State  -->
394   <state step="8" id="active:{s0,p0,p0.s0,p0.s0.s1,p0.s1};history:{s0.h0:{p0.s0.s1,p0.s1}}">
395     <transition members="      9                  "
396                 event="to.s2"
397                 target="active:{s0,p0,p0.s0}-via-27" />
398     <transition members="        8                "
399                 event="to.s1"
400                 target="active:{s0,p0,p0.s0}-via-29" />
401     <transition members="          7              "
402                 event="if.choosen"
403                 cond="counter.itemSum == 6 &amp;amp;&amp;amp; counter.indexSum == 3"
404                 target="active:{s0,p0,p0.s0}-via-31" />
405   </state>
406   <state transient="true" id="active:{s0,p0,p0.s0}-via-27">
407     <onexit>
408       <script>parallelVar1++</script>
409     </onexit>
410     <transition target="active:{s2}-via-28" />
411   </state>
412   <state transient="true" id="active:{s2}-via-28">
413     <onentry>
```

```
      <send delay="3000"
                event="cancel.delayed"
        sendid="cancel.delayed" />
      <cancel sendid="cancel.delayed" />
    </onentry>
    <transition target="active:{s2,s2.s0};history:{s0.h0:{p0.s0.s1,p0.s1}}" />
  </state>
  <state transient="true" id="active:{s0,p0,p0.s0}-via-29">
    <onexit>
      <script>parallelVar1++</script>
    </onexit>
    <transition target="active:{s1}-via-30" />
  </state>
  <state transient="true" id="active:{s1}-via-30">
    <onentry>
      <send delay="1000" event="trigger.child" />
    </onentry>
    <invoke
      id="03fc22f6-847c-4972-8ebf-31fe0112b0fc"
      parent="s1"
      persist="true"
      autoforward="true"
      type="scxml">
      <content>
        <scxml datamodel="promela">
          <state
            index="8"
            id="waitForEvent">
            <transition event="trigger.child">
              <send event="back.to.history" target="#_parent" />
            </transition>
          </state>
        </scxml>
      </content>
      <finalize>
        <script>finalizeVar1++;</script>
      </finalize>
    </invoke>
    <transition target="active:{s1};history:{s0.h0:{p0.s0.s1,p0.s1}}" />
  </state>
  <state transient="true" id="active:{s0,p0,p0.s0}-via-31">
    <onexit>
      <script>parallelVar1++</script>
    </onexit>
    <transition target="active:{s0,p0,p0.s0,p0.s0.s1}-via-32" />
  </state>
  <state transient="true" id="active:{s0,p0,p0.s0,p0.s0.s1}-via-32">
    <onentry>
      <if cond="_x.states['p0'] &amp;amp;&amp;amp; histVar1 == 1">
        <raise event="to.s2" />
        <else />
        <raise event="to.s1" />
      </if>
    </onentry>
    <transition target="active:{s0,p0,p0.s0,p0.s0.s1,p0.s1};history:{s0.h0:{p0.s0.s1,p0.s1}}" />
  </state>

  <!-- Global State  -->
  <state step="9" id="active:{pass};history:{s0.h0:{p0.s0.s1,p0.s1}}" final="true" />

  <!-- Global State  -->
  <state step="10" id="active:{s2,s2.s0};history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}">
    <transition members="              5          "
                target="active:{s2,s2.s1}-via-33" />
    <transition members="                4        "
                event="done.state.s2"
                cond="_event.data.Var1 == 'foo'"
                target="active:{}-via-34" />
    <transition members="                  3      "
                event="done.state.s2"
                target="active:{fail}-via-35" />
    <transition members="                    2    "
                event="cancel.delayed"
                target="active:{fail}-via-36" />
  </state>
  <state transient="true" id="active:{s2,s2.s1}-via-33">
    <onentry>
      <raise event="done.state.s2">
        <param expr="'foo'"
          name="Var1" />
      </raise>
    </onentry>
    <transition target="active:{s2,s2.s1};history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}" />
  </state>
```

```
498   <state transient="true" id="active:{}-via-34">
499     <onexit>
500       <assign expr="8" location="histVar1" />
501     </onexit>
502     <transition target="active:{s3,s3.s1};history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}" />
503   </state>
504   <state transient="true" id="active:{fail}-via-35">
505     <onentry>
506       <log expr="'fail'" label="Outcome" />
507     </onentry>
508     <transition target="active:{fail};history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}" />
509   </state>
510   <state transient="true" id="active:{fail}-via-36">
511     <onentry>
512       <log expr="'fail'" label="Outcome" />
513     </onentry>
514     <transition target="active:{fail};history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}" />
515   </state>
516
517   <!-- Global State  -->
518   <state step="11" id="active:{s2,s2.s1};history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}">
519     <transition members="               4        "
520                 event="done.state.s2"
521                 cond="_event.data.Var1 == 'foo'"
522                 target="active:{}-via-37" />
523     <transition members="                3      "
524                 event="done.state.s2"
525                 target="active:{fail}-via-38" />
526     <transition members="                  2    "
527                 event="cancel.delayed"
528                 target="active:{fail}-via-39" />
529     <transition members="                   1  "
530                 cond="histVar1 == 8"
531                 target="active:{pass}-via-40" />
532   </state>
533   <state transient="true" id="active:{}-via-37">
534     <onexit>
535       <assign expr="8" location="histVar1" />
536     </onexit>
537     <transition target="active:{s3,s3.s1};history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}" />
538   </state>
539   <state transient="true" id="active:{fail}-via-38">
540     <onentry>
541       <log expr="'fail'" label="Outcome" />
542     </onentry>
543     <transition target="active:{fail};history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}" />
544   </state>
545   <state transient="true" id="active:{fail}-via-39">
546     <onentry>
547       <log expr="'fail'" label="Outcome" />
548     </onentry>
549     <transition target="active:{fail};history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}" />
550   </state>
551   <state transient="true" id="active:{pass}-via-40">
552     <onentry>
553       <log expr="'pass'" label="Outcome" />
554     </onentry>
555     <transition target="active:{pass};history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}" />
556   </state>
557
558   <!-- Global State  -->
559   <state step="12" id="active:{s3,s3.s1};history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}">
560     <transition members="                        0"
561                 target="active:{s2}-via-41" />
562   </state>
563   <state transient="true" id="active:{s2}-via-41">
564     <onentry>
565       <send delay="3000" event="cancel.delayed" sendid="cancel.delayed" />
566       <cancel sendid="cancel.delayed" />
567     </onentry>
568     <transition target="active:{s2,s2.s0};history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}" />
569   </state>
570
571   <!-- Global State  -->
572   <state step="13" id="active:{fail};history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}" final="true" />
573
574   <!-- Global State  -->
575   <state step="14" id="active:{pass};history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}" final="true" />
576 </scxml>
```

Listing A.2: Example SCXML state-chart document from listing A.1 as an SCXML state-machine.

Finally, listing A.3 shows the complete PROMELA program for the SCXML state-machine in listing A.2 as it can be used with the SPIN model-checker. As semantic equivalence between the state-chart and state-machine representation was shown in section 5.3.7, this program can ultimately be used to ascertain temporal properties of the original state-chart in listing A.1.

```
/* event name identifiers */
#define BACK_TO_HISTORY 13      /* from "back.to.history" */
#define CANCEL_DELAYED 14       /* from "cancel.delayed" */
#define DONE_INVOKE_INV_E25D0 6 /* from "done.invoke.INV_e25d0" */
#define DONE_STATE_P0 5         /* from "done.state.p0" */
#define DONE_STATE_P0_S0 2      /* from "done.state.p0.s0" */
#define DONE_STATE_S0 1         /* from "done.state.s0" */
#define DONE_STATE_S2 3         /* from "done.state.s2" */
#define DONE_STATE_S3 4         /* from "done.state.s3" */
#define ELSE_CHOOSEN 7          /* from "else.choosen" */
#define ELSEIF_CHOOSEN 8        /* from "elseif.choosen" */
#define IF_CHOOSEN 11           /* from "if.choosen" */
#define TO_S1 10                /* from "to.s1" */
#define TO_S2 9                 /* from "to.s2" */
#define TRIGGER_CHILD 12        /* from "trigger.child" */

/* state name identifiers */
#define s6 6 /* from "active:{fail}" */
#define s7 7 /* from "active:{pass}" */
#define s0 0 /* from "active:{s0,p0,p0.s0,p0.s0.s0,p0.s1}" */
#define s1 1 /* from "active:{s0,p0,p0.s0,p0.s0.s1,p0.s1}" */
#define s3 3 /* from "active:{s1}" */
#define s2 2 /* from "active:{s2,s2.s0}" */
#define s4 4 /* from "active:{s2,s2.s1}" */
#define s5 5 /* from "active:{s3,s3.s1}" */

/* string literals */
#define INV_E25D0 1            /* INV_e25d0 */
#define INV_E25D0__NAME 10     /* INV_e25d0__name */
#define INV_E25D0__SESSIONID 9 /* INV_e25d0__sessionid */
#define MAIN__NAME 4           /* MAIN__name */
#define MAIN__SESSIONID 3      /* MAIN__sessionid */
#define A_STRING_LITERAL 5     /* a string literal */
#define BAZ 2                  /* baz */
#define FOO 7                  /* foo */

/* original state names */
#define FAIL 13         /* from "fail" */
#define P0 10           /* from "p0" */
#define P0_S0 1         /* from "p0.s0" */
#define P0_S0_S0 2      /* from "p0.s0.s0" */
#define P0_S0_S1 3      /* from "p0.s0.s1" */
#define P0_S1 4         /* from "p0.s1" */
#define PASS 12         /* from "pass" */
#define S0 0            /* from "s0" */
#define S0_H0 14        /* from "s0.h0" */
#define S1 5            /* from "s1" */
#define S2 6            /* from "s2" */
#define S2_S0 7         /* from "s2.s0" */
#define S2_S1 11        /* from "s2.s1" */
#define S3 8            /* from "s3" */
#define S3_H0 15        /* from "s3.h0" */
#define S3_S1 9         /* from "s3.s1" */
#define WAITFOREVENT 16 /* from "waitForEvent" */

/* history assignments for s0.h0
   4: p0
   0: p0.s0
   1: p0.s0.s0
   2: p0.s0.s1
   3: p0.s1
*/
bool MAIN__hist_s0_h0[5];
/* history assignments for s3.h0
   0: s3.s1
*/
bool MAIN__hist_s3_h0[1];

/* type definitions */
typedef _event_data_t {
  int Var1;
  int bar;
  int foo;
  int sendVar1;
};

typedef ifVar1_t {
  int bar;
```

```
  byte foo;
};

typedef counter_t {
  int indexSum;
  int itemSum;
};

typedef _x_t {
  bool states[17];
};

typedef _event_t {
  int delay;
  int name;
  int invokeid;
  _event_data_t data;
};


/* global variables for MAIN_ */
_event_t MAIN__event;               /* current event */
unsigned MAIN_s : 4;                /* current state */
chan MAIN_iQ   = [7] of {_event_t}  /* internal queue */
chan MAIN_eQ   = [8] of {_event_t}  /* external queue */
chan MAIN_start = [1] of {int}      /* nested machines to start at next macrostep */
hidden int MAIN_procid;             /* the process id running this machine */
bool MAIN_spontaneous;              /* whether to take spontaneous transitions */
bool MAIN_done;                     /* is the state machine stopped? */
bool MAIN_canceled;                 /* is the state machine canceled? */
_x_t MAIN__x;


/* data model variables for MAIN_ */
int MAIN_foreachArray1[3];
int MAIN_parallelVar1;
ifVar1_t MAIN_ifVar1;
counter_t MAIN_counter;
int MAIN_sendVar1;
int MAIN_histVar1;
int MAIN_finalizeVar1;
hidden int MAIN_foreachIndex1;
hidden int MAIN_foreachItem1;


/* global variables for INV_e25d0_ */
_event_t INV_e25d0__event;              /* current event */
unsigned INV_e25d0_s : 1;               /* current state */
chan INV_e25d0_iQ   = [1] of {_event_t}  /* internal queue */
chan INV_e25d0_eQ   = [7] of {_event_t}  /* external queue */
hidden int INV_e25d0_procid;            /* the process id running this machine */
bool INV_e25d0_spontaneous;             /* whether to take spontaneous transitions */
bool INV_e25d0_done;                    /* is the state machine stopped? */
bool INV_e25d0_canceled;                /* is the state machine canceled? */
_x_t INV_e25d0__x;


/* data model variables for INV_e25d0_ */



/* global inline functions */
hidden _event_t tmpE;
hidden int tmpIndex;
hidden _event_t _iwdQ[7];
hidden int _iwdQLength = 0;
hidden int _iwdIdx1 = 0;
hidden int _iwdIdx2 = 0;
hidden _event_t _iwdTmpE;
hidden _event_t _iwdLastE;
bool _iwdInserted = false;

/* last event in given queue is potentially at wrong position */
inline insertWithDelay(queue) {
  d_step {

    /* only process for non-trivial queues */
    if
    :: len(queue) > 1 -> {

      /* move all events but last over and remember the last one */
      _iwdIdx1 = 0;
      _iwdQLength = len(queue) - 1;

```

```
163       do
164       :: _iwdIdx1 < _iwdQLength -> {
165         queue?_iwdTmpE;
166         _iwdQ[_iwdIdx1].name = _iwdTmpE.name;
167         _iwdQ[_iwdIdx1].data.Var1 = _iwdTmpE.data.Var1;
168         _iwdQ[_iwdIdx1].data.bar = _iwdTmpE.data.bar;
169         _iwdQ[_iwdIdx1].data.foo = _iwdTmpE.data.foo;
170         _iwdQ[_iwdIdx1].data.sendVar1 = _iwdTmpE.data.sendVar1;
171         _iwdQ[_iwdIdx1].delay = _iwdTmpE.delay;
172         _iwdQ[_iwdIdx1].invokeid = _iwdTmpE.invokeid;
173         _iwdIdx1++;
174       }
175       :: else -> break;
176       od
177
178       queue?_iwdLastE;
179
180       /* _iwdQ now contains all but last item in _iwdLastE */
181       assert(len(queue) == 0);
182
183       /* reinsert into queue and place _iwdLastE correctly */
184       _iwdInserted = false;
185       _iwdIdx2 = 0;
186
187       do
188       :: _iwdIdx2 < _iwdIdx1 -> {
189         _iwdTmpE.name = _iwdQ[_iwdIdx2].name;
190         _iwdTmpE.data.Var1 = _iwdQ[_iwdIdx2].data.Var1;
191         _iwdTmpE.data.bar = _iwdQ[_iwdIdx2].data.bar;
192         _iwdTmpE.data.foo = _iwdQ[_iwdIdx2].data.foo;
193         _iwdTmpE.data.sendVar1 = _iwdQ[_iwdIdx2].data.sendVar1;
194         _iwdTmpE.delay = _iwdQ[_iwdIdx2].delay;
195         _iwdTmpE.invokeid = _iwdQ[_iwdIdx2].invokeid;
196
197         if
198         :: _iwdTmpE.delay > _iwdLastE.delay -> {
199           queue!_iwdLastE;
200           _iwdInserted = true;
201         }
202         :: else -> skip
203         fi;
204
205         queue!_iwdTmpE;
206         _iwdIdx2++;
207       }
208       :: else -> break;
209       od
210
211       if
212       :: !_iwdInserted -> queue!_iwdLastE;
213       :: else -> skip;
214       fi;
215
216     }
217     :: else -> skip;
218     fi;
219   }
220 }
221
222 inline determineSmallestDelay(smallestDelay, queue) {
223   if
224   :: len(queue) > 0 -> {
225     queue?<tmpE>;
226     if
227     :: (tmpE.delay < smallestDelay) -> { smallestDelay = tmpE.delay; }
228     :: else -> skip;
229     fi;
230   }
231   :: else -> skip;
232   fi;
233 }
234
235 inline advanceTime(increment, queue) {
236   tmpIndex = 0;
237   do
238   :: tmpIndex < len(queue) -> {
239     queue?tmpE;
240     if
241     :: tmpE.delay >= increment -> tmpE.delay = tmpE.delay - increment;
242     :: else -> skip;
243     fi
244     queue!tmpE;
245     tmpIndex++;
246   }
```

