Proceedings of the 2nd EICS Workshop on Engineering Interactive Computer Systems with SCXML

Dirk Schnelle-Walka, Stefan Radomski, Jim Barnett, Max Mühlhäuser (eds.)

Fachbereich Informatik
Telekooperation
Prof. Dr. Max Mühlhäuser

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Table of Contents

PREFACE

The W3C MMI Working Group suggests the use of SCXML [1] to express the dialog control of multimodal applications. The overall approach has already been shown to be suitable i.e. to decouple the control flow and presentation layer in multimodal dialog systems [6]. It has been used in several applications to express dialog states [2], control handheld gaming consoles [3] or to easily incorporate information [5] from external systems.

With the first implementations of the suite of recommendations beginning to mature, more deployments in industry are starting to appear, e.g. for in-car infotainment systems, home automation and general dialog control. This gave rise to a new set of problems with regard to the operationalization and unveiled short-comings and new requirements we hope to discuss.

Despite these rather practical issues, there are still very interesting research questions revolving SCXML and related recommendations. The strong focus on state-charts could enable many formal approaches inapplicable to other dialog management techniques. Automated dynamic and static testing of dialogs expressed in SCXML or even model checking [4].

The workshop provided a forum to discuss submissions detailing the use of SCXML, in particular, multi-modal dialog systems adhering to the concepts outlined by the various W3C standards in general and related approaches of declarative dialog modeling to engineer interactive systems.

Our goal was to attract a wide range of submissions related to the declarative modeling of interactive multi-modal dialog systems to leverage the discussion and thus to advance the research of modeling interactive multi-modal dialog systems.

These proceedings contain the keynote from Jim Barnett and six submissions around the different aspects of engineering interactive systems with SCXML.

Format

The workshop was conducted as a two-tiered event: i In the first part the scientific contributions with regard to application and extensions of SCXML were presented, while ii the second part was in the format of an open-panel discussion, where suggestions that arose during the first part were detailed and elaborated.

ORGANIZERS AND PROGRAM COMMITTEE

The organizers are early adaptors of SCXML as well as leading experts from the SCXML working group.

Dirk Schnelle-Walka leads the "Talk&Touch" group at the Telecooperation Lab at TU Darmstadt. His main research interest is on multimodal interaction in smart spaces.
Stefan Radomski is a PhD candidate at the Telecooperation Lab at TU Darmstadt. His main research interest is about multimodal dialog management in pervasive environments.
Jim Barnett is a software architect at Genesys, a contact center software company. He is the editor of the SCXML specification.

Max Mühlhäuser is full professor and heads the Telecooperation Lab at TU Darmstadt. He has over 300 publications on UbiComp, HCI, IUI, e-learning and multimedia.

The list of program committee members is as follows:

- **Rahul Akolkar** (IBM Research, USA)
- **Kazuyuki Ashimura** (W3C, Japan)
- **Stephan Borgert** (TU Darmstadt, Germany)
- **Deborah Dahl** (Conversational Technologies, USA)
- **David Junger** (University of Gothenburg, Sweden)
- **Stephan Radeck-Arneth** (TU Darmstadt, Germany)
- **David Suendermann-Oeft** (DHBW Stuttgart, Germany)
- **Raj Tumuluri** (Openstream, USA)

ACKNOWLEDGEMENTS

The 2nd EICS Workshop on Engineering Interactive Systems with SCXML was an interesting experience where participants with all their different backgrounds had lively discussions about their applications and research regarding SCXML. If you contributed to it in any way, we are grateful for your involvement. We wish that these proceedings are a valuable source of information in your efforts. We hope that you will enjoy reading the following pages. We would like to thank the organizers and the program committee for all their work.

REFERENCES

1. Barnett, J., Akolkar, R., Auburn, R., Bodell, M., Burnett, D. C., Carter, J., McGlashan, S., Lager, T., Helbing, M., Hosn, R., Raman, T., Reifenrath, K., Rosenthal, N., and Roxendal, J. State chart XML (SCXML): State machine notation for control abstraction. W3C working draft, W3C, May 2014. http://www.w3.org/TR/2014/WD-scxml-20140529/.

2. Brusk, J., Lager, T., Hjalmarsson, A., and Wik, P. DEAL: dialogue management in SCXML for believable game characters. In *Proceedings of the 2007 conference on Future Play*, ACM (2007), 137–144.

3. Kistner, G., and Nuernberger, C. Developing User Interfaces using SCXML Statecharts. In *Workshop on Engineering Interactive Systems with SCXML* (July 2014).

4. Radomski, S., Neubacher, T., and Schnelle-Walka, D. From Harel To Kripke: A Provable Datamodel for SCXML. In *Workshop on Engineering Interactive Systems with SCXML* (July 2013).

5. Sigüenza Izquierdo, Á., Blanco Murillo, J. L., Bernat Vercher, J., and Hernández Gómez, L. A. Using scxml to integrate semantic sensor information into context-aware user interfaces. In *International Workshop on Semantic Sensor Web, In conjunction with IC3K 2010*, Telecomunicacion (2011).

6. Wilcock, G. SCXML and voice interfaces. In *3rd Baltic Conference on Human Language Technologies, Kaunas, Lithuania* (2007).

SCXML: Current Status and Future Prospects

Jim Barnett
Genesys
jim.barnett@genesyslab.com

INVITED TALK

We have completed all the technical work necessary for SCXML to become a recommendation. Only bureaucratic steps remain, and they are in the hands of W3C staff. We received 4 implementation reports with at least two for all mandatory features, as well as the ECMAScript data model and the HTTP Event I/O Processors. We did not receive the necessary two implementation reports for either the DOM Event I/O Processor or the XPath data model. We have therefore removed them from the specification and published them as separate Working Group Notes, in case anyone is interested in picking up work on them.

Once the SCXML is published as a W3C Recommendation, the Voice Browser Group will close since SCXML is the last item it is working on. It is likely that the Multimodal Working Group will inherit SCXML and other Voice Browser Group specifications, so future work on them could take place in that group. It would also be possible to form a Community Group. Such groups are loosely structured and produce Reports rather than standards-track documents, but they allow people to work quickly with a minimum of bureaucracy. They would be particularly well suited to work that is peripheral to the core standard, for example defining the IDL for the interpreter to serve as an interface to GUI tools. If three or four implementations were interested in collaborating on such an IDL, it could be produced quickly in a Community Group.

We can also consider extensions to the SCXML standard itself. There was very little interest in the XPath data model when it was dropped from the specification, so it is not a candidate for future work. It is disappointing that the SCXML Recommendation lacks two data models since the Voice Browser Group went to great lengths to define the language to abstract away the details of the data model. One can ask whether this abstraction was successful since the standard contains only a single data model. However in practice implementations are not having difficulty defining their own data models, so I think it is safe to say that SCXML has been shown to have a pluggable data model.

EICS'15 Workshop, Engineering Interactive Systems with SCXML, June 23, 2015, Duisburg, Germany

There was interest in the DOM Event I/O Processor, though unfortunately none of it came from browser vendors. The current definition of this processor is purely theoretical, in the sense that it was produced before anyone had an implementation, so it may well need significant modification before it can be truly useful. Such an effort might also be suitable for a community group.

Other possibilities for future work include making the preemption algorithm pluggable. The Voice Browser Group considered this option, but decided to make the execution of SCXML markup as deterministic as possible. One question is whether a pluggable preemption algorithm would be of any more than academic interest. This is an area where individual implementations could experiment and produce useful results. It would be interesting to know if application developers would make use of a pluggable algorithm, or whether they would stick with the default. Another area for experimentation would be the parallel execution of executable content. The sequential, lock-step manner in which executable content is processed simplifies the definition and implementation of SCXML, but the processing of executable content could be parallelized without changing the semantics of the rest of the language. An externally modifiable data model would also be useful, as well as compatible with Harels original state machine semantics. Leaving aside the system variables, an SCXML data model can only be modified by executable content, and the executable content can only be executed as the result of the state machine starting up or taking a transition. Thus a SCXML state machine cannot have a data element representing the continuously changing value of an external sensor for example a thermometer. More precisely, an application author cannot write a transition with a condition like `cond="temp>70"` and expect the transition to be taken whenever the temperature goes above 70. Instead, the developer would have to ensure that the sensor injected an external event each time the temperature changed, and then modify his application to check for that event and the new temperature value. This is feasible, but hardly elegant. It therefore might make sense to define an implicit data changed event that platforms could generate whenever the data model was modified. This implicit event would not be seen by mark-up, but would trigger re-evaluation of all enabled eventless transitions. A number of questions would need to be addressed before such implicit events could be incorporated into SCXML, most specifically how they would be interleaved with existing internal and external events.

It is interesting to consider the relationship between state machines and planning systems. Any given plan looks a lot like

a state machine, with the goals representing states and the actions being transitions. For example, given a complex state machine representing a reactive system, it would be possible to use a planner as a help system. If the user is in state S_1 and wants to get to state S_2, the system could present him with a plan on how to get from S_1 to S_2. For this to be possible, the state machine must be statically analyzable, meaning that for any pair of states S_x and S_y, it is possible to determine if there is any sequence of events that will move the state machine from S_x to S_y. This implies restrictions on the data model and particularly on conditions on transitions. Static analyzability will also put restrictions on dynamic state schemes, in which states are added or removed from the state machine at runtime. (Such schemes are not part of the SCXML standard, but have been discussed at this workshop both this year and last.) It would be interesting to know which dynamic state schemes preserve static analyzability.

Requirements and Challenges of In-Vehicle Spoken Dialog Systems Specification from an Industrial Viewpoint

Patricia Braunger, Hansjörg Hofmann, Steffen Werner, Benoit Larochelle
Daimler AG
{patricia.braunger, hansjoerg.hofmann, steffen.werner, benoit.larochelle}@daimler.com

ABSTRACT
The development process of in-vehicle spoken dialog systems (SDS) is characterized by intensive cooperation of different parties. Voice user interface designers specify the system requirements which are then implemented by integration experts. Therefore adequate specification formats are an important factor for the development process. Because the complexity of SDS quickly increases, new formats are of interest. Specification formats strongly influence the possibilities of specification and the quality of the final product and should be therefore based on the requirements and challenges. The aim of the paper is to outline the requirements of the modelling in-vehicle SDS and to explore the challenges that follow, such as specifying natural language input.

Author Keywords
Specification, Spoken Dialog System, Industry, Natural Language Input

ACM Classification Keywords
D.2.m Miscellaneous

INTRODUCTION

Nowadays, cars without modern and extensive infotainment systems are hardly imaginable. Voice control becomes more and more an important role in cars as ensuring safety by allowing the driver to keep his hands on the wheel and his eyes on the road. Since spoken dialog systems (SDS) consist of voice user interaction and of graphical user interaction, they are considered as multimodal systems. Due to an expanding scope of infotainment functions the number of voice control features is constantly increasing and from the multiple interfaces result complex systems. Because the core competencies of automobile manufacturers lie in the developing of vehicles and user experience (UX) development, software is often implemented by suppliers [5]. They define their requirements in a functional specification which is the basis for integration experts to implement the software. Thereby the main task of the specification is to describe the dialogs which are determined by customer demands and market requirements.

EICS'15Workshop, Engineering Interactive Systems with SCXML, June 23, 2015, Duisburg, Germany

At predefined intervals, the automobile manufacturer receives software versions from the supplier. The client receives the software as a "black box". This is why the manufacturer verifies the received software versions on the basis of the specification. The manufacturer informs the supplier about faulty system behavior or conceptual changes. These iterations are repeated until the end of the development process. The development follows the established V-model [8]. Figure 1 shows the development process and the interaction of the client and the supplier in the V-model.

It follows that the quality of the product is directly connected with the quality of the specification. Thus, the selection of an adequate specification format is of great significance for an effective development process.

Specification can be used in different ways in the development process. It is common in automobile industry to use specification for a formal description of the system. Moreover, a formal description of dialogs can be used to automatically generate test cases and to simulate the dialogs. Therefore, a machine-readable format could be of interest [3]. Two formats are common for a formal description of dialogs: finite-state-based approaches and flow charts. Flow charts describe the system performance by a sequence of single activities. Finite-state-based approaches describe the system performance as system states affected by events and conditions [9].

With an increasing know-how in the field of dialog systems and progressive technology development, new challenges for specification are arising. While today command based systems are common, a more natural mood of communication is aspired. That is, the system should be able to interpret any spoken user utterances. Specifying natural language input is a critical issue because a limited amount of grammatically well-structured utterances becomes an unlimited amount.

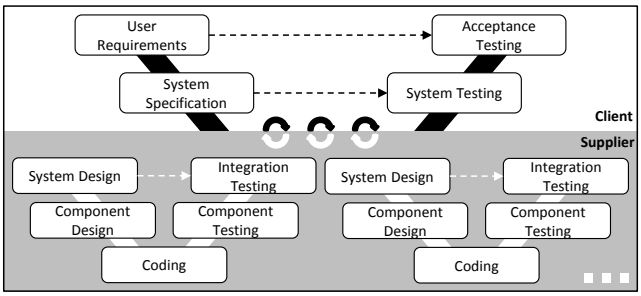

Figure 1: Interaction between system supplier and client [8]

From this perspective, the present specification formats quickly show limitations while new formats like SCXML [1] are taken into consideration. This paper focuses on requirements and the corresponding challenges of specification from a certain industrial viewpoint. Figure 2 gives an overview of specification aspects which are considered when choosing or assessing specification formats. The outlined requirements and open issues refer to formal description of dialogs.

The paper at hand is structured as follows. The next section presents an overview of specification requirements that follow from the development process and the requirements related to in-vehicle SDS. Then, we focus on current and future challenges of SDS specification and provide an outlook for future development. The last section summarizes the stated points.

REQUIREMENTS OF SPECIFYING IN-VEHICLE SPOKEN DIALOG SYSTEMS

To guarantee a high quality, the specification must meet several requirements that are mentioned in this section. New specification formats must fulfill these requirements.

Software errors often result from a wrong interpretation of informal or vague information [6]. A complete and consistent description of the implemented system is indispensable, because different parties are involved in the development of in-vehicle software. Such a description clarifies the goals and all the expectations of the system to be developed [4]. To ensure a complete, consistent and unambiguous specification, a formal specification is needed [6, 4]. Because spoken dialog systems are multimodal, a formal dialog description must include user actions like haptic activation or voice entries and system reactions like speech output, text output and other system activities.