```
247   :: else -> break;
248   od
249 }
250
251 inline rescheduleProcess(smallestDelay, procId, internalQ, externalQ) {
252   set_priority(procId, 1);
253   if
254   :: len(internalQ) > 0 -> set_priority(procId, 10);
255   :: else {
256     if
257     :: len(externalQ) > 0 -> {
258       externalQ?<tmpE>;
259       if
260       :: smallestDelay == tmpE.delay -> set_priority(procId, 10);
261       :: else -> skip;
262       fi;
263     }
264     :: else -> skip;
265     fi;
266   }
267   fi;
268 }
269
270 inline scheduleMachines() {
271   /* schedule state-machines with regard to their event's delay */
272   skip;
273   d_step {
274
275 /* determine smallest delay */
276     int smallestDelay = 2147483647;
277     determineSmallestDelay(smallestDelay, MAIN_eQ);
278     determineSmallestDelay(smallestDelay, INV_e25d0_eQ);
279
280 /* prioritize processes with lowest delay or internal events */
281     rescheduleProcess(smallestDelay, MAIN_procid, MAIN_iQ, MAIN_eQ);
282     rescheduleProcess(smallestDelay, INV_e25d0_procid, INV_e25d0_iQ, INV_e25d0_eQ);
283
284 /* advance time by subtracting the smallest delay from all event delays */
285     if
286     :: (smallestDelay > 0) -> {
287       advanceTime(smallestDelay, MAIN_eQ);
288       advanceTime(smallestDelay, INV_e25d0_eQ);
289     }
290     :: else -> skip;
291     fi;
292   }
293   set_priority(_pid, 10);
294
295 }
296
297 inline cancelSendId(sendIdentifier, invokerIdentifier) {
298   cancelSendIdOnQueue(sendIdentifier, MAIN_eQ, invokerIdentifier);
299   cancelSendIdOnQueue(sendIdentifier, INV_e25d0_eQ, invokerIdentifier);
300 }
301
302 inline cancelSendIdOnQueue(sendIdentifier, queue, invokerIdentifier) {
303   tmpIndex = 0;
304   do
305   :: tmpIndex < len(queue) -> {
306     queue?tmpE;
307     if
308     :: tmpE.invokeid != invokerIdentifier || tmpE.sendid != sendIdentifier || tmpE.delay == 0 -> queue!tmpE;
309     :: else -> skip;
310     fi
311     tmpIndex++;
312   }
313   :: else -> break;
314   od
315 }
316
317 inline removePendingEventsForInvoker(invokeIdentifier) {
318   removePendingEventsForInvokerOnQueue(invokeIdentifier, MAIN_eQ);
319   removePendingEventsForInvokerOnQueue(invokeIdentifier, INV_e25d0_eQ);
320 }
321
322 inline removePendingEventsForInvokerOnQueue(invokeIdentifier, queue) {
323   tmpIndex = 0;
324   do
325   :: tmpIndex < len(queue) -> {
326     queue?tmpE;
327     if
328     :: tmpE.delay == 0 || tmpE.invokeid != invokeIdentifier -> queue!tmpE;
329     :: else -> skip;
330     fi
```

```
    tmpIndex++;
  }
  :: else -> break;
  od
}



proctype MAIN_run() {
  d_step {
    MAIN_done = false;
    MAIN_canceled = false;
    MAIN_spontaneous = true;
    MAIN_procid = _pid;
  }

/* transition to initial state */

MAIN_t0: /* ########################
     from state:
 ----- on event: SPONTANEOUS --
       to state: s0, p0, p0.s0, p0.s0.s0, p0.s1 with no history
############################## */

  skip;
  d_step {
    MAIN__x.states[S0] = true;
    MAIN__x.states[P0] = true;
    MAIN__x.states[P0_S0] = true;
    MAIN__x.states[P0_S0_S0] = true;
/* executable content for entering state p0.s0.s0 */
    if
    :: (MAIN_ifVar1.foo == 3) -> {
      printf("if choosen");
      printf("ifVar1.bar is: %d", MAIN_ifVar1.bar);
      for (MAIN_foreachIndex1 in MAIN_foreachArray1) {
        MAIN_foreachItem1 = MAIN_foreachArray1[MAIN_foreachIndex1];
        MAIN_counter.indexSum = MAIN_counter.indexSum + MAIN_foreachIndex1;
        MAIN_counter.itemSum = MAIN_counter.itemSum + MAIN_foreachItem1;
        printf("foreach counter.indexSum is: %d", MAIN_counter.indexSum);
        printf("foreach counter.itemSum is: %d", MAIN_counter.itemSum);
      }
      {
        tmpE.data.Var1 = 0;
        tmpE.data.bar = 0;
        tmpE.data.foo = 0;
        tmpE.data.sendVar1 = 0;
        tmpE.delay = 0;
        tmpE.invokeid = 0;
        tmpE.name = IF_CHOOSEN;
        tmpE.delay = 0;
        MAIN_iQ!tmpE;
      }
    }
    :: else -> {
      if
      :: (MAIN_ifVar1.bar == BAZ) -> {
        printf("elseif choosen");
        printf("ifVar1.bar is: %d", MAIN_ifVar1.bar);
        MAIN_ifVar1.foo = 3;
        {
          tmpE.data.Var1 = 0;
          tmpE.data.bar = 0;
          tmpE.data.foo = 0;
          tmpE.data.sendVar1 = 0;
          tmpE.delay = 0;
          tmpE.invokeid = 0;
          tmpE.name = ELSEIF_CHOOSEN;
          tmpE.delay = 0;
          tmpE.data.foo = MAIN_sendVar1 + 16;
          tmpE.data.bar = A_STRING_LITERAL;
          tmpE.data.sendVar1 = MAIN_sendVar1;
          MAIN_eQ!tmpE;
          insertWithDelay(MAIN_eQ);
        }
      }
      :: else -> {
        printf("else choosen");
        printf("ifVar1.foo is: %d", MAIN_ifVar1.foo);
        printf("ifVar1.bar is: %d", MAIN_ifVar1.bar);
        {
          tmpE.data.Var1 = 0;
          tmpE.data.bar = 0;
          tmpE.data.foo = 0;
```

```
415           tmpE.data.sendVar1 = 0;
416           tmpE.delay = 0;
417           tmpE.invokeid = 0;
418           tmpE.name = ELSE_CHOOSEN;
419           tmpE.delay = 0;
420           MAIN_iQ!tmpE;
421         }
422       }
423       fi;
424     }
425     fi;
426     MAIN_parallelVar1++
427     MAIN__x.states[P0_S1] = true;
428 /* to state s0, p0, p0.s0, p0.s0.s0, p0.s1 */
429     MAIN_s = s0;
430   }
431   goto MAIN_microStep;
432
433
434 MAIN_t2: /* ########################
435      from state: s0, p0, p0.s0, p0.s0.s0, p0.s1
436  ----- on event: else.choosen --
437        to state: s0, p0, p0.s0, p0.s0.s0, p0.s1 with no history
438 ############################### */
439
440   skip;
441   d_step {
442     MAIN__x.states[P0_S1] = false;
443 /* executable content for exiting state p0.s1 */
444     MAIN_parallelVar1++
445     MAIN__x.states[P0_S0_S0] = false;
446     MAIN__x.states[P0_S0] = false;
447     MAIN__x.states[P0] = false;
448 /* executable content for transition */
449     MAIN_ifVar1.bar = BAZ;
450     MAIN__x.states[P0] = true;
451     MAIN__x.states[P0_S0] = true;
452     MAIN__x.states[P0_S0_S0] = true;
453 /* executable content for entering state p0.s0.s0 */
454     if
455     :: (MAIN_ifVar1.foo == 3) -> {
456       printf("if choosen");
457       printf("ifVar1.bar is: %d", MAIN_ifVar1.bar);
458       for (MAIN_foreachIndex1 in MAIN_foreachArray1) {
459         MAIN_foreachItem1 = MAIN_foreachArray1[MAIN_foreachIndex1];
460         MAIN_counter.indexSum = MAIN_counter.indexSum + MAIN_foreachIndex1;
461         MAIN_counter.itemSum = MAIN_counter.itemSum + MAIN_foreachItem1;
462         printf("foreach counter.indexSum is: %d", MAIN_counter.indexSum);
463         printf("foreach counter.itemSum is: %d", MAIN_counter.itemSum);
464       }
465       {
466         tmpE.data.Var1 = 0;
467         tmpE.data.bar = 0;
468         tmpE.data.foo = 0;
469         tmpE.data.sendVar1 = 0;
470         tmpE.delay = 0;
471         tmpE.invokeid = 0;
472         tmpE.name = IF_CHOOSEN;
473         tmpE.delay = 0;
474         MAIN_iQ!tmpE;
475       }
476     }
477     :: else -> {
478       if
479       :: (MAIN_ifVar1.bar == BAZ) -> {
480         printf("elseif choosen");
481         printf("ifVar1.bar is: %d", MAIN_ifVar1.bar);
482         MAIN_ifVar1.foo = 3;
483         {
484           tmpE.data.Var1 = 0;
485           tmpE.data.bar = 0;
486           tmpE.data.foo = 0;
487           tmpE.data.sendVar1 = 0;
488           tmpE.delay = 0;
489           tmpE.invokeid = 0;
490           tmpE.name = ELSEIF_CHOOSEN;
491           tmpE.delay = 0;
492           tmpE.data.foo = MAIN_sendVar1 + 16;
493           tmpE.data.bar = A_STRING_LITERAL;
494           tmpE.data.sendVar1 = MAIN_sendVar1;
495           MAIN_eQ!tmpE;
496           insertWithDelay(MAIN_eQ);
497         }
498       }
```

```
      :: else -> {
        printf("else choosen");
        printf("ifVar1.foo is: %d", MAIN_ifVar1.foo);
        printf("ifVar1.bar is: %d", MAIN_ifVar1.bar);
        {
          tmpE.data.Var1 = 0;
          tmpE.data.bar = 0;
          tmpE.data.foo = 0;
          tmpE.data.sendVar1 = 0;
          tmpE.delay = 0;
          tmpE.invokeid = 0;
          tmpE.name = ELSE_CHOOSEN;
          tmpE.delay = 0;
          MAIN_iQ!tmpE;
        }
      }
      fi;
    }
    fi;
    MAIN_parallelVar1++
    MAIN__x.states[P0_S1] = true;
/* to state s0, p0, p0.s0, p0.s0.s0, p0.s1 */
    MAIN_s = s0;
  }
  MAIN_spontaneous = true;
  goto MAIN_microStep;

MAIN_t3: /* ########################
     from state: s0, p0, p0.s0, p0.s0.s0, p0.s1
 ----- on event: elseif.choosen --
       to state: s0, p0, p0.s0, p0.s0.s0, p0.s1 with no history
############################## */

  skip;
  d_step {
    MAIN__x.states[P0_S1] = false;
/* executable content for exiting state p0.s1 */
    MAIN_parallelVar1++
    MAIN__x.states[P0_S0_S0] = false;
    MAIN__x.states[P0_S0] = false;
    MAIN__x.states[P0] = false;
    MAIN__x.states[P0] = true;
    MAIN__x.states[P0_S0] = true;
    MAIN__x.states[P0_S0_S0] = true;
/* executable content for entering state p0.s0.s0 */
    if
    :: (MAIN_ifVar1.foo == 3) -> {
      printf("if choosen");
      printf("ifVar1.bar is: %d", MAIN_ifVar1.bar);
      for (MAIN_foreachIndex1 in MAIN_foreachArray1) {
        MAIN_foreachItem1 = MAIN_foreachArray1[MAIN_foreachIndex1];
        MAIN_counter.indexSum = MAIN_counter.indexSum + MAIN_foreachIndex1;
        MAIN_counter.itemSum = MAIN_counter.itemSum + MAIN_foreachItem1;
        printf("foreach counter.indexSum is: %d", MAIN_counter.indexSum);
        printf("foreach counter.itemSum is: %d", MAIN_counter.itemSum);
      }
      {
        tmpE.data.Var1 = 0;
        tmpE.data.bar = 0;
        tmpE.data.foo = 0;
        tmpE.data.sendVar1 = 0;
        tmpE.delay = 0;
        tmpE.invokeid = 0;
        tmpE.name = IF_CHOOSEN;
        tmpE.delay = 0;
        MAIN_iQ!tmpE;
      }
    }
    :: else -> {
      if
      :: (MAIN_ifVar1.bar == BAZ) -> {
        printf("elseif choosen");
        printf("ifVar1.bar is: %d", MAIN_ifVar1.bar);
        MAIN_ifVar1.foo = 3;
        {
          tmpE.data.Var1 = 0;
          tmpE.data.bar = 0;
          tmpE.data.foo = 0;
          tmpE.data.sendVar1 = 0;
          tmpE.delay = 0;
          tmpE.invokeid = 0;
          tmpE.name = ELSEIF_CHOOSEN;
          tmpE.delay = 0;
          tmpE.data.foo = MAIN_sendVar1 + 16;
```

```
583           tmpE.data.bar = A_STRING_LITERAL;
584           tmpE.data.sendVar1 = MAIN_sendVar1;
585           MAIN_eQ!tmpE;
586           insertWithDelay(MAIN_eQ);
587         }
588       }
589       :: else -> {
590         printf("else choosen");
591         printf("ifVar1.foo is: %d", MAIN_ifVar1.foo);
592         printf("ifVar1.bar is: %d", MAIN_ifVar1.bar);
593         {
594           tmpE.data.Var1 = 0;
595           tmpE.data.bar = 0;
596           tmpE.data.foo = 0;
597           tmpE.data.sendVar1 = 0;
598           tmpE.delay = 0;
599           tmpE.invokeid = 0;
600           tmpE.name = ELSE_CHOOSEN;
601           tmpE.delay = 0;
602           MAIN_iQ!tmpE;
603         }
604       }
605       fi;
606     }
607     fi;
608     MAIN_parallelVar1++
609     MAIN__x.states[P0_S1] = true;
610 /* to state s0, p0, p0.s0, p0.s0.s0, p0.s1 */
611     MAIN_s = s0;
612   }
613   MAIN_spontaneous = true;
614   goto MAIN_microStep;
615
616 MAIN_t1: /* ########################
617      from state: s0, p0, p0.s0, p0.s0.s0, p0.s1
618  ----- on event: if.choosen --
619        to state: s0, p0, p0.s0, p0.s0.s1, p0.s1 with no history
620 ############################### */
621
622   skip;
623   d_step {
624     MAIN__x.states[P0_S1] = false;
625 /* executable content for exiting state p0.s1 */
626     MAIN_parallelVar1++
627     MAIN__x.states[P0_S0_S0] = false;
628     MAIN__x.states[P0_S0] = false;
629     MAIN__x.states[P0] = false;
630     MAIN__x.states[P0] = true;
631     MAIN__x.states[P0_S0] = true;
632     MAIN__x.states[P0_S0_S1] = true;
633 /* executable content for entering state p0.s0.s1 */
634     if
635     :: (MAIN__x.states[P0] && MAIN_histVar1 == 1) -> {
636       {
637         tmpE.data.Var1 = 0;
638         tmpE.data.bar = 0;
639         tmpE.data.foo = 0;
640         tmpE.data.sendVar1 = 0;
641         tmpE.delay = 0;
642         tmpE.invokeid = 0;
643         tmpE.name = T0_S2;
644         tmpE.delay = 0;
645         MAIN_iQ!tmpE;
646       }
647     }
648     :: else -> {
649       {
650         tmpE.data.Var1 = 0;
651         tmpE.data.bar = 0;
652         tmpE.data.foo = 0;
653         tmpE.data.sendVar1 = 0;
654         tmpE.delay = 0;
655         tmpE.invokeid = 0;
656         tmpE.name = T0_S1;
657         tmpE.delay = 0;
658         MAIN_iQ!tmpE;
659       }
660     }
661     fi;
662     MAIN__x.states[P0_S1] = true;
663 /* to state s0, p0, p0.s0, p0.s0.s1, p0.s1 */
664     MAIN_s = s1;
665   }
666   MAIN_spontaneous = true;
```