In addition, a specification of in-vehicle SDS must allow testability of the complete dialogs to verify the software. The combination of manual testing and automated testing is a common testing practice. Manual testing is still necessary, not at least to evaluate usability. As the visual presentation of dialogs facilitates the analysis and editing of dialogs, commonly graphic notations are used for the formal specification of dialogs [10, 8]. Flow charts or finite-state-machines (e.g. according to the UML notation) are suitable [5, 8]. These possibilities allow a complete specification of all imaginable dialog states and testability of dialogs. It is important for manual testing that the formal description be readable also by non-IT-experts. Even though the complexity of the dialogs increases, the specifications must remain comprehensible. To ensure readability, a modular approach is pursued. That is, dialogs are subdivided into modules that are called in specified sequences [8].

Besides format requirements dialog requirements also influence the specification. In general, commercially used dialog systems require that no undesired finite states and no loops exist. Hence, a robust dialog is especially important for in-vehicle SDS to enable the driver to focus on the road traffic [5].

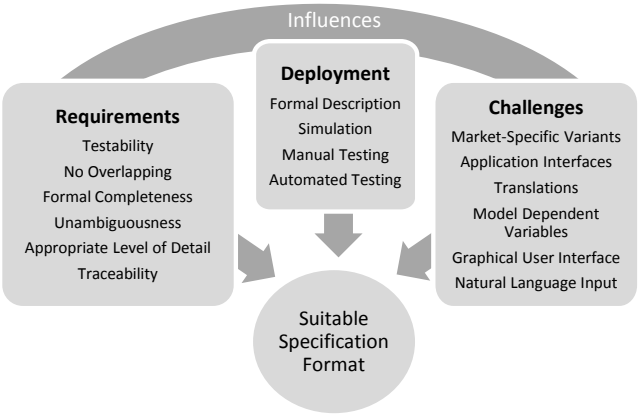

Figure 2: Aspects of in-vehicle SDS specification that influence the selection of a suitable specification format

Also, a longer dialog has to be initiated in the case of incomplete user input or unrecognized user input. From these requirements follow that all possible dialog states and transitions must be completely specified [5]. A robust dialog should also avoid overlapping of different voice entries. In case of overlaps the user would be directed to a wrong dialog. Hence, a grammar must include all possible user utterances.

CHALLENGES REGARDING IN-VEHICLE SPOKEN DIALOG SYSTEMS SPECIFICATION

Current Challenges

The formal description of dialogs and the requirements of specification in the context of developing in-vehicle systems faces several challenges.

The design of SDS has to fulfill market-specific requirements, because dialogs may differ depending on the target market or country. For example, entering an address in China differs from doing it in Europe. Due to these variations, different dialog variants must be specified. The differences often refer to particular sequences within one dialog, such as the order of the elements when entering an address. Besides different dialog flows, markets can vary in the nomenclature within dialogs. For example, Japan is subdivided into provinces while the US consists of states. These regional distinctions have to be considered in dialog flows. Moreover, certain functions are available only in specific countries, e.g. Sirius Satellite Radio in the US. In addition to markets and country variations, different car model series with different functions and technologies have to be taken into consideration.

Furthermore the specification has to challenge many different user utterances. Thus, the occurrence of overlapping commands becomes more probable with the increase in controllable features. The same problem must be resolved in all supported languages. Because of a worldwide distribution, numerous languages and therefore many translations have to be managed which are included

in the specification document. Translations often result in overlapping commands because of wrong interpretation or similar wording.
Besides voice input and output, a formal description of dialogs must consider haptic user actions and graphical output. It is important to formally define the interaction transitions between the graphical user interface and the voice user interface.
The definition of interfaces to other functional applications such as navigation, phone, tuner or player is also challenging. A formal specification must include a formal (machine-readable) definition of interface calls as the experts have to integrate different application modules.
Due to the illustrated challenges, it is quite difficult to fulfill the requirements of specification because a complete and unambiguous specification often results in unreadability and formal incompleteness.
The technical progress in recent years has changed user expectations of SDS. At the beginning only single-word-commands were possible. Then, "Command & Control" enabled the user to speak short phrases. Following improvements in automatic speech recognition recent systems are able to recognize natural language input in restricted domains (see Apple's Siri[1]) [12]. Thus, a more natural human machine interaction is demanded by users. With the claim to a natural mode of communication new challenges arise for the specification formats.

New Challenge Natural Language Input

Specification of SDS has to fulfill several requirements and faces several challenges. Figure 3 shows the current difficulty. Up to now, the requirements and challenges are more or less balanced. That is, current specification is still appropriate. The development of a more natural human machine interaction leads to significant leaks of current specification.
The term natural language input refers to the properties of human speech. The most important properties of natural language compared to commands are listed in the following [11, 2]:

- Flexible sentence constructions, e.g. flexible constituent order, different sentence types
- Variable wording
- Anaphora
- Lack of precision
- Colloquial language (contextual knowledge needed)
- Ambiguity

In addition, more naturalness of SDS requires a consideration of spoken language properties. Spoken language differs from written language and grammatically well-structured utterances in spontaneity. Spontaneous speech production is characterized by interruptions, hesitation phenomena and grammatically incorrect sentences [11]. These characteristics must be considered when developing SDS to make natural language input possible. Natural language input results in a not limited amount of possible user utterances. As mentioned above, specifying all possible user utterances is common practice for in-vehicle spoken dialog systems. While users demand a larger variety of possible utterances, grammars increase significantly in complexity. This leads to an unreadable and untraceable specification. Therefore, the current specification is no more appropriate. Natural language input breaks the equilibrium of current specification as Figure 3 shows.
The properties of natural language raise the question of how to specify all possible user input utterances and the corresponding system reaction. One possibility of specifying natural language input is to extract the meaning of imaginable utterances. In doing so, attributes and corresponding values could be given instead of utterances [7]. It should be kept in mind that software tests are necessary. Usually, when evaluating the speech input capabilities of a SDS, people assess the recognition performance on an utterance [7]. It is common to assess the result of the automatic speech recognition module (ASR). Specifying semantically by attribute-value-pairs means no verbalized user utterances. This raises the question, how to test the functionality of the system without knowing the capability. The procedure of specifying semantically seems not appropriate for testing the software in the normal manner. When specifying semantically, the language understanding (LU) performance becomes more important because the LU module extracts the meaning of the recognized utterance. The LU performance is often assessed on the basis of attribute-value-pairs [7]. Specification of natural language input requires therefore a change in measurement. In any case, it is unclear how to generate automatically test cases on the basis of attribute-value-pairs and how to test the LU performance. Collecting data from crowdsourcing or smaller studies might be one possibility identifying possible user utterances.
In the course of further research we will focus on process steps of developing more natural spoken dialog systems. We will concentrate especially on the problems of specifying and testing natural language input. A possible solution might be a semantical specification. We will assess specification formats in regard to specification of natural language input and the possibility of automatically generating test cases. In addition, further research will investigate test procedures such as collecting data from crowdsourcing or smaller studies. We will also consider other development process methods such as SCRUM and specifying user stories.

[1] http://www.apple.com/ios/siri/

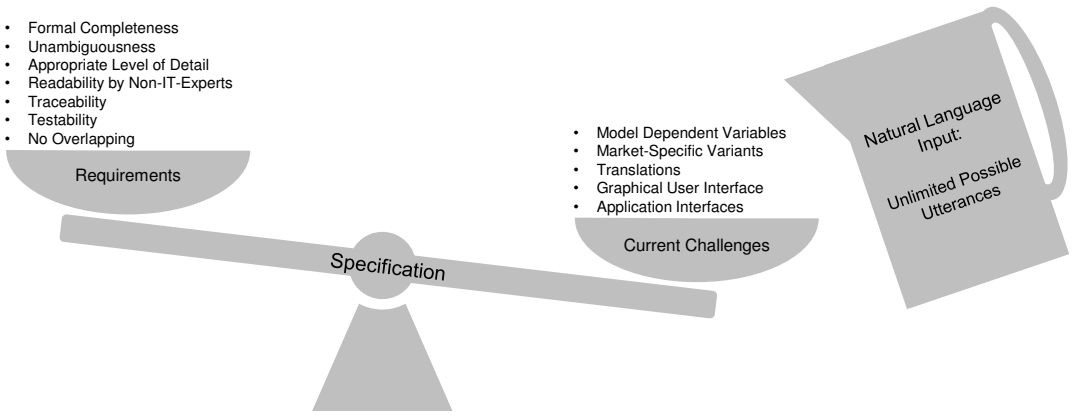

Figure 3: Indifferent equilibrium of requirements and challenges.

SUMMARY

A formal specification clarifies goals and expectations and prevents wrong interpretations. Complete, consistent and unambiguous specifications are necessary because several parties are involved in developing in-vehicle SDS. Because of new challenges that are associated with natural language input, current specification formats show limitations. Therefore, the requirements and challenges mentioned in the paper require the use of new specification formats. A formal completeness, unambiguousness, an appropriate level of detail, readability by non-IT-experts, no overlapping and traceability of complex dialogs are the given requirements. Some of these requirements cause challenges for current specification. To provide a complete specification, market demands, model dependent variables and translations should be taken into account. A great number of functions operated by voice, the graphical user interface and the definition of application interfaces were also mentioned as a challenge. Readability often suffers when one prepares a complete, detailed and unambiguous specification. On the one side, a full description of dialogs is required. On the other side, the described properties of natural language result in a very extensive grammar and an unreadable specification. Hence, specifying natural communication raises several challenges. Specifying all possible utterances or even finding them out, pose a problem to current specification formats and processes. The specification format should also assist with the testing of this complex system.

REFERENCES

1. Barnett, J., Akolkar, R., Auburn, R., Bodell, M., Burnett, D. C., Carter, J., McGlashan, S., Lager, T., Helbing, M., Hosn, R., Raman, T., Reifenrath, K., Rosenthal, N., and Roxendal, J., State Chart XML (SCXML): State machine notation for control abstraction. W3C Working Draft (2013).
2. Berg, M., Natürlichsprachlichkeit in Dialogsystemen. *Informatik-Spektrum, 36*, 4 (2012), 371–381.
3. Bock, C., Einsatz formaler Spezifikationen im Entwicklungsprozess von Mensch-Maschine-Schnittstellen. Technische Universität Kaiserslautern (2007).
4. Brost, M., Automatisierte Testfallerzeugung auf Grundlage einer zustandsbasierten Funktionsbeschreibung für Kraftfahrzeugsteuergeräte. expert verlag (2009).
5. Hamerich, S., Sprachbedienung im Automobil - Teilautomatisierte Entwicklung benutzerfreundlicher Dialogsysteme. Springer (2009).
6. Haubelt, C., Teich, J., Digitale Hardware/Software-Systeme. Spezifikation und Verifikation. Springer (2010).
7. Hofmann, H., Intuitive Speech Interface Technology for Information Exchange Tasks. PhD thesis, Universität Ulm (2014).
8. Lütze, L., Modellbasierter Testprozess der akustischen Mensch-Maschine-Schnittstelle eines Infotainmentsystems im Kraftfahrzeug. PhD thesis, Universität Stuttgart (2013).
9. Lütze, L., Werner, S. Qualitätssicherung im Linguatronic Entwicklungsprozess – Modellbasiertes Testen auf Basis formaler Beschreibung von Sprachdialogsystemen. *Elektronische Sprachsignalverarbeitung (ESSV)*, *Tagungsband*, TUDpress (2012), 196-203.
10. Rumpe, B., Modellierung mit UML: Sprache, Konzepte und Methodik. Springer (2011).
11. Schwitalla, J., Gesprochenes Deutsch. Erich Schmidt Verlag (2006).
12. Werner, S., Sprachdialogsysteme im Automobil – von der Theorie in die Serienproduktion. *Systemtheorie, Signalverarbeitung, Sprachtechnologie*, TUDpress (2013)

Multimodal Dialogmanagement in a Smart Home Context with SCXML

Dirk Schnelle-Walka
S1NN Gmbh & Co KG
Germany
dirk.schnelle-walka@s1nn.de

Stephan Radeck-Arneth
TU Darmstadt
Germany
stephan.radeck-arneth@cs.tu-darmstadt.de

Jürgen Striebinger
Cibek GmbH
Germany
juergen.striebinger@cibek.de

ABSTRACT

The W3C MMI architecture is a recommendation for a common conceptualization for multimodal interaction focusing on the components involved and the messages passed between them. However, the standard does not cover integration of multimodal fusion and fission as addressed in the multitude of prototypical implementations, frameworks and applications prior to this standard. In this paper we describe an integration of current multimodal fusion and fission into this standard with an SCXML dialog manager in the context of smart homes.

Author Keywords

multimodal architecture; dialog management; SCXML; MMI, EMMA, smart homes

ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

INTRODUCTION

Our homes are becoming smarter. Controlling devices in these smart homes by multiple modalities is already a reality. A typical architecture of such a smart home is shown in figure 1.

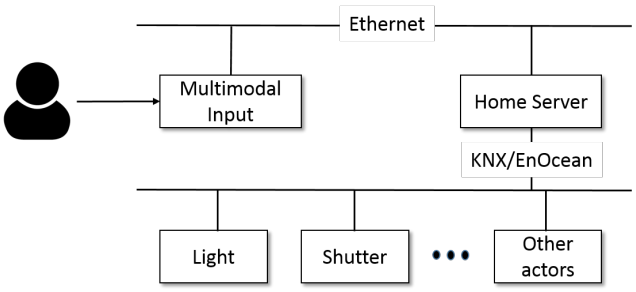

Figure 1. Typical architecture of a Smart Home

EICS'15, June 23–25, 2015, Duisburg, Germany.

Multimodal input into these system should follow the concepts shown in figure 2.