```
667   goto MAIN_microStep;
668
669 MAIN_t5: /* ########################
670      from state: s0, p0, p0.s0, p0.s0.s1, p0.s1
671  ----- on event: to.s2 --
672        to state: s2, s2.s0 with history:{s0.h0:{p0.s0.s1,p0.s1}}
673 ############################## */
674
675   skip;
676   d_step {
677     MAIN__hist_s0_h0[2] = 1;    /* p0.s0.s1 */
678     MAIN__hist_s0_h0[3] = 1;    /* p0.s1 */
679     MAIN__x.states[P0_S1] = false;
680 /* executable content for exiting state p0.s1 */
681     MAIN_parallelVar1++
682     MAIN__x.states[P0_S0_S1] = false;
683     MAIN__x.states[P0_S0] = false;
684     MAIN__x.states[P0] = false;
685     MAIN__x.states[S0] = false;
686     MAIN__x.states[S2] = true;
687 /* executable content for entering state s2 */
688     {
689       tmpE.data.Var1 = 0;
690       tmpE.data.bar = 0;
691       tmpE.data.foo = 0;
692       tmpE.data.sendVar1 = 0;
693       tmpE.delay = 0;
694       tmpE.invokeid = 0;
695       tmpE.name = CANCEL_DELAYED;
696       tmpE.delay = 3000;
697       MAIN_eQ!tmpE;
698       insertWithDelay(MAIN_eQ);
699     }
700     cancelSendId(CANCEL_DELAYED, 0);
701     MAIN__x.states[S2_S0] = true;
702 /* to state s2, s2.s0 */
703     MAIN_s = s2;
704   }
705   MAIN_spontaneous = true;
706   goto MAIN_microStep;
707
708 MAIN_t6: /* ########################
709      from state: s0, p0, p0.s0, p0.s0.s1, p0.s1
710  ----- on event: to.s1 --
711        to state: s1 with history:{s0.h0:{p0.s0.s1,p0.s1}}
712 ############################## */
713
714   skip;
715   d_step {
716     MAIN__hist_s0_h0[2] = 1;    /* p0.s0.s1 */
717     MAIN__hist_s0_h0[3] = 1;    /* p0.s1 */
718     MAIN__x.states[P0_S1] = false;
719 /* executable content for exiting state p0.s1 */
720     MAIN_parallelVar1++
721     MAIN__x.states[P0_S0_S1] = false;
722     MAIN__x.states[P0_S0] = false;
723     MAIN__x.states[P0] = false;
724     MAIN__x.states[S0] = false;
725     MAIN__x.states[S1] = true;
726 /* executable content for entering state s1 */
727     {
728       tmpE.data.Var1 = 0;
729       tmpE.data.bar = 0;
730       tmpE.data.foo = 0;
731       tmpE.data.sendVar1 = 0;
732       tmpE.delay = 0;
733       tmpE.invokeid = 0;
734       tmpE.name = TRIGGER_CHILD;
735       tmpE.delay = 1000;
736       MAIN_eQ!tmpE;
737       insertWithDelay(MAIN_eQ);
738     }
739     MAIN_start!INV_E25D0;
740 /* to state s1 */
741     MAIN_s = s3;
742   }
743   MAIN_spontaneous = true;
744   goto MAIN_microStep;
745
746 MAIN_t4: /* ########################
747      from state: s0, p0, p0.s0, p0.s0.s1, p0.s1
748  ----- on event: if.choosen --
749        to state: s0, p0, p0.s0, p0.s0.s1, p0.s1 with no history
750                  s0, p0, p0.s0, p0.s0.s1, p0.s1 with history:{s0.h0:{p0.s0.s1,p0.s1}}
```

```
751 ############################## */
752
753   skip;
754   d_step {
755     MAIN__x.states[P0_S1] = false;
756 /* executable content for exiting state p0.s1 */
757     MAIN_parallelVar1++
758     MAIN__x.states[P0_S0_S1] = false;
759     MAIN__x.states[P0_S0] = false;
760     MAIN__x.states[P0] = false;
761     MAIN__x.states[P0] = true;
762     MAIN__x.states[P0_S0] = true;
763     MAIN__x.states[P0_S0_S1] = true;
764 /* executable content for entering state p0.s0.s1 */
765     if
766     :: (MAIN__x.states[P0] && MAIN_histVar1 == 1) -> {
767       {
768         tmpE.data.Var1 = 0;
769         tmpE.data.bar = 0;
770         tmpE.data.foo = 0;
771         tmpE.data.sendVar1 = 0;
772         tmpE.delay = 0;
773         tmpE.invokeid = 0;
774         tmpE.name = T0_S2;
775         tmpE.delay = 0;
776         MAIN_iQ!tmpE;
777       }
778     }
779     :: else -> {
780       {
781         tmpE.data.Var1 = 0;
782         tmpE.data.bar = 0;
783         tmpE.data.foo = 0;
784         tmpE.data.sendVar1 = 0;
785         tmpE.delay = 0;
786         tmpE.invokeid = 0;
787         tmpE.name = T0_S1;
788         tmpE.delay = 0;
789         MAIN_iQ!tmpE;
790       }
791     }
792     fi;
793     MAIN__x.states[P0_S1] = true;
794 /* to state s0, p0, p0.s0, p0.s0.s1, p0.s1 */
795     MAIN_s = s1;
796   }
797   MAIN_spontaneous = true;
798   goto MAIN_microStep;
799
800 MAIN_t12: /* ########################
801      from state: s1
802  ----- on event: back.to.history --
803        to state: s0, p0, p0.s0, p0.s0.s1, p0.s1 with history:{s0.h0:{p0.s0.s1,p0.s1}}
804 ############################## */
805
806   skip;
807   d_step {
808     MAIN__x.states[S1] = false;
809     do
810     :: MAIN_start??INV_E25D0 -> skip
811     :: else -> break;
812     od
813     INV_e25d0_canceled = true;
814     removePendingEventsForInvoker(INV_E25D0);
815 /* executable content for transition */
816     MAIN_histVar1 = 4-3;
817     MAIN__x.states[S0] = true;
818     MAIN__x.states[P0] = true;
819     MAIN__x.states[P0_S0] = true;
820     MAIN__x.states[P0_S0_S1] = true;
821 /* executable content for entering state p0.s0.s1 */
822     if
823     :: (MAIN__x.states[P0] && MAIN_histVar1 == 1) -> {
824       {
825         tmpE.data.Var1 = 0;
826         tmpE.data.bar = 0;
827         tmpE.data.foo = 0;
828         tmpE.data.sendVar1 = 0;
829         tmpE.delay = 0;
830         tmpE.invokeid = 0;
831         tmpE.name = T0_S2;
832         tmpE.delay = 0;
833         MAIN_iQ!tmpE;
834       }
```

```
835     }
836     :: else -> {
837       {
838         tmpE.data.Var1 = 0;
839         tmpE.data.bar = 0;
840         tmpE.data.foo = 0;
841         tmpE.data.sendVar1 = 0;
842         tmpE.delay = 0;
843         tmpE.invokeid = 0;
844         tmpE.name = TO_S1;
845         tmpE.delay = 0;
846         MAIN_iQ!tmpE;
847       }
848     }
849     fi;
850     MAIN__x.states[P0_S1] = true;
851 /* to state s0, p0, p0.s0, p0.s0.s1, p0.s1 */
852     MAIN_s = s1;
853   }
854   MAIN_spontaneous = true;
855   goto MAIN_microStep;
856
857 MAIN_t11: /* ########################
858      from state: s2, s2.s0
859  ----- on event: SPONTANEOUS --
860        to state: s2, s2.s1 with history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}
861                  s2, s2.s1 with history:{s0.h0:{p0.s0.s1,p0.s1}}
862 ############################### */
863
864   skip;
865   d_step {
866     MAIN__x.states[S2_S0] = false;
867     MAIN__x.states[S2_S1] = true;
868     {
869       tmpE.data.Var1 = 0;
870       tmpE.data.bar = 0;
871       tmpE.data.foo = 0;
872       tmpE.data.sendVar1 = 0;
873       tmpE.delay = 0;
874       tmpE.invokeid = 0;
875       tmpE.name = DONE_STATE_S2;
876       tmpE.delay = 0;
877       tmpE.data.Var1 = FOO;
878       MAIN_iQ!tmpE;
879     }
880 /* to state s2, s2.s1 */
881     MAIN_s = s4;
882   }
883   goto MAIN_microStep;
884
885 MAIN_t7: /* ########################
886      from state: s2, s2.s0
887  ----- on event: done.state.s2 --
888        to state: s3, s3.s1 with history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}
889                  s3, s3.s1 with history:{s0.h0:{p0.s0.s1,p0.s1}}
890 ############################### */
891
892   skip;
893   d_step {
894     MAIN__x.states[S2_S0] = false;
895     MAIN__x.states[S2] = false;
896 /* executable content for transition */
897     MAIN_histVar1 = 8;
898     MAIN__x.states[S3] = true;
899     if
900     :: (MAIN__hist_s3_h0[0]) -> skip;
901     :: else -> {
902 /* executable content for transition */
903       printf("history transition");
904       MAIN_histVar1 = 4;
905     }
906     fi
907
908     MAIN__x.states[S3_S1] = true;
909 /* to state s3, s3.s1 */
910     MAIN_s = s5;
911   }
912   MAIN_spontaneous = true;
913   goto MAIN_microStep;
914
915 MAIN_t8: /* ########################
916      from state: s2, s2.s0
917  ----- on event: done.state.s2 --
918        to state: fail with history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}
```

```
                fail with history:{s0.h0:{p0.s0.s1,p0.s1}}
############################### */

  skip;
  d_step {
    MAIN__x.states[S2_S0] = false;
    MAIN__x.states[S2] = false;
    MAIN__x.states[FAIL] = true;
/* executable content for entering state fail */
    printf("Outcome: %d", FAIL);
/* to state fail */
    MAIN_s = s6;
  }
  goto MAIN_terminate;

MAIN_t9: /* ########################
     from state: s2, s2.s0
 ----- on event: cancel.delayed --
       to state: fail with history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}
                 fail with history:{s0.h0:{p0.s0.s1,p0.s1}}
############################### */

  skip;
  d_step {
    MAIN__x.states[S2_S0] = false;
    MAIN__x.states[S2] = false;
    MAIN__x.states[FAIL] = true;
/* executable content for entering state fail */
    printf("Outcome: %d", FAIL);
/* to state fail */
    MAIN_s = s6;
  }
  goto MAIN_terminate;

MAIN_t13: /* ########################
     from state: s2, s2.s1
 ----- on event: done.state.s2 --
       to state: s3, s3.s1 with history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}
                 s3, s3.s1 with history:{s0.h0:{p0.s0.s1,p0.s1}}
############################### */

  skip;
  d_step {
    MAIN__x.states[S2_S1] = false;
    MAIN__x.states[S2] = false;
/* executable content for transition */
    MAIN_histVar1 = 8;
    MAIN__x.states[S3] = true;
    if
    :: (MAIN__hist_s3_h0[0]) -> skip;
    :: else -> {
/* executable content for transition */
      printf("history transition");
      MAIN_histVar1 = 4;
    }
    fi

    MAIN__x.states[S3_S1] = true;
/* to state s3, s3.s1 */
    MAIN_s = s5;
  }
  MAIN_spontaneous = true;
  goto MAIN_microStep;

MAIN_t14: /* ########################
     from state: s2, s2.s1
 ----- on event: done.state.s2 --
       to state: fail with history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}
                 fail with history:{s0.h0:{p0.s0.s1,p0.s1}}
############################### */

  skip;
  d_step {
    MAIN__x.states[S2_S1] = false;
    MAIN__x.states[S2] = false;
    MAIN__x.states[FAIL] = true;
/* executable content for entering state fail */
    printf("Outcome: %d", FAIL);
/* to state fail */
    MAIN_s = s6;
  }
  goto MAIN_terminate;

MAIN_t15: /* ########################
```

```
1003        from state: s2, s2.s1
1004     ----- on event: cancel.delayed --
1005           to state: fail with history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}
1006                     fail with history:{s0.h0:{p0.s0.s1,p0.s1}}
1007    ############################### */
1008
1009      skip;
1010      d_step {
1011        MAIN__x.states[S2_S1] = false;
1012        MAIN__x.states[S2] = false;
1013        MAIN__x.states[FAIL] = true;
1014    /* executable content for entering state fail */
1015        printf("Outcome: %d", FAIL);
1016    /* to state fail */
1017        MAIN_s = s6;
1018      }
1019      goto MAIN_terminate;
1020
1021    MAIN_t16: /* ########################
1022         from state: s2, s2.s1
1023     ----- on event: SPONTANEOUS --
1024           to state: pass with history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}
1025                     pass with history:{s0.h0:{p0.s0.s1,p0.s1}}
1026    ############################### */
1027
1028      skip;
1029      d_step {
1030        MAIN__x.states[S2_S1] = false;
1031        MAIN__x.states[S2] = false;
1032        MAIN__x.states[PASS] = true;
1033    /* executable content for entering state pass */
1034        printf("Outcome: %d", PASS);
1035    /* to state pass */
1036        MAIN_s = s7;
1037      }
1038      goto MAIN_terminate;
1039
1040    MAIN_t17: /* ########################
1041         from state: s3, s3.s1
1042     ----- on event: SPONTANEOUS --
1043           to state: s2, s2.s0 with history:{s0.h0:{p0.s0.s1,p0.s1},s3.h0:{s3.s1}}
1044    ############################### */
1045
1046      skip;
1047      d_step {
1048        MAIN__hist_s3_h0[0] = 1;    /* s3.s1 */
1049        MAIN__x.states[S3_S1] = false;
1050        MAIN__x.states[S3] = false;
1051        MAIN__x.states[S2] = true;
1052    /* executable content for entering state s2 */
1053        {
1054          tmpE.data.Var1 = 0;
1055          tmpE.data.bar = 0;
1056          tmpE.data.foo = 0;
1057          tmpE.data.sendVar1 = 0;
1058          tmpE.delay = 0;
1059          tmpE.invokeid = 0;
1060          tmpE.name = CANCEL_DELAYED;
1061          tmpE.delay = 3000;
1062          MAIN_eQ!tmpE;
1063          insertWithDelay(MAIN_eQ);
1064        }
1065        cancelSendId(CANCEL_DELAYED, 0);
1066        MAIN__x.states[S2_S0] = true;
1067    /* to state s2, s2.s0 */
1068        MAIN_s = s2;
1069      }
1070      goto MAIN_microStep;
1071
1072    MAIN_macroStep: skip;
1073      /* start pending invokers */
1074      int invokerId;
1075      do
1076      :: MAIN_start?invokerId -> {
1077        if
1078        :: invokerId == INV_E25D0 -> {
1079          run INV_e25d0_run() priority 20;
1080        }
1081        :: else -> skip;
1082        fi
1083      }
1084      :: else -> break;
1085      od
1086
```