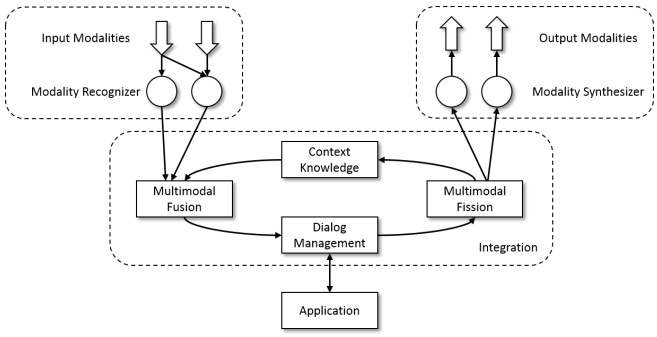

Figure 2. High-level multimodal architecture, adapted from [5]

Researchers as well as industry developed a plethora of deviations thereof for decades [5]. This makes it harder to reuse established knowledge and best-practices. With the advent of the *W3C Multimodal Architecture and Interfaces* recommendation [3] a promising candidate to standardize multimodal systems is available. A first analysis of the *nuts and bolts* is provided by [20]. However, the actual approach on how input coming from multiple sources is fused into a coherent meaning (multimodal fusion [12]) as well as state-of-the-art concepts on how to deliver information using more than a single available modality (multimodal fission [12]) is vaguely specified in the respective standards. Some first thoughts are described by Schnelle-Walka et al.in [21] and Ruf et al. [19]. This paper builds on [19]. While the latter deals on the aspects of implementing multimodal fusion and fission within the W3C architecture, this paper focuses on dialog management in the same setting. The W3C suggests the use of SCXML [2] as the dialog manager which has been proven to be suitable to decouple the control flow and presentation layer in dialog management [25]. It has been used in several applications to express dialog states [4] or to easily incorporate external information [22].

RELATED WORK

A similar multimodal architecture was proposed by Fernandez et al. in [9] for the smart home. They combined the model-view-presenter pattern (MVP) with a service oriented paradigm. MVP is derived from the Model-view-controller pattern (MVC) and synchronizes different views with a presenter component. The fusion and fission functionality are

integrated in the presenter. The communication between presenter and modalities is realized via an event communication channel within the OSGi infrastructure. The user preferences, available entities and execution context are integrated in a multimodal interface instance. The prototype supports visual, haptic, gesture and voice. The platform uses OSGi service and is implemented as an OSGi service factory. In comparison to our work, the components of the resulting architecture are coupled and some components are not open-source.

In [6] Cohen et al. describe Sketch-Thru-Plan (STP) as another, but more recent closed-source multimodal system. It combines gesture, handwriting, touch and speech. For robust command & control speech recognition, a grammar based speech recognition was selected. STP enables collaborative planning between several users through the multimodal interface. For speech recognition the Microsoft Speech Engine (SAPI) of Windows 7/8 is used. Push-to-talk (PTT) is correlated with the interaction of the screen by touch events to reduce the effects of conversational speech. Cohen et al. consider statistical language models (SLM) for future extensions, but the authors consider grammar-based recognition to be sufficient, since the users are trained to use a specific vocabulary. However, STP is not designed for reusability.

Another WWHT-based [18] multimodal fission contribution were introduced by Costa et al. [7]. The developed GUIDE system supports elderly users to communicate through employing appropriate modalities. The system uses video, audio and haptic interfaces and is integrated with the television. The multimodal fission selects the interface best suited for the communication for several users individually. However the system focus on multimodal fission. The integration in a multimodal architecture including fusion is out of scope for the work described in this paper.

As an alternate approach to WWHT, Pitsikalis et al. [14] trained Hidden Markov Models for multimodal fission. HMMs also proofed to be useful when fusing input from multiple modalities. Potamianos et al. [15] rely on HMM for audiovisual ASR, i.e. multimodal fusion. We consider HMMs for a later stage of the project. In order to actually train the models sufficient data is required which may be obtained by the rule based approach described later on.

MULTIMODAL ARCHITECTURE WITH THE W3C

The recommended architecture by the W3C decomposes a multimodal application into an interconnected structure of *interaction managers* (IM) for dialog control and *modality components* (MC) for in- and output. An implementation is formulated as a set of control documents reflected e.g. in SCXML for the interaction managers and a set of presentation documents with modality-specific markup for the modality components [21]. A topmost root controller document articulates the global dialog and instantiates modality components as required. Each modality component can, in turn, again be an interaction manager, managing more fine granular concerns of dialog control, such as error correction or even sensor fusion/fission.

Multimodal Fusion

In multimodal systems users are capable to express their dialog move by more than a unique modality. Multimodal fusion synthesizes the input arriving from the varying modalities into a unified semantic interpretation that declares the user's interaction intent [1]. Multimodal fusion mainly serves two purposes (i) provide an abstraction that enables usage of the provided information regardless of the used modality and (ii) derive a meaning thereof. Hence, the fusion engine needs an application independent representation of the current application context to infer meaning [8] which is available with EMMA[1]. Following the MMI W3C architectural pattern, the MCs send their input as EMMA events to the upper IM for further refinement.

For the fusion engine, Bui [5] infers in his survey paper a high-level view onto multi-level fusion as shown in Figure 3. According to Bui, this type of fusion consists of two main lev-

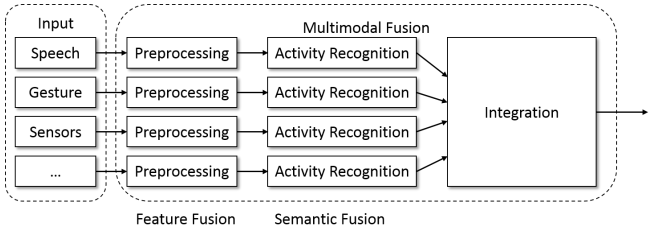

Figure 3. Multi-Level fusion, adapted from [5]

els of fusion: (i) feature-level fusion and (ii) semantic-level fusion. In the feature-level fusion the features as an output of the modality component are transformed into a modality-independent format. The subsequent semantic fusion decides, in a frame-based fusion manner [24], if the provided information is already sufficient to execute an action. In case it is insufficient, it is stored for later integration. The last integration step fuses the received semantic information into a coherent meaning. If required information is not received within a predefined time span, the available information is also forwarded to the upper IM, e.g. to initiate actions to ask explicitly for the missing piece.

Multimodal Fission

Multimodal systems provide for the consolidated or alternative application of separate input modalities and to choose output modalities most suitable for a given context. Alongside the benefits nominated by Oviatt and Cohen [13], particularly that the system increases stability and is fewer prone to errors, a suitable selection and combining of output modalities has the opportunity to facilitate or allow communication. One concept for fusion was established as WWHT by Rousseau et al. [17] (see Figure 4). (i) **W**hat is the information to render, (ii) **W**hich modalities should we utilize to present this information, (iii) **H**ow to roll out the information applying these modalities and (iv) and **T**hen, how to manage the progress of the deriving presentation [18]. These questions also shape the processing stages during fission and are described in [21] as follows:

[1] `http://www.w3.org/TR/EMMA/`

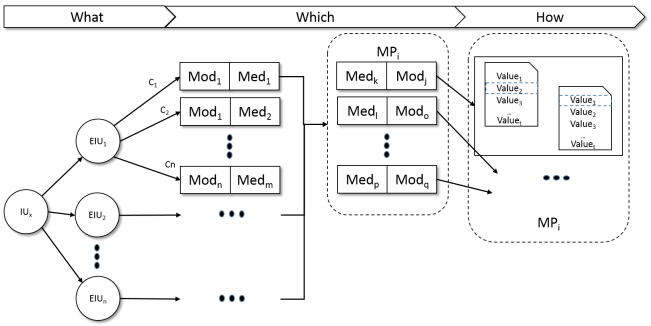

Figure 4. Multi-Level Fission with WWHT, derived from [17]

In the *What* level, a message is split into basic information unit.

The *Which* level offers for the choice of suitable modalities founded on rules and the affordances of the unique modalities.

In the *How* level, the output is actually rendered. This is achieved by the specified MCs.

Dialogmanagement with SCXML

Throughout the W3C MMI architecture recommendation, SCXML is stated as the preferred option for controller documents. At its origin, SCXML is a specification of finite state machine with parallel and nested states, as defined by Harel et al [11]. An SCXML implementation can, e.g., initiate actions when states are entered or transitions are taken. These actions include updating an internal data model or submitting events to other components. Transitions can be guarded by conditions, their evaluation is elicited via internal or external events.

In essence, SCXML is able to design dialog behavior even in the uppermost IM as the current application or nested in MCs e.g. for form filling or local error correction.

IMPLEMENTATION

In this section, we describe our implementation and some short-comings as a proof-of concept of the theoretical approaches mentioned above. We rely on OSGi as it used in the smart home controller by Cibek[2]. OSGi has proven to provide sufficient flexibility to address the different settings in actual deployments of smart homes. Multimodal fusion and fission are already described in [19] but are translated into English.

Multimodal Fusion

The nature of the WWHT approach to multimodal fusion is rule-based. A suitable framework for such rule-based systems is JBoss Drools[3]. It can be triggered by an OSGi bundle with an HTTP server to receive incoming MMI POST requests. The OSGi bundle injects the request into the knowledge base. Usually these MMI events will feature EMMA content that are extracted and injected as objects into the knowledge base. In the following, the request can be processed within the multimodal fusion via specialized rules. We used a separate file per modality to keep an overview. Consecutive processing includes further refinement of the received object and execution of Java code. At this stage of development we offered support for touch, speech, gesture and sensor input.

We had serious problems to carefully design the rules in order to avoid endless loops. Special care has to be taken to remove objects from the knowledge base when they are no longer needed.

Multimodal Fission

After an action has been executed an additional output may be requested or the application has to query for additional information. Note that an action is already one type of output so that an output is not needed in all cases.

We considered the modalities: text based output on a TV screen or a wall-mounted monitor, text-to-speech and avatar. Following the concept of WWHT the election in the *Which* stage first considered all available modalities. Based on the user's abilities, e.g. visual or aural impairment which are not uncommon in AAL settings, all modality-medium pairs with the corresponding inaccessible medium were removed. Then, all pairs that did not match the current context, were filtered. For instance, a sensor notified that the user started the interaction at a wall-mounted display and moved to the TV set. In this case, all subsequent output would be displayed using the TV screen.

In order to render the output in the *How* stage, corresponding MMI messages carry modality-specific markup, e.g. VoiceXML [**?**] for spoken interaction or BML [23] for the Avatar, in the data element.

Dialogmanagement with SCXML

Following the suggestion of the W3C MMI architectural pattern, the topmost interaction manager deals with high level tasks while fine-granular concerns of dialog control are handled by modality components acting as interaction managers. We employed uSCXML[4] as the SCXML interpreter. For a smart home control scenario the main task of the interpreter is on adapting to the current context, represented as states in an SCXML document (see figure 5). These states feature

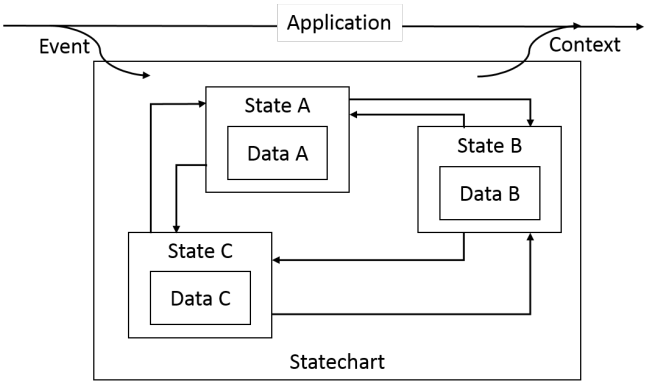

Figure 5. State-based context changer

[2] `http://www.cibek.de`

[3] `http://www.drools.org/`

[4] `http://github.com/tklab-tud/uscxml`

information relevant for the current situation. For example, consider grammars for a speech recognizer. Usually, speech input is recognized with higher accuracy if grammars are used [16, 10]. Incoming data from the fusion engine may cause a context switch, thus updating the application with new contextual information. The following SCXML snippet listing 1 shows how grammar can be injected into a speech recognizer using the `<send>` tag when the state is entered.

Listing 1. SCXML snippet

```
<scxml version="1.0">
 <state id="State_A">
  <onentry>
   <send type="application">
    <content>
     <output>[...]</output>
     <grammar>
      <rule>[...]</rule>
      <rule>[...]</rule>
     </grammar>
     [...]
    </content>
   </send>
  </onentry>
 <transition event="event.B" target="State_B"/>
</state>

<state id="state_B">
 [...]
</state>

[...]
</scxml>
```

The voice modality is handled by VoiceXML. Here we employed JVoiceXML [20] since it already features MMI communication capabilities. VoiceXML is a dialog manager on its own. For the topmost interaction SCXML-based interaction manager VoiceXML serves as a modality component that is capable of handling synthesized spoken output and speech input. This is achieved by sending corresponding MMI events to the voice browser with VoiceXML snippets in their data element. The grammars mentioned above are part of it. As a dialog manager, the voice browser itself is another interaction manager that independently handles the spoken dialog. This includes error handling within the modality and is in line with the principle of the *Russian Doll* as mentioned in the W3C MMI standard. Local error management however violates the concepts of multimodal error correction according to the architecture described in the high-level multimodal architecture (ref. to figure 2). Following the general idea that is described here, error management should be another iteration through the complete processing pipeline to allow for error correction by other modalities. We solved this by issuing MMI extension notifications to the multimodal fission in case of detected error correction within VoiceXML as shown in figure 6. This way, it is possible to exploit VoiceXML's

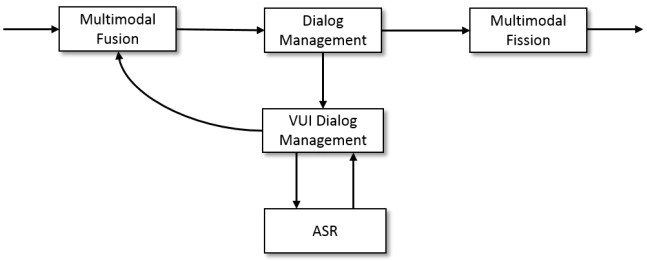

Figure 6. Use of multiple interaction managers

built-in capabilities to handle errors while being able, e.g., to show a puzzled avatar on a screen.

WALKTHROUGH

The following exemplifies the usage of the system. Imagine the following a scenario. In the morning, Alex enters the kitchen for the first time. The wall-mounted monitor shows how an avatar enters the scene to greet him: "Good Morning Alex. How may I help you?" As it is still dark inside the kitchen, Alex asks top open the shutter. While they are opening, the Avatar asks: "OK. Anything else I can help you with?"