```
1087 /* Determine machines with smallest delay and set their process priority */
1088   scheduleMachines();
1089
1090   atomic {
1091 /* pop an event */
1092     if
1093     :: len(MAIN_iQ) != 0 -> MAIN_iQ ? MAIN__event   /* from internal queue */
1094     :: else -> MAIN_eQ ? MAIN__event                /* from external queue */
1095     fi;
1096
1097 /* terminate if we are stopped */
1098     if
1099     :: MAIN_done -> goto MAIN_terminate;
1100     :: else -> skip;
1101     fi;
1102
1103 /* <finalize> event */
1104     if
1105     :: MAIN__event.invokeid == INV_E25D0 -> {
1106       MAIN_finalizeVar1++;
1107     }
1108     :: else -> skip;
1109     fi;
1110
1111 /* autoforward event to INV_e25d0 invokers */
1112     if
1113     :: INV_e25d0_done -> skip;
1114     :: INV_e25d0_canceled -> skip;
1115     :: else -> { INV_e25d0_eQ!MAIN__event; insertWithDelay(INV_e25d0_eQ); }
1116     fi;
1117
1118
1119 MAIN_microStep:
1120 /* event dispatching per state */
1121     if
1122
1123 /* ### current state fail ######################### */
1124     :: (MAIN_s == s6) -> {
1125 /* no transition applicable */
1126       MAIN_spontaneous = false;
1127       goto MAIN_macroStep;
1128     }
1129
1130 /* ### current state pass ######################### */
1131     :: (MAIN_s == s7) -> {
1132 /* no transition applicable */
1133       MAIN_spontaneous = false;
1134       goto MAIN_macroStep;
1135     }
1136
1137 /* ### current state s0, p0, p0.s0, p0.s0.s0, p0.s1 ######################### */
1138     :: (MAIN_s == s0) -> {
1139       if
1140       :: (!MAIN_spontaneous && MAIN__event.name == ELSE_CHOOSEN) -> {
1141 /* transition to s0, p0, p0.s0, p0.s0.s0, p0.s1 */
1142         goto MAIN_t2;
1143       }
1144       :: else -> {
1145         if
1146         :: ((!MAIN_spontaneous && MAIN__event.name == ELSEIF_CHOOSEN) && (MAIN__event.data.foo == 20 && MAIN__event.data.sendVar1 == 4 &&
      MAIN__event.data.bar == A_STRING_LITERAL)) -> {
1147 /* transition to s0, p0, p0.s0, p0.s0.s0, p0.s1 */
1148           goto MAIN_t3;
1149         }
1150         :: else -> {
1151           if
1152           :: ((!MAIN_spontaneous && MAIN__event.name == IF_CHOOSEN) && (MAIN_counter.itemSum == 6 && MAIN_counter.indexSum == 3)) -> {
1153 /* transition to s0, p0, p0.s0, p0.s0.s1, p0.s1 */
1154             goto MAIN_t1;
1155           }
1156           :: else -> {
1157 /* no transition applicable */
1158             MAIN_spontaneous = false;
1159             goto MAIN_macroStep;
1160           }
1161           fi;
1162         }
1163         fi;
1164       }
1165       fi;
1166     }
1167
1168 /* ### current state s0, p0, p0.s0, p0.s0.s1, p0.s1 ######################### */
1169     :: (MAIN_s == s1) -> {
```

```
1170        if
1171        :: (!MAIN_spontaneous && MAIN__event.name == TO_S2) -> {
1172  /* transition to s2, s2.s0 */
1173          goto MAIN_t5;
1174        }
1175        :: else -> {
1176          if
1177          :: (!MAIN_spontaneous && MAIN__event.name == TO_S1) -> {
1178  /* transition to s1 */
1179            goto MAIN_t6;
1180          }
1181          :: else -> {
1182            if
1183            :: ((!MAIN_spontaneous && MAIN__event.name == IF_CHOOSEN) && (MAIN_counter.itemSum == 6 && MAIN_counter.indexSum == 3)) -> {
1184  /* transition to s0, p0, p0.s0, p0.s0.s1, p0.s1 */
1185              goto MAIN_t4;
1186            }
1187            :: else -> {
1188  /* no transition applicable */
1189              MAIN_spontaneous = false;
1190              goto MAIN_macroStep;
1191            }
1192            fi;
1193          }
1194          fi;
1195        }
1196        fi;
1197      }
1198
1199  /* ### current state s1 ######################### */
1200      :: (MAIN_s == s3) -> {
1201        if
1202        :: ((!MAIN_spontaneous && MAIN__event.name == BACK_TO_HISTORY) && (MAIN_finalizeVar1 == 1)) -> {
1203  /* transition to s0, p0, p0.s0, p0.s0.s1, p0.s1 */
1204          goto MAIN_t12;
1205        }
1206        :: else -> {
1207  /* no transition applicable */
1208          MAIN_spontaneous = false;
1209          goto MAIN_macroStep;
1210        }
1211        fi;
1212      }
1213
1214  /* ### current state s2, s2.s0 ######################### */
1215      :: (MAIN_s == s2) -> {
1216        if
1217        :: (MAIN_spontaneous) -> {
1218  /* transition to s2, s2.s1 */
1219          goto MAIN_t11;
1220        }
1221        :: else -> {
1222          if
1223          :: ((!MAIN_spontaneous && MAIN__event.name == DONE_STATE_S2) && (MAIN__event.data.Var1 == FOO)) -> {
1224  /* transition to s3, s3.s1 */
1225            goto MAIN_t7;
1226          }
1227          :: else -> {
1228            if
1229            :: (!MAIN_spontaneous && MAIN__event.name == DONE_STATE_S2) -> {
1230  /* transition to fail */
1231              goto MAIN_t8;
1232            }
1233            :: else -> {
1234              if
1235              :: (!MAIN_spontaneous && MAIN__event.name == CANCEL_DELAYED) -> {
1236  /* transition to fail */
1237                goto MAIN_t9;
1238              }
1239              :: else -> {
1240  /* no transition applicable */
1241                MAIN_spontaneous = false;
1242                goto MAIN_macroStep;
1243              }
1244              fi;
1245            }
1246            fi;
1247          }
1248          fi;
1249        }
1250        fi;
1251      }
1252
1253  /* ### current state s2, s2.s1 ######################### */
```

```
    :: (MAIN_s == s4) -> {
      if
      :: ((!MAIN_spontaneous && MAIN__event.name == DONE_STATE_S2) && (MAIN__event.data.Var1 == FOO)) -> {
/* transition to s3, s3.s1 */
        goto MAIN_t13;
      }
      :: else -> {
        if
        :: (!MAIN_spontaneous && MAIN__event.name == DONE_STATE_S2) -> {
/* transition to fail */
          goto MAIN_t14;
        }
        :: else -> {
          if
          :: (!MAIN_spontaneous && MAIN__event.name == CANCEL_DELAYED) -> {
/* transition to fail */
            goto MAIN_t15;
          }
          :: else -> {
            if
            :: ((MAIN_spontaneous) && (MAIN_histVar1 == 8)) -> {
/* transition to pass */
              goto MAIN_t16;
            }
            :: else -> {
/* no transition applicable */
              MAIN_spontaneous = false;
              goto MAIN_macroStep;
            }
            fi;
          }
          fi;
        }
        fi;
      }
      fi;
    }

/* ### current state s3, s3.s1 ######################### */
    :: (MAIN_s == s5) -> {
      if
      :: (MAIN_spontaneous) -> {
/* transition to s2, s2.s0 */
        goto MAIN_t17;
      }
      :: else -> {
/* no transition applicable */
        MAIN_spontaneous = false;
        goto MAIN_macroStep;
      }
      fi;
    }
/* this is an error as we dispatched all valid states */
    :: else -> assert(false);
    fi;

MAIN_terminate: skip;
  }
}


init {
/* initialize data model variables */
  MAIN_foreachArray1[0] = 1;
  MAIN_foreachArray1[1] = 2;
  MAIN_foreachArray1[2] = 3;
  MAIN_parallelVar1 = 0;
  MAIN_ifVar1.bar = BAZ;
  MAIN_ifVar1.foo = 1;
  MAIN_counter.indexSum = 0;
  MAIN_counter.itemSum = 0;
  MAIN_sendVar1 = 4;
  MAIN_histVar1 = 0;
  MAIN_finalizeVar1 = 0;

  run MAIN_run() priority 10;
}

proctype INV_e25d0_run() {
  d_step {
    INV_e25d0_done = false;
    INV_e25d0_canceled = false;
    INV_e25d0_spontaneous = true;
    INV_e25d0_procid = _pid;
```

```
1338   }
1339
1340  /* transition to initial state */
1341
1342  INV_e25d0_t0: /* ########################
1343       from state:
1344   ----- on event: SPONTANEOUS --
1345         to state: waitForEvent with no history
1346  ############################### */
1347
1348    skip;
1349    d_step {
1350  /* to state waitForEvent */
1351      INV_e25d0_s = s0;
1352    }
1353    goto INV_e25d0_microStep;
1354
1355
1356  INV_e25d0_t1: /* ########################
1357       from state: waitForEvent
1358   ----- on event: trigger.child --
1359         to state: waitForEvent with no history
1360  ############################### */
1361
1362    skip;
1363    d_step {
1364  /* executable content for transition */
1365      {
1366        tmpE.data.Var1 = 0;
1367        tmpE.data.bar = 0;
1368        tmpE.data.foo = 0;
1369        tmpE.data.sendVar1 = 0;
1370        tmpE.delay = 0;
1371        tmpE.invokeid = 0;
1372        tmpE.name = BACK_TO_HISTORY;
1373        tmpE.invokeid = INV_E25D0;
1374        tmpE.delay = 0;
1375        MAIN_eQ!tmpE;
1376        insertWithDelay(MAIN_eQ);
1377      }
1378  /* to state waitForEvent */
1379      INV_e25d0_s = s0;
1380    }
1381    INV_e25d0_spontaneous = true;
1382    goto INV_e25d0_microStep;
1383
1384  INV_e25d0_macroStep: skip;
1385  /* Determine machines with smallest delay and set their process priority */
1386    scheduleMachines();
1387
1388    atomic {
1389  /* pop an event */
1390      if
1391      :: len(INV_e25d0_iQ) != 0 -> INV_e25d0_iQ ? INV_e25d0__event   /* from internal queue */
1392      :: else -> INV_e25d0_eQ ? INV_e25d0__event                     /* from external queue */
1393      fi;
1394
1395  /* terminate if we are stopped */
1396      if
1397      :: INV_e25d0_done -> goto INV_e25d0_terminate;
1398      :: INV_e25d0_canceled -> goto INV_e25d0_cancel;
1399      :: else -> skip;
1400      fi;
1401
1402
1403  INV_e25d0_microStep:
1404  /* event dispatching per state */
1405      if
1406
1407  /* ### current state waitForEvent ######################### */
1408      :: (INV_e25d0_s == s0) -> {
1409        if
1410        :: (!INV_e25d0_spontaneous && INV_e25d0__event.name == TRIGGER_CHILD) -> {
1411  /* transition to waitForEvent */
1412          goto INV_e25d0_t1;
1413        }
1414        :: else -> {
1415  /* no transition applicable */
1416          INV_e25d0_spontaneous = false;
1417          goto INV_e25d0_macroStep;
1418        }
1419        fi;
1420      }
1421  /* this is an error as we dispatched all valid states */
```

```
1422      :: else -> assert(false);
1423      fi;
1424
1425  INV_e25d0_terminate: skip;
1426      {
1427        tmpE.data.Var1 = 0;
1428        tmpE.data.bar = 0;
1429        tmpE.data.foo = 0;
1430        tmpE.data.sendVar1 = 0;
1431        tmpE.delay = 0;
1432        tmpE.invokeid = 0;
1433        tmpE.name = DONE_INVOKE_INV_E25D0;
1434        tmpE.invokeid = INV_E25D0;
1435        MAIN_eQ!tmpE;
1436        insertWithDelay(MAIN_eQ);
1437      }
1438  INV_e25d0_cancel: skip;
1439      removePendingEventsForInvoker(INV_E25D0)
1440    }
1441  }
```

Listing A.3: PROMELA program for the SCXML state-machine in listing A.2.

A.2 State Chart XML for Nvidia Shield Gaming Console

The following SCXML document is a cleaned state-chart used for an early version of the Nvidia Shield[1] hand-held gaming console. Permission for publication was granted by Gavin Kistner (gkistner@nvidia.com) from Nvidia in July 2014.

```
<?xml version="1.0" encoding="UTF-8" ?>
<scxml xmlns="http://www.w3.org/2005/07/scxml" xmlns:thor="http://nvidia.com/uic/shield" name="Dashboard" version="1" initial="intro">
  <state id="intro">
    <transition event="data.initialized" cond="restoreAppFlag" target="ValidRunning" thor:specialRestorationTransition="true"/>
    <transition event="data.initialized" cond="(not restoreAppFlag) and hasGamesFlag" target="Games"/>
    <transition event="data.initialized" cond="(not restoreAppFlag) and not hasGamesFlag" target="Store"/>
  </state>
  <state id="ValidRunning">
    <transition event="error.*">
      <log label="An error occurred!" expr="_event.name .. '--' .. _event.data" thor:level="e"/>
    </transition>
    <parallel id="MainContent">
      <state id="FlashBottom">
        <state id="FlashBottomOff">
          <transition event="flash.bottom" target="FlashBottomOn"/>
        </state>
        <state id="FlashBottomOn">
          <transition event="animdone.flashBottom" target="FlashBottomOff"/>
        </state>
      </state>
      <state id="FlashTop">
        <state id="FlashTopOff">
          <transition event="flash.top" target="FlashTopOn"/>
        </state>
        <state id="FlashTopOn">
          <transition event="animdone.flashTop" target="FlashTopOff"/>
        </state>
      </state>
      <state id="FlashRight">
        <state id="FlashRightOff">
          <transition event="flash.right" target="FlashRightOn"/>
        </state>
        <state id="FlashRightOn">
          <transition event="animdone.flashRight" target="FlashRightOff"/>
        </state>
      </state>
      <state id="FlashLeft">
        <state id="FlashLeftOff">
          <transition event="flash.left" target="FlashLeftOn"/>
        </state>
        <state id="FlashLeftOn">
          <transition event="animdone.flashLeft" target="FlashLeftOff"/>
        </state>
      </state>
      <state id="Navigation">
        <state id="Games" initial="GamesContent">
          <transition event="bumper.left" cond="lastFlashEvent ~= _event.name">
            <raise event="flash.left"/>
            <assign location="lastFlashEvent" expr="_event.name"/>
          </transition>
          <state id="Games0">
            <transition event="dpad.down touch.focusContent" target="Games1">
              <script>bumpDownArrow()</script>
            </transition>
            <transition event="dpad.left.down" cond="lastFlashEvent ~= _event.name">
              <raise event="flash.left"/>
              <assign location="lastFlashEvent" expr="_event.name"/>
            </transition>
            <transition event="dpad.right.down touch.focusStore0 bumper.right" target="Store0"/>
            <transition event="dpad.up" cond="lastFlashEvent ~= 'dpad.up'">
              <log label="lastFlashEvent" expr="lastFlashEvent" thor:level="w"/>
              <raise event="flash.top"/>
              <assign location="lastFlashEvent" expr="'dpad.up'"/>
            </transition>
            <transition event="back" target="quit"/>
            <transition event="touch.focusGrid0" target="Grid0"/>
          </state>
          <state id="GamesContent">
            <transition event="bumper.right" target="Store1"/>
            <state id="Games1">
              <transition event="dpad.up.down touch.prevScreen" target="Games0">
                <script>bumpUpArrow()</script>
              </transition>
              <transition event="dpad.down touch.nextScreen" cond="hasAnyGamesFlag" target="Games2">
```

[1] http://shield.nvidia.com

```
75                <script>bumpDownArrow()</script>
76              </transition>
77              <transition event="streaming.quit" target="ConfirmQuit"/>
78              <transition event="games1.pastLeft" cond="lastFlashEvent ~= _event.name">
79                <raise event="flash.left"/>
80                <assign location="lastFlashEvent" expr="_event.name"/>
81              </transition>
82              <transition event="games1.pastRight" cond="lastFlashEvent ~= _event.name">
83                <raise event="flash.right"/>
84                <assign location="lastFlashEvent" expr="_event.name"/>
85              </transition>
86              <transition event="back" target="Games0"/>
87              <transition event="streaming.resume" target="ResumeDialog"/>
88            </state>
89            <state id="Games2">
90              <transition event="back games2.pastTop touch.prevScreen" target="Games1">
91                <script>bumpUpArrow()</script>
92              </transition>
93              <transition event="games2.pastLeft" cond="lastFlashEvent ~= _event.name">
94                <raise event="flash.left"/>
95                <assign location="lastFlashEvent" expr="_event.name"/>
96              </transition>
97              <transition event="games2.pastRight" cond="lastFlashEvent ~= _event.name">
98                <raise event="flash.right"/>
99                <assign location="lastFlashEvent" expr="_event.name"/>
100             </transition>
101             <transition event="games2.pastBottom" cond="lastFlashEvent ~= _event.name">
102               <raise event="flash.bottom"/>
103               <assign location="lastFlashEvent" expr="_event.name"/>
104             </transition>
105           </state>
106           <history id="GameSection">
107             <transition target="Games1"/>
108           </history>
109         </state>
110       </state>
111       <state id="Store" initial="StoreContent">
112         <state id="Store0">
113           <transition event="dpad.down touch.focusContent" target="Store1">
114             <script>bumpDownArrow()</script>
115           </transition>
116           <transition event="dpad.left.down back touch.focusGames0 bumper.left" target="Games0"/>
117           <transition event="dpad.right.down touch.focusGrid0 bumper.right" target="Grid0"/>
118           <transition event="dpad.up" cond="lastFlashEvent ~= 'dpad.up'">
119             <raise event="flash.top"/>
120             <assign location="lastFlashEvent" expr="'dpad.up'"/>
121           </transition>
122         </state>
123         <state id="StoreContent">
124           <transition event="store.showDetails" target="StoreDetails"/>
125           <transition event="back bumper.left" target="Games1"/>
126           <transition event="bumper.right" cond="hasRecentServersFlag" target="Grid1"/>
127           <transition event="bumper.right" cond="not hasRecentServersFlag and hasAvailableServersFlag" target="Grid2"/>
128           <transition event="bumper.right" cond="not hasRecentServersFlag and not hasAvailableServersFlag" target="Looking"/>
129           <state id="Store1">
130             <transition event="dpad.up.down touch.prevScreen" target="Store0">
131               <script>bumpUpArrow()</script>
132             </transition>
133             <transition event="dpad.down touch.nextScreen" target="Store2">
134               <script>bumpDownArrow()</script>
135             </transition>
136             <transition event="store1.pastLeft" cond="lastFlashEvent ~= _event.name">
137               <raise event="flash.left"/>
138               <assign location="lastFlashEvent" expr="_event.name"/>
139             </transition>
140             <transition event="store1.pastRight" cond="lastFlashEvent ~= _event.name">
141               <raise event="flash.right"/>
142               <assign location="lastFlashEvent" expr="_event.name"/>
143             </transition>
144           </state>
145           <state id="Store2">
146             <transition event="store2.pastTop back touch.prevScreen" target="Store1">
147               <script>bumpUpArrow()</script>
148             </transition>
149             <transition event="store2.pastLeft" cond="lastFlashEvent ~= _event.name">
150               <raise event="flash.left"/>
151               <assign location="lastFlashEvent" expr="_event.name"/>
152             </transition>
153             <transition event="store2.pastRight" cond="lastFlashEvent ~= _event.name">
154               <raise event="flash.right"/>
155               <assign location="lastFlashEvent" expr="_event.name"/>
156             </transition>
157             <transition event="store2.pastBottom" cond="lastFlashEvent ~= _event.name">
158               <raise event="flash.bottom"/>
```

```
159               <assign location="lastFlashEvent" expr="_event.name"/>
160             </transition>
161           </state>
162           <history id="StoreSection1">
163             <transition target="Store1"/>
164           </history>
165           <history id="StoreSection2">
166             <transition target="Store2"/>
167           </history>
168         </state>
169       </state>
170       <state id="Grid" initial="GridContent">
171         <transition event="bumper.right" cond="lastFlashEvent ~= _event.name">
172           <raise event="flash.right"/>
173           <assign location="lastFlashEvent" expr="_event.name"/>
174         </transition>
175         <state id="Grid0">
176           <transition event="dpad.down touch.focusContent" cond="hasRecentServersFlag" target="Grid1">
177             <script>bumpDownArrow()</script>
178           </transition>
179           <transition event="dpad.right.down" cond="lastFlashEvent ~= _event.name">
180             <raise event="flash.right"/>
181             <assign location="lastFlashEvent" expr="_event.name"/>
182           </transition>
183           <transition event="dpad.left.down back touch.focusStore0 bumper.left" target="Store0"/>
184           <transition event="dpad.up" cond="lastFlashEvent ~= 'dpad.up'">
185             <raise event="flash.top"/>
186             <assign location="lastFlashEvent" expr="'dpad.up'"/>
187           </transition>
188           <transition event="dpad.down touch.focusContent" cond="not hasAvailableServersFlag" target="Looking"/>
189           <transition event="dpad.down touch.focusContent" cond="hasAvailableServersFlag and not hasRecentServersFlag" target="Grid2">
190             <script>bumpDownArrow()</script>
191           </transition>
192           <transition event="touch.focusGames0" target="Games0"/>
193         </state>
194         <state id="GridContent">
195           <transition event="bumper.left" target="Store1"/>
196           <state id="Grid1">
197             <transition event="dpad.up.down touch.prevScreen" target="Grid0">
198               <script>bumpUpArrow()</script>
199             </transition>
200             <transition event="dpad.down touch.nextScreen" target="Grid2">
201               <script>bumpDownArrow()</script>
202             </transition>
203             <transition event="grid1.pastLeft" cond="lastFlashEvent ~= _event.name">
204               <raise event="flash.left"/>
205               <assign location="lastFlashEvent" expr="_event.name"/>
206             </transition>
207             <transition event="grid1.pastRight" cond="lastFlashEvent ~= _event.name">
208               <raise event="flash.right"/>
209               <assign location="lastFlashEvent" expr="_event.name"/>
210             </transition>
211             <transition event="server.showgames" target="ConnectingDialog"/>
212             <transition event="server.forget" target="ConfirmForget"/>
213             <transition event="streaming.quit" target="ConfirmQuit"/>
214             <transition event="streaming.resume" target="ResumeDialog"/>
215             <transition event="server.updated" cond="not hasRecentServersFlag and hasAvailableServersFlag" target="Grid2"/>
216             <transition event="server.updated" cond="not hasRecentServersFlag and not hasAvailableServersFlag" target="Looking"/>
217           </state>
218           <state id="Grid2">
219             <transition event="grid2.pastLeft" cond="lastFlashEvent ~= _event.name">
220               <raise event="flash.left"/>
221               <assign location="lastFlashEvent" expr="_event.name"/>
222             </transition>
223             <transition event="grid2.pastRight" cond="lastFlashEvent ~= _event.name">
224               <raise event="flash.right"/>
225               <assign location="lastFlashEvent" expr="_event.name"/>
226             </transition>
227             <transition event="grid2.pastBottom" cond="lastFlashEvent ~= _event.name">
228               <raise event="flash.bottom"/>
229               <assign location="lastFlashEvent" expr="_event.name"/>
230             </transition>
231             <transition event="grid2.pastTop back touch.prevScreen" cond="hasRecentServersFlag" target="Grid1">
232               <script>bumpUpArrow()</script>
233             </transition>
234             <transition event="grid2.pastTop back touch.prevScreen" cond="not hasRecentServersFlag" target="Grid0"/>
235             <transition event="server.showgames" target="ConnectingDialog"/>
236           </state>
237           <history id="GridSection">
238             <transition target="Grid1"/>
239           </history>
240           <state id="BrowseGames">
241             <transition event="streaming.start" target="LaunchingDialog"/>
242             <transition event="touch.grid1 back" target="Grid1"/>
```

```
            <transition event="dpad.up.down touch.prevScreen" target="Grid0">
              <script>bumpUpArrow()</script>
            </transition>
            <transition event="gamesList.pastLeft" cond="lastFlashEvent ~= _event.name">
              <raise event="flash.left"/>
              <assign location="lastFlashEvent" expr="_event.name"/>
            </transition>
            <transition event="gamesList.pastRight" cond="lastFlashEvent ~= _event.name">
              <raise event="flash.right"/>
              <assign location="lastFlashEvent" expr="_event.name"/>
            </transition>
            <transition event="gamesList.pastBottom" cond="lastFlashEvent ~= _event.name">
              <raise event="flash.bottom"/>
              <assign location="lastFlashEvent" expr="_event.name"/>
            </transition>
          </state>
        </state>
        <state id="Troubleshooting">
          <transition event="bumper.left back" target="Store1"/>
          <transition event="dpad.up touch.prevScreen" target="Grid0">
            <script>bumpUpArrow()</script>
          </transition>
          <transition event="server.updated" cond="not hasRecentServersFlag and hasAvailableServersFlag" target="Grid2"/>
          <transition event="server.updated" cond="hasRecentServersFlag" target="Grid1"/>
          <state id="Looking">
            <transition event="animdone.looking" cond="not wiFiOnFlag" target="WiFiOff"/>
            <transition event="animdone.looking" cond="wiFiOnFlag and not wiFiConnectedFlag" target="NetworkError"/>
            <transition event="animdone.looking" cond="wiFiOnFlag and wiFiConnectedFlag" target="OtherError"/>
          </state>
          <state id="OtherError"/>
          <state id="NetworkError">
            <transition event="button.Y">
              <script>launchWiFiSettings()</script>
            </transition>
          </state>
          <state id="WiFiOff">
            <transition event="button.Y">
              <script>launchWiFiSettings()</script>
            </transition>
          </state>
        </state>
      </state>
      <history id="ContentHistory" type="deep"/>
    </state>
    <state id="Miracast">
      <state id="MiracastOff">
        <transition event="miracast.on" target="MiracastOn"/>
      </state>
      <state id="MiracastOn">
        <transition event="miracast.off" target="MiracastOff"/>
      </state>
    </state>
  </parallel>
  <state id="StoreDetails">
    <transition event="touch.store back" cond="not startupGameNotFeatured" target="StoreSection1"/>
    <transition event="touch.store back" cond="startupGameNotFeatured" target="StoreSection2"/>
    <state id="ScrollingList">
      <transition event="game.buy touch.buy">
        <script>launchGooglePlay()</script>
      </transition>
      <transition event="screenshot.show" target="VisualsBig"/>
      <transition event="game.details" target="DetailsBig"/>
      <transition event="dpad.left">
        <script>focusPrevStoreDetail()</script>
      </transition>
      <transition event="dpad.right">
        <script>focusNextStoreDetail()</script>
      </transition>
      <transition event="button.A">
        <script>selectFocusedStoreDetail()</script>
      </transition>
    </state>
    <state id="VisualsBig">
      <transition event="back" target="ScrollingList"/>
      <transition event="dpad.left">
        <script>showPrevStoreVisual()</script>
      </transition>
      <transition event="dpad.right">
        <script>showNextStoreVisual()</script>
      </transition>
    </state>
    <state id="DetailsBig">
      <transition event="back game.closedetails" target="ScrollingList"/>
      <transition event="dpad.down">
```