This is reflected by our system as follows: Alex enters the kitchen and is recognized by a motion sensor. We employed an OPUS greenNet motion sensor for this purpose. This is fed into the fusion engine as an MMI new context request to initialize the dialog. The data attribute contains the location *kitchen* encoded in an EMMA document. A drools rule extracts the location data and updates the knowledge source. Additionally, it issues a start request to the upper IM, the context changer. The context changer is defined in a separate OSGi bundle with its own SCXML document *default.scxml* (see Listing 2).

Listing 2. Excerpt from default.scxml with information for the context changer

```
<state id="default-state">
 <onentry>
  <send type="drools">
   <content>
    <type>VXML</type>
    <currentState>default-state</currentState>
    ...
    <transitions>
     <contextChanges>
      <contextChange>cooking</contextChange>
      <contextChange>select-recipe</contextChange>
      <contextChange>morning</contextChange>
      ...
     </contextChanges>
     ...
    </transitions>
   </content>
  </send>
 </onentry>
 <transition event="event.cooking"
  target="cooking" />
 <transition event="event.select-recipe"
```

```
23    target="cooking" />
24   <transition event="event.morning"
25    target="morning" />
26   ...
27  </state>
```

We employed a special sort of IO processors that are used within the OSGi system. The context changer loads the corresponding *morning.scxml* file into uSCXML for further processing. First, the SCXML is validated to ensure that all transitions are valid. In addition it guarantees that interpretation of the SCXML document is only executed if all employed IO processors are available. This needs to be done to account for the modular nature of OSGi and to ensure that it can communicate with the environment.

The dedicated *morning* process then issues a start request to the multimodal fission bundle to let the avatar appear on the display and greet the user. The multimodal fission knows about the available modalities, avatar and speech output and sends a corresponding BML document to the avatar and a VoiceXML document to JVoiceXML. Knowledge about the available modalities and user preferences is configured as rules in a similar fashion as the multimodal fusion. The avatar is configured as a TTS engine in JVoiceXML which enables parallel use of this modality from within VoiceXML and by the multimodal fission engine.

The avatar movements of lower granularity follow a state-based approach to control the movements at a higher level in the context changer. This includes, appearance on the screen, leaving and some more high-level activities like putting on a cooking hat to adapt to the current context. Controlling the avatar at a lower level, like looking puzzled if a user input could not be matched or TTS output is triggered by dedicated SCXML scripts per dialog. Additionally, the avatar is used as output from JVoiceXML, requiring synchronization of lip movements and TTS output. We rely on the capabilities of the avatar's BML interpreter to merge this into smooth movements.

As a result the avatar appears on the display (controlled by uSCXML) and greets the user "Good morning." (controlled by JVoiceXML) as shown in figure 7. The grammars contained in the SCXML, as shown in listing 1, to continue the dialog are sent to JVoiceXML in the same start request. The VoiceXML document contains a single field with a prompt "How may I help you" and expects input for the received grammars. A dialog turn has one JVoiceXML session at maximum. Additional turns require processing by the complete chain from fusion to fission. Alternatively, it may be canceled by a cancel request if voice input is no longer required.

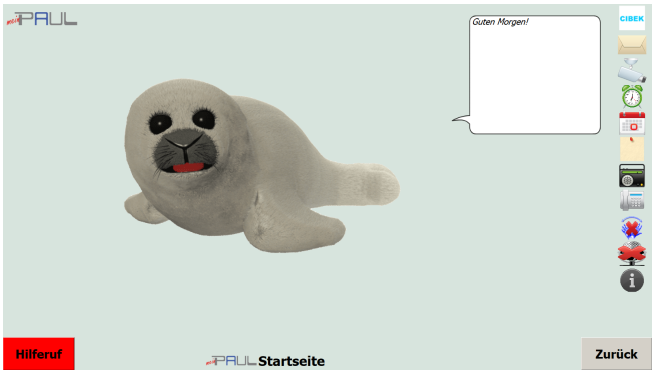

Figure 7. The avatar appears on the screen and greets the user

Once the user utters "Open the shutters" the command field gets filled. The dialog terminates and sends out a done notification containing the semantic interpretation as is is obtained from the grammar encoded as EMMA in the data tag. Here, we violate the concepts of MMI and do not send it to the invoking IM but to the multimodal fusion engine for further processing. This way, it is possible to keep the multimodal processing chain. Again the fusion engine fuses this command with the location information that is in the knowledge base and forwards the result to the dialog manager to actually execute the command. In this case, the SCXML triggers another OSGi bundle to send a corresponding command over KNX to the shutter (see figure 1). In addition, the avatar looks up and states "OK" following the same pipeline as described above.

SUMMARY AND CONCLUSION

In this paper we described a multimodal system to control smart homes employing open source software components. Some of the concepts have been described in previous publications which we now integrated into a fully functional prototype. It follows the W3C MMI architectural recommendation and integrates proven theoretical concepts of multimodal fusion and fission. SCXML turned out to be a good candidate for the topmost interaction manager. However, we had to violate the principle of a tree structure as suggested in the W3C MMI standard to enable multimodal error correction. Our final architecture is shown in Figure 8.

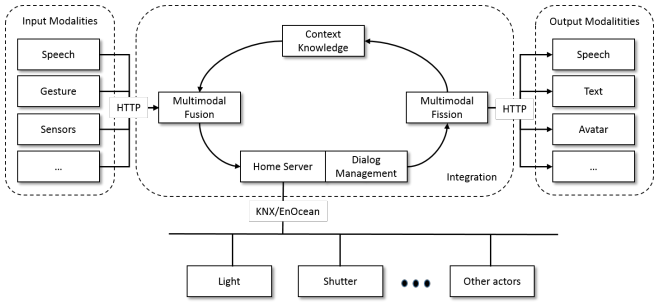

Figure 8. Implemented architecture

It is our hope that the results presented here stimulate the discussion around the W3C MMI standard to integrate multimodal fusion and fission as well as the interplay between multiple dialog managers within an architecture following the standard.

Currently, we are about to prepare a user study to evaluate the interaction in an ongoing project.

Acknowledgments

This work was partly supported by the Bundesministerium für Bildung und Forschung (BMBF), Germany under the programme "KMU-innovativ: Mensch-Technik-Interaktion für den demografischen Wandel".