```
          <script>scrollStoreDetailsDown()</script>
        </transition>
        <transition event="dpad.up">
          <script>scrollStoreDetailsUp()</script>
        </transition>
      </state>
    </state>
    <state id="ConfirmForget">
      <transition event="dialog.cancel back" target="Grid1"/>
      <transition event="dialog.ok" target="Grid1">
        <script>forgetSelectedServer()</script>
      </transition>
    </state>
    <state id="ConfirmQuit">
      <transition event="dialog.ok" target="Grid1">
        <script>quitGameForSelectedServer()</script>
      </transition>
      <transition event="dialog.cancel back" target="Grid1"/>
    </state>
    <state id="Connecting">
      <transition event="back" target="GridSection"/>
      <state id="ConnectingDialog">
        <onentry>
          <send event="timeout.gamesList" id="connectingTimeout" delay="180s"/>
        </onentry>
        <onexit>
          <cancel sendid="connectingTimeout"/>
        </onexit>
        <transition event="connect.error timeout.gamesList" target="ConnectError"/>
        <transition event="connect.established" target="Connected"/>
      </state>
      <state id="ConnectError">
        <transition event="dialog.cancel back" target="GridSection"/>
      </state>
      <state id="Connected">
        <transition event="animdone.notification" target="BrowseGames"/>
      </state>
    </state>
    <state id="Launching">
      <state id="LaunchingDialog">
        <onentry>
          <send event="timeout.launchGame" id="launchingTimeout" delay="300s"/>
        </onentry>
        <onexit>
          <cancel sendid="launchingTimeout"/>
        </onexit>
        <transition event="streaming.error timeout.launchGame" target="LaunchingError"/>
        <transition event="back" target="GridSection">
          <script>cancelActiveGameLaunch()</script>
        </transition>
      </state>
      <state id="LaunchingError">
        <transition event="dialog.cancel back" target="GridSection"/>
      </state>
    </state>
    <state id="Resuming">
      <state id="ResumeDialog">
        <transition event="back" target="ContentHistory">
          <script>cancelResumeStreaming()</script>
        </transition>
        <transition event="animdone.notification" target="InitiateResume"/>
      </state>
      <state id="InitiateResume">
        <onentry>
          <send event="timeout.resumeGame" id="resumingTimeout" delay="60s"/>
        </onentry>
        <onexit>
          <cancel sendid="resumingTimeout"/>
        </onexit>
        <transition event="streaming.error timeout.resumeGame" target="ResumeError"/>
        <transition event="back" target="ContentHistory">
          <script>cancelResumeStreaming()</script>
        </transition>
      </state>
      <state id="ResumeError">
        <transition event="dialog.cancel back" target="ContentHistory"/>
        <transition event="dialog.retry" target="InitiateResume"/>
      </state>
    </state>
  </state>
  <final id="quit"/>
</scxml>
```

Listing A.4: Cleaned SCXML for an early version of the Nvidia Shield handheld console.

B Abbreviations

BDI Beliefs, Desires and Intentions 29, 63, *Glossary:* Beliefs, Desires and Intentions

CCXML Call Control eXtensible Markup Language 84, *Glossary:* Call Control eXtensible Markup Language

CSS Cascading Style Sheets 39

CTL Computation Tree Logic 129, 132–134, 202

DFA Deterministic Finite Automaton 30, 45–48, 58, 103, 104, 106, 108, 122, 123, 126, 129, 132, 135, 137, 155, 159, 200–203

DME Dialog Move Engine 61, 62

DOM Document Object Model 77, 97, 101, 187–190, 197

DQA Deterministic Queue Automaton 11, 57, 103, 124–126, 200

DTMF Dual-Tone Multi-Frequency signaling *Glossary:* Dual-Tone Multi-Frequency signaling

EMMA Extensible MultiModal Annotation Markup Language 25, *Glossary:* Extensible MultiModal Annotation Markup Language

HCI Human-Computer Interaction 9, 13–17, 22, 25, 26, 28, 29, 46, 199, *Glossary:* Human-Computer Interaction

HMM Hidden Markov Model 67

HTML Hypertext Markup Language 31, 39, 57, 59, 77, 78, 97, 181, 182, 184, 185, 187–192, 194–196, 202, 242, 243, *Glossary:* Hypertext Markup Language

IDL Interface Description Language 81

IM Interaction Manager 33, 39–44, 92–95, 179, 182, 192, 243, *Glossary:* Interaction Manager

IRP Implementation Report Plan 83, 104, 105, 118, 143, 144, 162, 165, 178–180, 200, *Glossary:* Implementation Report Plan

ISU Information State Update 12, 60, 63, 90

IVR Interactive Voice Response 185, *Glossary:* Interactive Voice Response

LTL Linear Temporal Logic 9, 11, 129, 132–134, 161–164, 170, 177, 178, 199, 200

MC Modality Component 33, 40–44, 92–95, 170, 178–180, 182, 184–187, 191, 192, 243, *Glossary:* Modality Component

MDP Markov Decision Process 64–67

MDS Multimodal Dialog System 30–32, 34–36, 38–41, 43, 44, 56, 63, 92, 94, 95, 103, 192, 199, 243, *Glossary:* Multimodal Dialog System

MVC Model-View-Controller 17, 33, *Glossary:* Model-View-Controller

NFA Non-Deterministic Finite Automaton 106, 108, 200

PAC Presentation-Abstraction-Control 33, *Glossary:* Presentation-Abstraction-Control

PDA Push-Down Automaton 103, 122, 123, 126, 155, 200, 201

POMDP Partially Observable Markov Decision Process 65, 66

PROMELA PRocess MEta LAnguage 9–12, 104, 105, 129–131, 135–157, 159–168, 170, 171, 174, 175, 177, 178, 194, 199–202, 205, 217, 234, 243, *Glossary:* PRocess MEta LAnguage

SCXML State Chart eXtensible Markup Language 9–12, 31, 44–46, 50, 54, 56, 57, 67, 69–83, 85, 87–99, 101, 103–115, 118–131, 136, 137, 140, 143, 144, 146, 148, 154, 155, 159–165, 170, 177–182, 184–196, 198–203, 205, 207, 209, 210, 216, 217, 234, 242, *Glossary:* State Chart eXtensible Markup Language

SDS Spoken Dialog System 36, 37, 46, 59, 60, 65, 131, 185, *Glossary:* Spoken Dialog System

SMV Symbolic Model Verifier 11, 129–131, *Glossary:* Symbolic Model Verifier

UIMS User-Interface Management System 27, 28, 31–33, 199, *Glossary:* User-Interface Management System

UML Unified Modeling Language 69, 72, 130

VoiceXML Voice eXtensible Markup Language 39, 59, 60, 95, 96, 182, 184–187, 191, 242, *Glossary:* Voice eXtensible Markup Language

W3C World Wide Web Consortium 10, 31, 38–41, 43, 44, 56, 67, 69, 74, 81, 92–95, 97, 107, 118–120, 127, 128, 131, 144, 146, 170, 171, 179, 185, 190, 199, 200, 203, 242, 243, *Glossary:* World Wide Web Consortium

W3C MMI architecture W3C Multimodal Architecture and Interfaces 33, 41, 43, 44, 92, 178, 180, 182, 184–187, 191–193, 199, *Glossary:* W3C Multimodal Architecture and Interfaces

W3C MMI framework W3C Multimodal Interaction Framework 31, 33, 34, 40–44, 92, 199, *Glossary:* W3C Multimodal Interaction Framework

W3C MMI WG W3C Multimodal Interaction Working Group 10–12, 25, 31, 38, 39, 56, 69, 192, 199, 202, *Glossary:* W3C Multimodal Interaction Working Group

Web IDL Web Interface Description Language 181, *Glossary:* Web Interface Description Language

WER Word-Error-Rate 20, *Glossary:* Word-Error-Rate

WIMP Window Icon Menu Pointer 14, 15

WWHT What-Which-How-Then *Glossary:* What-Which-How-Then

XHTML Extensible HyperText Markup Language 39, 79, 182, 185, 192, *Glossary:* Extensible HyperText Markup Language

C Terms

Beliefs, Desires and Intentions is a conceptualization for the mental state of an agent or software actor. In the context of dialog modeling Traum & Allen also proposed to include *obligations* as reactions expected by other agents (see [TA94]). 29, 63

Call Control eXtensible Markup Language is designed to provide telephony call control support for dialog systems. While CCXML can be used with any dialog systems capable of handling media, CCXML has been designed to complement and integrate with a Voice eXtensible Markup Language (VoiceXML) interpreter [BS11]. 84

Data-Model The data-model is a component of a compliant SCXML interpreter and serves as an embedded scripting language available in the document. In this function, it is not unlike the ECMAScript runtime available for HTML documents. 11, 12, 45, 46, 56, 57, 67, 68, 70–80, 82, 84, 89–92, 94, 96, 97, 99–101, 103–108, 110, 112, 114, 118, 120, 122, 124, 127, 129, 137–146, 148–153, 161, 163–168, 170, 171, 178–181, 185, 187, 188, 190, 192–194, 196–201, 203

ECMAScript (also JavaScript, JScript or ActionScript) is a popular scripting language in the context of markup documents and *client-side* scripting in the World Wide Web. It is standardized as ECMA-262 and ISO/IEC 16262. 56, 57, 68, 70, 73, 75, 77, 90, 94, 100, 104, 105, 119, 120, 127, 143–145, 165, 178–181, 186–189, 193, 194

Extensible HyperText Markup Language is a reformulation of HTML 4 in XML 1.0. This is required if HTML is to parsed by a generic XML parser. Eventually, HTML 5 will also be available as a proper XML dialect, but for now there is no such standard defined. Pragmatically, however, XHTML for HTML 5 would merely include all additional elements of HTML 5. 39, 79, 182

Extensible MultiModal Annotation Markup Language is intended for use by systems that provide semantic interpretations for a variety of inputs, including but not necessarily limited to, speech, natural language text, GUI and ink input. 25

Human-Computer Interaction is the study of the *interaction between people and computers [concerning the] physical, psychological and theoretical aspects of this process* [DFAB03] 9, 13, 46, 199

Hypertext Markup Language is the core language of the World Wide Web. It was primarily designed as a language for semantically describing scientific documents, although its general design and adaptations over the years have enabled it to be used to describe a number of other types of documents [BFL+14]. 31, 77, 181, 202

Implementation Report Plan is document accompanying a W3C standard and is required for a standard to move beyond the Candidate Recommendation phase. In the scope of this thesis, this virtually always refers to the SCXML Implementation Report Plan and most notably the 232 tests for functional requirements of a compliant SCXML interpreter. 83, 104, 143, 178, 200

Interaction Manager is a component in the W3C Multimodal Architecture and Interfaces recommendation for the coordination of the different modalities [BBD+12]. 33, 39, 92, 179, 243

Interactive Voice Response is a technology for humans to interact with computer systems via a telephone, employing DTMF or voice as modalities. 185

Modality Component is a component in the W3C Multimodal Architecture and Interfaces recommendation to provide modality-specific interaction capabilities [BBD+12]. 33, 40, 92, 170, 243

Model-View-Controller is a pattern to organize responsibilities and relations when describing a (i.e. graphical) user interface. The *model* holds all of the domain dependent application state. The *view* presents a suitable subset of this state to a user and is registered with the model to get notified and updated when the model's relevant values are changed. The *controller* listens for user input issued on the views display and will update the model accordingly, leading to the *view* to be updated. 17, 33

Multimodal Dialog System is a kind of human computer interface, where user input and system output are conveyed via multiple modalities 30, 31, 92, 103, 192, 199

Presentation-Abstraction-Control PAC is an implementation model for building user interfaces. It recursively structures an interactive application in three parts: the presentation to define the interaction syntax, the abstraction to represent the application's semantics and the control to maintain a mapping between the presentation and the abstraction [Cou87]. 33

PRocess MEta LAnguage is a modeling language for concurrent processes; it is the input language of the Spin model-checker to verify LTL expressions. 9, 104, 129, 135, 170, 199

SPIN (originally Simple PROMELA Interpreter) is a popular, automaton model-checker for system models given in PROMELA and properties as linear temporal logic expressions[Hol97]. `http://spinroot.com` 9–11, 69, 104, 129–131, 133–139, 141, 142, 144, 149–151, 153, 156, 157, 159–161, 163–168, 174, 177, 199–202, 217

Spoken Dialog System is a kind of human computer interface, where user input and system output is predominantly via voice. 36, 131, 185

State Chart eXtensible Markup Language is a general-purpose event-based state machine language that combines concepts from CCXML and Harel State Tables [BAA+15]. 9, 31, 69, 103, 129, 170, 199

Symbolic Model Verifier is a tool for checking finite state systems against specifications in the temporal logic CTL. A more modern reimplementation and extension is found in the NuSMV model-checker. 11, 129

User-Interface Management System are interactive systems for the development and execution of an interactive software system's human-computer interface. They help you specify, design, prototype, implement, execute, evaluate, modify, and maintain such interfaces [Hix90]. 27, 31, 199

Voice eXtensible Markup Language VoiceXML is designed for creating audio dialogs that feature synthesized speech, digitized audio, recognition of spoken and DTMF key input, recording of spoken input, telephony, and mixed initiative conversations. The most widely used variant is VoiceXML 2.1 [BAO+07] as an addendum of VoiceXML 2.0 [TCB+04] recommendation. Standardization efforts for the succeeding VoiceXML 3.0 [MBO+10] standard are currently dormant. 39, 95, 182, 242

W3C Multimodal Architecture and Interfaces The W3C Multimodal Architecture and Interfaces specification is a W3C recommendation detailing the collaboration of Interaction Managers (IM) and Modality Components (MC) to realize a MDS. It, foremost, defines a set of life-cycle events for a dialog manager as an IM to instantiate and control modality-specific components as MCs (`http://www.w3.org/TR/mmi-arch/`). 33, 92, 178, 199

W3C Multimodal Interaction Framework The W3C Multimodal Interaction Framework specification is a W3C note outlining the general structure and responsibilities of components in a MDS (`http://www.w3.org/TR/mmi-framework/`). 31, 92, 199

W3C Multimodal Interaction Working Group The W3C Multimodal Interaction Working Group is part of the W3C Multimodal Interaction Activity. Its mission is to develop open standards that extend the Web to allow multiple modes of interaction GUI, Speech, Vision, Pen, Gestures, Haptic interfaces, etc. and enable interfaces *Anyone, Anywhere, Any device, Any time* accessible through the user's preferred modes of interaction with services that adapt to the device, user and environmental conditions (`http://www.w3.org/2002/mmi/`). 10, 25, 31, 69, 192, 199

Web Interface Description Language is a formal description for an programming interface popular with many W3C standards. `http://www.w3.org/TR/WebIDL/` 181

Word-Error-Rate is a measure for the (in-)accuracy of an automatic speech recognition system. It is usually calculated as the normalized Levenshtein- or Edit-distance between the text recognized by an automated speech recognition system and the actual utterance. 20

World Wide Web Consortium The W3C is the principal standardization body for the World Wide Web. It provides a forum to propose and discuss standards that may eventually become recommendations to be employed in the context of the World Wide Web. It is best known for maintaining and developing the HTML standard. 10, 31, 69, 107, 131, 170, 199

[ABB+96] R. J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *ACM SIGSOFT Software Engineering Notes*, 21(6):156–166, October 1996.

[AP80] J. F. Allen and C. Perrault. Analyzing intention in utterances. *Artificial Intelligence*, 15(3):143 – 178, 1980.

[APM07] J. G. Amores, G. Pérez, and P. Manchón. MIMUS: a multimodal and multilingual dialogue system for the home domain. In *ACL '07: Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, pages 1–4. Association for Computational Linguistics, June 2007.

[ASK+94] B. Anderson, M. Smyth, R. P. Knott, M. Bergan, J. Bergan, and J. L. Alty. Minimising conceptual baggage: Making choices about metaphor. In *Proceedings of HCI '94 People and Computers IX*, pages 179–194, January 1994.

[Aus62] J. L. Austin. *How to do Things with Words*. Oxford University Press, New York, 1962.

[AWM95] G. D. Abowd, H.-M. Wang, and A. F. Monk. A formal technique for automated dialogue development. In *Proceedings of the 1st Conference on Designing Interactive Systems: Processes, Practices, Methods, & Techniques*, DIS '95, pages 219–226, New York, NY, USA, 1995. ACM.

[AYK+13] T. Ando, H. Yatsu, W. Kong, K. Hisazumi, and A. Fukuda. Formalization and model checking of SysML state machine diagrams by csp#. In B. Murgante, S. Misra, M. Carlini, C. M. Torre, H.-Q. Nguyen, D. Taniar, B. O. Apduhan, and O. Gervasi, editors, *Computational Science and Its Applications – ICCSA 2013*, volume 7973 of *Lecture Notes in Computer Science*, pages 114–127. Springer Berlin Heidelberg, 2013.

[BAA+15] J. Barnett, R. Akolkar, R. J. Auburn, M. Bodell, D. C. Burnett, J. Carter, S. Mcglashan, T. o. r. Lager, M. Helbing, R. Hosn, T. V. Raman, K. Reifenrath, and N. Rosenthal. State Chart XML (SCXML): State Machine Notation for Control Abstraction. W3C recommendation, W3C, September 2015. http://www.w3.org/TR/2015/REC-scxml-20150901/.

[Bad00] A. Baddeley. The episodic buffer: a new component of working memory? *Trends in Cognitive Sciences*, 4(11):417 – 423, 2000.

[BAO+07] P. Baggia, R. Auburn, M. Oshry, K. Rehor, D. Burke, B. Porter, M. Bodell, J. Carter, S. McGlashan, D. Burnett, and A. Lee. Voice extensible markup language (VoiceXML) 2.1. W3C recommendation, W3C, June 2007. http://www.w3.org/TR/2007/REC-voicexml21-20070619/.

[BB87] R. Baecker and W. Buxton. *Readings in human-computer interaction: A multidisciplinary approach*, pages 357–365. Los Altos, CA: Morgan Kaufmann, 1987.

[BBD+12] J. Barnett, M. Bodell, D. Dahl, I. Kliche, J. Larson, B. Porter, D. Raggett, T. Raman, B. H. Rodriguez, M. Selvaraj, R. Tumuluri, A. Wahbe, P. Wiechno, and M. Yudkowsky. Multimodal architecture and interfaces. W3C recommendation, W3C, October 2012. http://www.w3.org/TR/2012/REC-mmi-arch-20121025/.

[BBS13] M. Bolton, E. Bass, and R. Siminiceanu. Using formal verification to evaluate human-automation interaction: A review. *Systems, Man, and Cybernetics: Systems, IEEE Transactions on*, 43(3):488–503, May 2013.

[BD90] M. M. Blattner and R. B. Dannenberg. Chi'90 workshop on multimedia and multimodal interface design. *ACM SIGCHI Bulletin*, 22(2):54–58, November 1990.

[BD07] B. Balentine and L. Degler. *It's Better to Be a Good Machine Than a Bad Person: Speech Recognition and Other Exotic User Interfaces in the Twilight of the Jetsonian Age*. Prentice-Hall signal processing series. ICMI Press, 2007.

[Bel57] R. Bellman. A markovian decision process. Technical report, U.S. Defense Technical Information Center, 1957.

[Ber96] N. O. Bernsen. A reference model for output information in intelligent multimedia presentation systems. In *Proceedings of the ECAI'96 Workshop on: Towards a Standard Reference Model for Intelligent Multimedia Presentation Systems*, 1996.

[BFF+97] M. Bordegoni, G. Faconti, S. Feiner, M. T. Maybury, T. Rist, S. Ruggieri, P. Trahanias, and M. Wilson. A standard reference model for intelligent multimedia presentation systems. *Computer standards & interfaces*, 18(6):477–496, 1997.

[BFL+92] L. Bass, R. Faneuf, R. Little, N. Mayer, B. Pellegrino, S. Reed, R. Seacord, S. Shepard, and M. Szczur. A metamodel for the runtime architecture of an interactive system: the uims tool developers workshop. *SIGCHI Bulletin*, 24(1), January 1992.

[BFL+14] R. Berjon, S. Faulkner, T. Leithead, S. Pfeiffer, E. O'Connor, and E. D. Navara. HTML5. Candidate recommendation, W3C, July 2014. http://www.w3.org/TR/2014/CR-html5-20140731/.

[BG96] M. M. Blattner and E. P. Glinert. Multimodal integration. *Multimedia, IEEE*, 3(4):14–24, 1996.

[BH74] A. D. Baddeley and G. J. Hitch. Working memory. *The psychology of learning and motivation*, 8:47–89, 1974.

[BH97] R. Bharadwaj and C. Heitmeyer. Verifying SCR requirements specifications using state exploration. In *In Proceedings of 1st ACM SIGPLAN Workshop on Automatic Analysis of Software*, pages 9–24, 1997.

[BHP+09] K. E. Boyer, E. Y. Ha, R. Phillips, M. D. Wallis, M. A. Vouk, and J. C. Lester. Inferring tutorial dialogue structure with hidden markov modeling. In *Proceedings of the 4th Workshop on Innovative Use of NLP for Building Educational Applications*, pages 19–26. Association for Computational Linguistics, 2009.

[BIP88] M. E. Bratman, D. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational intelligence*, 4(4):349–355, 1988.

[BMP13] G. Brat, C. Martinie, and P. Palanque. V&V of lexical, syntactic and semantic properties for interactive systems through model checking of formal description of dialog. In *Human-Computer Interaction. Human-Centred Design Approaches, Methods, Tools, and Environments*, pages 290–299. Springer Berlin Heidelberg, 2013.

[Bol80] R. A. Bolt. "Put-that-there": Voice and gesture at the graphics interface. In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*, pages 262–270, New York, NY, USA, 1980. ACM.

[Bol82] R. A. Bolt. Eyes at the interface. In *Proceedings of the CHI'82 conference*, pages 360–362, New York, New York, USA, 1982. ACM Press.

[BR04] P. Bhaduri and S. Ramesh. Model checking of statechart models: Survey and research directions. *Computing Research Repository*, cs.SE/0407038, 2004.

[Bra97] S. Bradner. Key words for use in RFCs to indicate requirement levels, March 1997.

[BS11] P. Baggia and M. Scott. Voice browser call control: CCXML version 1.0. W3C recommendation, W3C, July 2011. http://www.w3.org/TR/2011/REC-ccxml-20110705/.

[Buc58] W. Buchholz. The selection of an instruction language. In *Proceedings of the May 6-8, 1958, Western Joint Computer Conference: Contrasts in Computers*, IRE-ACM-AIEE '58 (Western), pages 128–130, New York, NY, USA, 1958. ACM.

[Büc62] J. R. Büchi. On a decision method in a restricted second order arithmetic. In Press, editor, *Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science*, pages 1–11, 1962.

[Bui06] T. H. Bui. Multimodal dialogue management - state of the art. Technical Report TR-CTIT-06-01, Centre for Telematics and Information Technology University of Twente, Enschede, Netherlands, January 2006.

[Bun95] H. Bunt. Dialogue control functions and interaction design. In *Dialogue in Instruction*, pages 197–214. Springer Verlag, 1995.

[Bun96] H. C. Bunt. Dynamic Interpretation and Dialogue Theory. In M. M. Taylor, F. Néel, and D. G. Bouwhuis, editors, *The Structure of Multimodal Dialogue, Volume 2*. John Benjamins, Amsterdam, 1996.

[CD89] E. M. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 428–437. Springer, 1989.

[CD05] M. L. Crane and J. Dingel. On the semantics of uml state machines: Categorization and comparision. In *Technical Report 2005-501, School of Computing, Queen's*, 2005.

[CE82] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[CH00] E. M. Clarke and W. Heinle. Modular translation of statecharts to SMV. Technical Report CMU-CS-00-XXX, Carnegie Mellon University School of Computer Science, 2000.

[CJM+97a] P. R. Cohen, M. Johnston, D. McGee, S. Oviatt, J. Pittman, I. Smith, L. Chen, and J. Clow. QuickSet: Multimodal interaction for distributed applications. In *Proceedings of the 5th ACM international conference on Multimedia*, pages 31–40. ACM, 1997.

[CJM+97b] P. R. Cohen, M. Johnston, D. McGee, S. Oviatt, J. Pittman, I. Smith, L. Chen, and J. Clow. QuickSet: multimodal interaction for simulation set-up and control. In *ANLC '97: Proceedings of the 5th conference on Applied natural language processing*. Association for Computational Linguistics, March 1997.

[CMP90] P. R. Cohen, J. L. Morgan, and M. E. Pollack. Rational interaction as the basis for communication. In *Intentions in Communication*, pages 221–235. MIT Press, Cambridge, Massachusetts, 1990.

[CMR90] S. K. Card, J. D. Mackinlay, and G. G. Robertson. The design space of input devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 117–124, New York, NY, USA, 1990. ACM.

[CNS93] J. Coutaz, L. Nigay, and D. Salber. Taxonomic issues for multimodal and multimedia interactive systems. In *ERCIM Workshop on Multimodal Human-Computer Interaction, Working Material, Nancy, France*, November 1993.

[Coh97] P. Cohen. Dialogue modeling. In R. Cole, editor, *Survey of the State of the Art in Human Language Technology*, pages 204–210. Cambridge University Press, New York, NY, USA, 1997.

[Cou87] J. Coutaz. PAC: An object-oriented model for dialog design. In *Proceedings of INTERACT '87: The IFIP Conference on Human Computer Interaction*, pages 431–436, 1987.

[Cou91] J. Coutaz. A taxonomy for multimedia and multimodal user interfaces. *Proceedings of the 1st ERCIM Workshop on Multimodal HCI*, 1991.

[CSW+03] R. Catizone, A. Setzer, Y. Wilks, et al. Multimodal dialogue management in the COMIC project. In *Proceeding of the Workshop on Dialogue Systems: Interaction, Adaptation and Styles of Management. 10th Conference of the EACL*, 2003.

[Dah13] D. A. Dahl. The W3C multimodal architecture and interfaces standard. *Journal on Multimodal User Interfaces*, 7(3):171–182, 2013.

[DBB52] K. H. Davis, R. Biddulph, and S. Balashek. Automatic recognition of spoken digits. *The Journal of the Acoustical Society of America*, 24(6):637–642, 1952.

[DFAB03] A. Dix, J. E. Finlay, G. D. Abowd, and R. Beale. *Human-Computer Interaction (3rd Edition)*. Prentice-Hall, Upper Saddle River, NJ, USA, 2003.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[Don78] W. C. Donelson. Spatial management of information. *ACM SIGGRAPH Computer Graphics*, 12(3):203–209, August 1978.

[DQT+09] M. Dinarelli, S. Quarteroni, S. Tonelli, A. Moschitti, and G. Riccardi. Annotating spoken dialogs: From speech segments to dialog acts and frame semantics. In *Proceedings of the 2Nd Workshop on Semantic Representation of Spoken Language*, SRSL '09, pages 34–41, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[Dud55] H. Dudley. Fundamentals of speech synthesis. *Journal of the Audio Engineering Society*, 3(4):170–185, 1955.

[Esh09] R. Eshuis. Reconciling statechart semantics. *Science of Computer Programming*, 74(3):65 – 99, 2009.

[FDK15] P. Forbrig, A. Dittmar, and M. Kühn. Extending scxml by a feature for creating dynamic state instances. In *Proceedings of the 2nd EICS Workshop on Engineering Interactive Computer Systems with SCXML*, 2015.

[FFC+10] M. Frappier, B. Fraikin, R. Chossart, R. Chane-Yack-Fa, and M. Ouenzar. Comparison of model checking tools for information systems. *ICFEM*, pages 581–596, 2010.

[FGRM09] J. G. Fiscus, J. S. Garofolo, R. T. Rose, and M. Michel. AVSS multiple camera person tracking challenge evaluation overview. In *6th IEEE International Conference on Advanced Video and Signal Based Surveillance, AVSS 2009, 2-4 September 2009, Genova, Italy*, page 219, 2009.

[Flo67] R. W. Floyd. Assigning meaning to programs. In *Proceedings of the Symposium on Applied Maths*, pages 19–32. AMS, 1967.

[FS12] E. A. Feinberg and A. Shwartz. *Handbook of Markov decision processes: methods and applications*, volume 40. Springer Science & Business Media, 2012.

[GPVW96] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, UK, 1996. Chapman & Hall, Ltd.

[Gre85] M. Green. The university of alberta user interface management system. In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '85, pages 205–213, New York, NY, USA, 1985. ACM.

[Gre86] M. Green. A survey of three dialogue models. *Transactions on Graphics*, 5(3), July 1986.

[Hag89] J. A. Hager. Software cost reduction methods in practice. *IEEE Transactions on Software Engineering*, 15(12):1638–1644, December 1989.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[Hix90] D. Hix. Generations of user-interface management systems. *IEEE Software*, 7(5):77–87, 1990.

[HK97] C. Heitmeyer and J. Kirby. Tools for formal specification, verification, and validation of requirements. *Proceedings of the 12th Annual IEEE Conference on Computer Assurance*, pages 35 – 47, 1997.

[HN96] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[Hol97] G. J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[Hol00] G. Holzmann. Logic verification of ANSI-C code with SPIN. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 131–147. Springer Berlin Heidelberg, 2000.

[HPSS87] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts (extended abstract). In *Proceedings of the Symposium on Logic in Computer Science*, pages 54–64, Ithaca, New York, USA, June 1987.

[HT10] C. Haubelt and J. Teich. *Digitale Hardware/Software-Systeme: Spezifikation und Verifikation*. eXamen.press. Springer, 2010.

[JAP+11] H. Jayawardana, H. Amarasekara, P. Peelikumbura, W. Jayathilaka, S. Abeyaratne, and S. Dewasurendra. Design and implementation of a statechart based reconfigurable elevator controller. In *6th IEEE International Conference on Industrial and Information Systems (ICIIS)*, pages 352–357, Aug 2011.

[JBB+09] M. Johnston, P. Baggia, D. C. Burnett, J. Carter, D. A. Dahl, G. McCobb, and D. Raggett. EMMA: Extensible multimodal annotation markup language. W3C recommendation, W3C, February 2009. http://www.w3.org/TR/2009/REC-emma-20090210/.

[JE89] J. Johnson and G. Engelbeck. Modes survey results. *ACM SIGCHI Bulletin*, 20(4):38–50, April 1989.

[JM09] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, October 2009.

[Jun14] D. Junger. Transforming a state chart at runtime. In *Proceedings of the 1st EICS Workshop on Engineering Interactive Computer Systems with SCXML*, 2014.

[Kas82] D. J. Kasik. A user interface management system. *SIGGRAPH Computer Graphics*, 16(3):99–106, July 1982.

[KL07] F. Kronlid and T. Lager. Implementing the information-state update approach to dialogue management in a slightly extended SCXML. *Proceedings of the SEMDIAL*, pages 99–106, 2007.

[KLC98] L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1–2):99 – 134, 1998.

[Kle52] S. Kleene. *Introduction to Metamathematics*. North Holland, 1952.

[KN14a] G. Kirstner and C. Nuernberger. Developing user interfaces using SCXML statecharts. *Proceedings of the 1st EICS Workshop on Engineering Interactive Computer Systems with SCXML*, pages 5–11, July 2014.

[KN14b] G. Kistner and C. Nuernberger. Developing user interfaces using scxml statecharts. *Engineering Interactive Computer Systems with SCXML*, page 5, 2014.

[Kur87] R. P. Kurshan. Complementing deterministic Büchi automata in polynomial time. *Journal of Computer and System Sciences*, 1987.

[LC62] J. C. R. Licklider and W. E. Clark. On-line man-computer communication. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, AIEE-IRE '62 (Spring), pages 113–128, New York, NY, USA, 1962. ACM.

[LE02] S. Larsson and S. Ericsson. GoDiS–issue-based dialogue management in a multi-domain, multi-language dialogue system. In *Demonstration Abstracts, ACL-02*, 2002.

[LHHR94] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.

[Lic60] J. Licklider. Man-computer symbiosis. *Human Factors in Electronics*, 1960.

[Lit09] M. L. Littman. A tutorial on partially observable markov decision processes. *Journal of Mathematical Psychology*, 53(3):119 – 125, 2009. Special Issue: Dynamic Decision Making.

[LLC+00] S. Larsson, P. Ljunglöf, R. Cooper, E. Engdahl, and S. Ericsson. GoDiS - an accommodating dialogue system. In *ANLP/NAACL 2000 Workshop*, pages 7–10, Morristown, NJ, USA, 2000. Association for Computational Linguistics.

[LMM99a] D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[LMM99b] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proceedings of the 3rd International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 331–347, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[LNP+09] D. Lalanne, L. Nigay, p. Palanque, P. Robinson, J. Vanderdonckt, and J.-F. Ladry. Fusion engines for multimodal input: A survey. In *Proceedings of the 2009 International Conference on Multimodal Interfaces*, ICMI-MLMI '09, pages 153–160, New York, NY, USA, 2009. ACM.

[Löc04] M. Löckelt. Dialogue management in the SmartKom system. *Kovens 2004*, 2004.

[LP99] J. Lilius and I. P. Paltor. The semantics of UML state machines. Technical report, Turku Centre for Computer Science, 1999.

[LPE98] E. Levin, R. Pieraccini, and W. Eckert. Using markov decision process for learning dialogue strategies. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 1, pages 201–204. IEEE, 1998.

[LRR03] J. A. Larson, D. Raggett, and T. Raman. W3C multimodal interaction framework. W3C note, W3C, May 2003. http://www.w3.org/TR/2003/NOTE-mmi-framework-20030506/.

[LT00] S. Larsson and D. R. Traum. Information state and dialogue management in the TRINDI dialogue move engine toolkit. *Natural language engineering*, 6(3&4):323–340, 2000.

[Mag82] M. Maguire. An evaluation of published recommendations on the design of man-computer dialogues. *International Journal of Man-Machine Studies*, 16(3):237 – 261, 1982.

[Mai78] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2), April 1978.

[Mar73] J. Martin. *Design of man-computer dialogues*. Prentice-Hall, 1973.

[May90] J. Mayes. Multimedia: A perspective. *IEE Colloquium on Multimedia: the Future of User Interfaces*, pages 1/1–1/3, November 1990.

[MBO+10] S. McGlashan, D. Burnett, M. Oshry, M. Deshmukh, X. Yang, M. Bodell, J. Carter, R. Hosn, P. Baggia, R. Akolkar, R. Auburn, M. Young, and K. Rehor. Voice extensible markup language (VoiceXML) 3.0. W3C working draft, W3C, December 2010. http://www.w3.org/TR/2010/WD-voicexml30-20101216/.

[MC01] W. E. McUmber and B. H. C. Cheng. A general framework for formalizing UML with formal languages. *ICSE*, pages 433–442, 2001.

[McC59] J. McCarthy. Programs with common sense: Mechanization of thought processes. In *National Physical Laboratory Symposium*, number 10 in 1, pages 75–84, 1959.

[MHC99] O. Madani, S. Hanks, and A. Condon. On the undecidability of probabilistic planning and infinite-horizon partially observable markov decision problems. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and the Eleventh Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence*, AAAI '99/IAAI '99, pages 541–548, Menlo Park, CA, USA, 1999. American Association for Artificial Intelligence.

[MLS97] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In *Advances in Computing Science - ASIAN '97, 3rd Asian Computing Science Conference, Kathmandu, Nepal, December 9-11, 1997, Proceedings*, pages 181–196, 1997.

[MLSH98] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. Implementing statecharts in PROMELA/SPIN. In *Proceedings of the 2nd Workshop on Industrial-Strength Formal Specification Techniques (WIFT '98)*, pages 90–101, Boca Raton, FL, USA, October 1998.

[MP90] Z. Manna and A. Pnueli. Tools and rules for the practicing verifier. STAN-CS-90-1321, Stanford University, Computer Science Department, 1990.

[MW98] M. T. Maybury and W. Wahlster. *Readings in Intelligent User Interfaces*. Morgan Kaufmann, 1998.

[Mye91] B. A. Myers. Separating application code from toolkits: Eliminating the spaghetti of call-backs. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology*, UIST '91, pages 211–220, New York, NY, USA, 1991. ACM.

[Mye98] B. A. Myers. A brief history of human-computer interaction technology. *Interactions*, 5(2):44–54, 1998.

[NC91] L. Nigay and J. Coutaz. Building user interfaces: Organizing software agents. *Proceedings of ESPRIT*, 91:707–719, 1991.

[NC93] L. Nigay and J. Coutaz. A design space for multimodal systems: concurrent processing and data fusion. In *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, pages 172–178, New York, NY, USA, 1993. ACM.

[Nes09] T. Nestorovič. Towards flexible dialogue management using frames. In *Text, Speech and Dialogue*, pages 419–426. Springer, Berlin, Heidelberg, 2009.

[Nes10] T. Nestorovic. General agent-based architecture for collaborative dialogue management. In *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, 2010.

[New68] W. M. Newman. A system for interactive graphical programming. *AFIPS Spring Joint Computing Conference*, pages 47–54, 1968.

[Nic69] R. Nickerson. Man-computer interaction: A challenge for human factors research. *Ergonomics*, 12(4):501–517, 1969.

[Nie87] J. Nielsen. Classification of dialog techniques. *ACM SIGCHI Bulletin*, 1987.

[Nol71] A. M. Noll. *Man-machine tactile communication*. Polytechnic Institute of Brooklyn, 1971.

[NPS09] A. Niewiadomski, W. Penczek, and M. Szreter. A new approach to model checking of UML state machines. *Fundamenta Informaticae*, 93(1-3):289–303, January 2009.

[NS88] J. G. Neal and S. C. Shapiro. Intelligent multi-media interface technology. *SIGCHI Bulletin*, 20:75–76, July 1988.

[NTD+89] J. G. Neal, C. Y. Thielman, Z. Dobes, S. M. Haller, and S. C. Shapiro. Natural language with integrated deictic and graphic gestures. In *Proceedings of the Workshop on Speech and Natural Language*, HLT '89, pages 410–423, Stroudsburg, PA, USA, 1989. Association for Computational Linguistics.

[NW05] A. Nguyen and W. Wobcke. An agent-based approach to dialogue management in personal assistants. In *Proceedings of the 10th International Conference on Intelligent User Interfaces*, IUI '05, pages 137–144, New York, NY, USA, 2005. ACM.

[Ols90] D. R. Olsen, Jr. Propositional production systems for dialog description. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 57–64, New York, NY, USA, 1990. ACM.

[Ovi99] S. Oviatt. Ten myths of multimodal interaction. *Communications of the ACM*, 42(11):74–81, November 1999.

[Ovi03] S. Oviatt. Advances in robust multimodal interface design. *IEEE Computer Graphics and Applications*, 23(5):62–68, 2003.

[Ovi06] S. Oviatt. Human-centered design meets cognitive load theory: Designing interfaces that help people think. In *Proceedings of the 14th Annual ACM International Conference on Multimedia*, MULTIMEDIA '06, pages 871–880, New York, NY, USA, 2006. ACM.

[Ovi12] S. Oviatt. Multimodal interfaces. In J. A. Jacko and A. Sears, editors, *The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications*, pages 286–304. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 2012.

[Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Hamburg, 1962.

[PJS78] H. L. Pick Jr and E. Saltzman. *Modes of Perceiving and Processing Information*. Psychology Press, 1978.

[Pnu86] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency, Overviews and Tutorials*, pages 510–584. Springer Berlin Heidelberg, 1986.

[Pos10] S. Poslad. *Ubiquitous Computing: Smart Devices, Environments and Interactions*. John Wiley and Sons, July 2010.

[PS91] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In *Theoretical Aspects of Computer Software*, pages 244–264. Springer, 1991.

[PS01] F. Paternò and C. Santoro. Integrating model checking and HCI tools to help designers verify user interface properties. In *Proceedings of the 7th International Conference on Design, Specification, and Verification of Interactive Systems*, DSV-IS'00, pages 135–150, Berlin, Heidelberg, 2001. Springer-Verlag.

[PSVW87] A. Prasad Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49(2-3):217–237, January 1987.

[PtH85] G. Pfaff and P. ten Hagen. Proceedings of the workshop on user interface management systems, 1985.

[QS82] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, UK, 1982. Springer-Verlag.

[Rab89] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[Rap64] B. Raphael. A computer program which "understands". In *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part I*, AFIPS '64 (Fall, part I), pages 577–589, New York, NY, USA, 1964. ACM.

[RBVB06] C. Rousseau, Y. Bellik, F. Vernier, and D. Bazalgette. A framework for the intelligent multimodal presentation of information. *Signal Processing*, 86(12):3696–3713, 2006.

[RE98] G. Rozenberg and J. Engelfriet. Elementary net systems. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 12–121. Springer Berlin Heidelberg, 1998.

[RNSW14] S. Radomski, T. Neubacher, and D. Schnelle-Walka. From Harel to Kripke: A provable datamodel for SCXML. In *EICS2014 Engineering Interactive Systems with SCXML Workshop Proceedings*, 2014.

[RPT00] N. Roy, J. Pineau, and S. Thrun. Spoken dialogue management using probabilistic reasoning. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, ACL '00, pages 93–100, Stroudsburg, PA, USA, 2000. Association for Computational Linguistics.

[RSW10] S. Radomski and D. Schnelle-Walka. Pervasive speech API demo. In Ieee, editor, *World of Wireless Mobile and Multimedia Networks (WoWMoM), 2010 IEEE International Symposium on a Digital Object Identifier*, pages 1–2, 2010.

[RSW12] S. Radomski and D. Schnelle-Walka. VoiceXML for pervasive environments. *International Journal of Mobile Human Computer Interaction*, 4(2):18–36, 2012.

[RSW13] S. Radomski and D. Schnelle-Walka. Spatial audio with the W3C architecture for multimodal interfaces. In *Workshop on Speech in Mobile and Pervasive Environments 2013, in conjunction with mobileHCI*, 2013.

[RSWL+14] S. Radomski, D. Schnelle-Walka, T. Lager, J. Barnett, D. Dahl, and M. Mühlhäuser. Proceedings of the 1st EICS Workshop on Engineering Interactive Computer Systems with SCXML, July 2014.

[RSWRA13] S. Radomski, D. Schnelle-Walka, and S. Radeck-Arneth. A prolog datamodel for State Chart XML. In *Proceedings of the SIGDIAL 2013 Conference*, pages 127–131, 2013.

[RSWS14] S. Radomski, D. Schnelle-Walka, and L. Singer. A debugger for SCXML documents. In *EICS'14 Workshop on Engineering Interactive Systems with SCXML*, 2014.

[Rud99] A. Rudnicky. An agenda-based dialog management architecture for spoken language systems. *IEEE Automatic Speech Recognition and Understanding*, 1999.

[Sch97] U. Schöning. *Theoretische Informatik - kurzgefaßt*. Hochschultaschenbuch. Spektrum Akademischer Verlag, 3 edition, 1997.

[Sea69] J. R. Searle. *Speech acts: An essay in the philosophy of language*. University Press, 1969.

[Sea76] J. R. Searle. A classification of illocutionary acts. *Language in Society*, 5(01):1–23, April 1976.

[Sea80] J. R. Searle. Minds, brains, and programs. *Behavioral and brain sciences*, 3(03):417–424, 1980.

[SEB08] M. Schmitz, C. Endres, and A. Butz. A survey of human-computer interaction design in science fiction movies. In *INTETAIN '08: Proceedings of the 2nd international conference on INtelligent TEchnologies for interactive enterTAINment*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, January 2008.

[Sha64] J. C. Shaw. Joss: A designer's view of an experimental on-line computing system. In *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part I*, AFIPS '64 (Fall, part I), pages 455–464, New York, NY, USA, 1964. ACM.

[SKRP15] G. Saon, H. J. Kuo, S. J. Rennie, and M. Picheny. The IBM 2015 english conversational telephone speech recognition system. *CoRR*, abs/1505.05899, 2015.

[SMH95] L. Schomaker, S. Munch, and K. Hartung. A taxonomy of multimodal interaction in the human information processing system. Technical Report ESPRIT Project 8579 MIAMI, Nijmegen Institute of Cognition and Information, The Netherlands, February 1995.

[SRB05] H. Shi, R. J. Ross, and J. Bateman. Formalising control in robust spoken dialogue systems. In *3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 332–341. IEEE, 2005.

[Sut64] I. E. Sutherland. Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE design automation workshop*, pages 6–329. ACM, 1964.

[Swe88] J. Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive Science*, 12(2):257–285, February 1988.

[Swe10] J. Sweller. Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review*, 22(2):123–138, April 2010.

[SWR10] D. Schnelle-Walka and S. Radomski. An API for voice user interfaces in pervasive environments. In *SimPE 2010 Conference Proceedings*, September 2010.

[SWR11] D. Schnelle-Walka and S. Radomski. Augmenting VoiceXML with information from pervasive environments. In *Proceedings of SiMPE 2011, Joint Workshop with mobileHCI*, 2011.

[SWR12a] D. Schnelle-Walka and S. Radomski. Entwicklung multimodaler Anwendungen mit W3C Standards. *OBJEKTspektrum*, 03:28–32, 2012.

[SWR12b] D. Schnelle-Walka and S. Radomski. A pattern language for dialogue management. In *Proceedings of VikingPLoP 2012 Conference*, pages 122–141, 2012.

[SWR15] D. Schnelle-Walka and S. Radomski. Modern standards for VoiceXML in pervasive multimodal applications. In *International Journal of Mobile Human Computer Interaction*, Hershey, PA, USA, 2015. IGI Global.

[SWRAS15] D. Schnelle-Walka, S. Radeck-Arneth, and J. Striebinger. Multimodal dialogmanagement in a smart home context with SCXML. *2nd SCXML Workshop on Engineering Interactive Computer Systems with SCXML*, page 10, 2015.

[SWRBM15] D. Schnelle-Walka, S. Radomski, J. Barnett, and M. Mühlhäuser. Proceedings of the 2nd EICS Workshop on Engineering Interactive Computer Systems with SCXML, July 2015.

[SWRM14] D. Schnelle-Walka, S. Radomski, and M. Mühlhäuser. Multimodal fusion and fission within W3C standards for nonverbal communication with blind persons. In *14th International Conference on Computers Helping People with Special Needs*. Springer, 2014.

[SWRM15] D. Schnelle-Walka, S. Radomski, and M. Mühlhäuser. Modern standards for VoiceXML in pervasive multimodal applications. *Emerging Perspectives on the Design, Use, and Evaluation of Mobile and Handheld Devices*, page 22, 2015.

[SWRMua13] D. Schnelle-Walka, S. Radomski, and M. M uhlh auser. JVoiceXML as a modality component in the W3C multimodal architecture. *Journal on Multimodal User Interfaces*, pages 1–12, 2013.

[SWRRA13] D. Schnelle-Walka, S. Radomski, and S. Radeck-Arneth. Probabilistic dialogue management. In *Proceedings of VikingPLoP 2013 Conference*, pages 114–125, 2013.

[SWRRAM14] D. Schnelle-Walka, S. Radomski, S. Radeck-Arneth, and M. Mühlhäuser. Towards an information state update model approach for nonverbal communication. In *14th International Conference on Computers Helping People with Special Needs*. Springer, 2014.

[TA94] D. R. Traum and J. F. Allen. Discourse obligations in dialogue processing. In *Proceedings of the 32Nd Annual Meeting on Association for Computational Linguistics*, ACL '94, pages 1–8, Stroudsburg, PA, USA, 1994. Association for Computational Linguistics.

[Tay67] R. Taylor. Man-computer input-output techniques. *IEEE Transactions on Human Factors in Electronics*, HFE-8(1):1–4, March 1967.

[TCB+04] S. Tryphonas, J. Carter, D. Burnett, B. Porter, P. Danielsen, S. McGlashan, B. Lucas, J. Ferrans, K. Rehor, and A. Hunt. Voice extensible markup language (VoiceXML) version 2.0. W3C recommendation, W3C, March 2004. http://www.w3.org/TR/2004/REC-voicexml20-20040316/.

[Tes81] L. Tesler. The smalltalk environment. *Byte*, 6(8):90–147, 1981.

[tH91] P. J. W. ten Hagen. Critique of the seeheim model. In *Proceedings of the Workshop on User Interface Management Systems and Environments on User Interface Management and Design*, UIMS, pages 3–6, New York, NY, USA, 1991. Springer-Verlag New York.

[TL03] D. R. Traum and S. Larsson. The information state approach to dialogue management. In J. Kuppevelt, R. W. Smith, and N. Ide, editors, *Current and New Directions in Discourse and Dialogue*, volume 22 of *Text, Speech and Language Technology*, pages 325–353. Springer Netherlands, 2003.

[Tra96] D. Traum. Conversational agency: The Trains-93 dialogue manager. *Proceedings of Twente Workshop on Language Technology II: Dialogue Management in Natural Language Systems*, pages 1–11, 1996.

[Tra08] D. Traum. Approaches to dialogue systems and dialogue management. Lecture Notes, University of Southern California, `http://people.ict.usc.edu/~traum/ESSLLI08/`, 2008.

[Tur36] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[Tur50] A. M. Turing. Computing machinery and intelligence. *Mind*, pages 433–460, 1950.

[Tur04] M. Turunen. Jaspis - a spoken dialogue architecture and its applications. *PhD thesis, University of Tampere, Department of Information Studies*, 2004.

[Var01] M. Vardi. Branching vs. linear time: Final showdown. In *Proceedings of the 2001 Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2001 (LNCS Volume 2031*, pages 1–22. Springer-Verlag, 2001.

[vB88] W. R. van Biljon. Extending petri nets for specifying man-machine dialogues. *International Journal of Man-Machine Studies*, 28(4):437 – 455, 1988.

[VD97] A. Van Dam. Post-WIMP user interfaces. *Communications of the ACM*, 1997.

[vdB94] M. von der Beeck. A comparison of statecharts variants. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 128–148. Springer Berlin Heidelberg, 1994.

[vdB01] M. von der Beeck. Formalization of UML-statecharts. In *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, pages 406–421. Springer Berlin Heidelberg, 2001.

[VGBP13] K. Veselý, A. Ghoshal, L. Burget, and D. Povey. Sequence-discriminative training of deep neural networks. In *INTERSPEECH 2013, 14th Annual Conference of the International Speech Communication Association, Lyon, France, August 25-29, 2013*, pages 2345–2349, 2013.

[vGGSU12] R. van Glabbeek, U. Goltz, and J.-W. Schicke-Uffmann. On distributability of petri nets. In *Foundations of Software Science and Computational Structures*, pages 331–345. Springer Berlin Heidelberg, 2012.

[Vit67] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13(2):260–269, April 1967.

[VW86] M. Y. Vardi and P. Wolper. *An Automata-Theoretic Approach to Automatic Program Verification.* IEEE Computer Society, 1986.

[VW94] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

[Wei66] J. Weizenbaum. Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, 1966.

[Wei76] J. Weizenbaum. *Computer power and human reason: From judgment to calculation.* WH Freeman & Co, 1976.

[Wei91] M. Weiser. The computer for the 21st century. *Scientific American*, 1991.

[WHNK05] W. Wobcke, V. Ho, A. Nguyen, and A. Krzywicki. A BDI agent architecture for dialogue modelling and coordination in a smart personal assistant. In *IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 323–329, Sept 2005.

[YGTW13] S. Young, M. Gasic, B. Thomson, and J. D. Williams. POMDP-based statistical spoken dialog systems: A review. *Proceedings of the IEEE*, 101(5):1160–1179, May 2013.

[You00] S. J. Young. Probabilistic methods in spoken-dialogue systems. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 358(1769):1389–1402, April 2000.

[Zai14] D. A. Zaitsev. Toward the minimal universal petri net. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 44(1):47–58, 2014.