REFERENCES

1. Atrey, P. K., Hossain, M. A., Saddik, A. E., and Kankanhalli, M. S. Multimodal fusion for multimedia analysis: a survey. *Multimedia systems 16*, 6 (2010), 345–379.
2. Barnett, J., Akolkar, R., Auburn, R., Bodell, M., Burnett, D. C., Carter, J., McGlashan, S., Lager, T., Helbing, M., Hosn, R., Raman, T., Reifenrath, K., Rosenthal, N., and Roxendal, J. State chart XML (SCXML): State machine notation for control abstraction. W3C working draft, W3C, May 2014. http://www.w3.org/TR/2014/WD-scxml-20140529/.
3. Bodell, M., Dahl, D., Kliche, I., Larson, J., Porter, B., Raggett, D., Raman, T., Rodriguez, B. H., Selvari, M., Tumuluri, R., Wahbe, A., Wiechno, P., and Yudkowsky, M. Multimodal Architecture and Interfaces. W3C recommendation, W3C, Oct. 2012. http://www.w3.org/TR/mmi-arch/.
4. Brusk, J., Lager, T., Hjalmarsson, A., and Wik, P. DEAL: dialogue management in SCXML for believable game characters. In *Proceedings of the 2007 conference on Future Play*, ACM (2007), 137–144.
5. Bui, T. Multimodal Dialogue Management - State of the art. Tech. Rep. TR-CTI, Enschede, Jan. 2006.
6. Cohen, P. R., Kaiser, E. C., Buchanan, M. C., Lind, S., Corrigan, M. J., and Wesson, R. M. Sketch-Thru-Plan: A Multimodal Interface for Command and Control. In *Communications of the ACM*, vol. 58 (Apr. 2015), 56–65.
7. Costa, D., and Duarte, C. Adapting Multimodal Fission to Users Abilities. In *Universal Access in Human-Computer Interaction. Design for All and eInclusion, 6th International Conference, UAHCI*, Springer (2011).
8. Dourlens, S., Ramdane-Cherif, A., and Monacelli, E. Multi levels semantic architecture for multimodal interaction. *Applied Intelligence* (2013), 1–14.
9. Fernandez, M., Pelaez, V., Lopez, G., Carus, J., and Lobato, V. Multimodal Interfaces for the Smart Home: Findings in the Process from Architectural Design to User Evaluation. *Ubiquitous Computing and Ambient Intelligence* (2012), 173–180.
10. Gorrell, G., Lewin, I., and Rayner, M. Adding intelligent help to mixed-initiative spoken dialogue systems. In *ACL-02 Companion Volume to the Proceedings of the Conference* (2002).
11. Harel, D., and Politi, M. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, Inc., Aug. 1998.
12. Landragin, F. Physical, semantic and pragmatic levels for multimodal fusion and fission. In *Proceedings of the Seventh International Workshop on Computational Semantics (IWCS-7)* (2007), 346–350.
13. Oviatt, S. L., and Cohen, P. R. Multimodal Interfaces That Process What Comes Naturally. *Communications of the ACM 43*, 3 (2000), 45–53.
14. Pitsikalis, V., Katsamanis, A., and Papandreou, G. Adaptive multimodal fusion by uncertainty compensation. In *IEEE Transactions on Audio, Speech, and Language Processing* (2009).
15. Potamianos, G., Huang, J., and Marcheret, E. e. a. Far-field multimodal speech processing and conversational interaction in smart spaces. *2008 Hands-free Speech Communication and Microphone Arrays, Proceedings* (2008), 119–123.
16. Rayner, E., Bouillon, P., Chatzichrisafis, N., Hockey, B. A., Santaholma, M. E., Starlander, M., Isahara, H., Kanzaki, K., and Nakao, Y. A methodology for comparing grammar-based and robust approaches to speech understanding. *Proceedings of Eurospeech-Interspeech, 9th European Conference on Speech Communication and Technology* (2005), 1103–1107.
17. Rousseau, C., Bellik, Y., and Vernier, F. WWHT: Un modèle conceptuel pour la présentation multimodale d'information. In *Proceedings of the 17th international conference on Francophone sur l'Interaction Homme-Machine*, ACM (2005), 59–66.
18. Rousseau, C., Bellik, Y., Vernier, F., and Bazalgette, D. A Framework for the Intelligent Multimodal Presentation of Information. *Signal Processing 86*, 12 (2006), 3696–3713.
19. Ruf, C., Striebinger, J., and Schnelle-Walka, D. Sprach- und Gestensteuerung fr das Smart Home. *JavaSPEKTRUM* (Mar. 2015).
20. Schnelle-Walka, D., Radomski, S., and Mühlhäuser, M. JVoiceXML as a Modality Component in the W3C Multimodal Architecture. *Journal on Multimodal User Interfaces* (Apr. 2013).
21. Schnelle-Walka, D., Radomski, S., and Mühlhäuser, M. Multimodal Fusion and Fission within W3C Standards for Nonverbal Communication with Blind Persons. In *Computers Helping People with Special Needs, 14th International Conference on Computers Helping People with Special Needs*, Springer (July 2014), 209–213.
22. Sigüenza Izquierdo, Á., Blanco Murillo, J. L., Bernat Vercher, J., and Hernández Gómez, L. A. Using SCXML to integrate semantic sensor information into context-aware user interfaces. In *International Workshop on Semantic Sensor Web, In conjunction with IC3K 2010*, Telecomunicacion (2011).

23. van Welbergen et al., H. BML 1.0 Standard. Standard, SAIBA, Apr. 2014. http://www.mindmakers.org/projects/bml-1-0/wiki#BML-10-Standard.

24. Vo, M. T., and Wood, C. Building an application framework for speech and pen input integration in multimodal learning interfaces. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)* (1996).

25. Wilcock, G. SCXML and voice interfaces. In *3rd Baltic Conference on Human Language Technologies, Kaunas, Lithuania* (2007).

State Machines as a Service
An SCXML Microservices Platform for the Internet of Things

Jacob Beard
Jacobean Research and Development
1732 1st Ave #25343
New York, NY 10128
jake@jacobeanrnd.com

ABSTRACT
The Internet of Things (IoT) describes a range of Internet-connected devices. In a typical IoT application, a fleet of IoT devices generates sensor data which is analyzed in real time to derive actionable intelligence; and IoT actuators are controlled in response to this intelligence. W3C SCXML can be used effectively as a domain-specific language to analyze IoT sensor data and control IoT actuators.

IoT has unique scaling requirements, and a solution is required to operationalize SCXML to support large-scale IoT deployments consisting of potentially millions of devices. State Machines as a Service (SMaaS) is a new category of Platform as a Service (PaaS) technology for simulating persistent state machines in a secure, distributed cloud computing environment. This platform enables SCXML to be operationalized in the cloud to control IoT networks of arbitrary size and complexity.

Author Keywords
SCXML; IoT; PaaS; LXC; Docker; Microservices

INTRODUCTION
Statecharts, a visual modelling language for describing timed, event-driven, stateful systems [6], has been used in applications including telecommunications [3], robotics [9], manufacturing [5], automotive systems [12], and user interfaces [7].

Statecharts is supported by industry standards, such as SCXML, a draft specification published by the W3C [2]. There exist a number of open source implementations of SCXML, including the SCXML Interpretation and Optimization eNgine (SCION), an implementation of SCXML in JavaScript [4]. SCION is used in production in enterprises for purposes including modeling complex user interface behaviour in financial web applications; managing navigation in multi-modal telephony applications; and controlling hardware in scientific instruments.

CHI'12, May 5–10, 2012, Austin, Texas, USA.

The Internet of Things
The Internet of Things (IoT) describes a new category of Internet-connected devices. These devices communicate via a range of wireless network protocols including WiFi, ZigBee, Z-Wave, and 6LoWPAN.

An example of a consumer IoT device is the Belkin WeMo Light Switch. This IoT light switch connects to a user's WiFi network, and can be controlled via a UPnP and SOAP application protocol.

SCXML for the Internet of Things
SCXML can interface with various IoT network protocols in order to parse sensor data and control devices actuators [11].

For example, a simple Statechart diagram can be seen in Figure 1. This Statechart models the behaviour of the WeMo light switch. It has three states: "on", "off", and "error", with transitions between the states on device events "device.turnOn", "device.turnOff", "device.error" and "device.reset". These events are emitted by the WeMo device via its UPnP SOAP API.

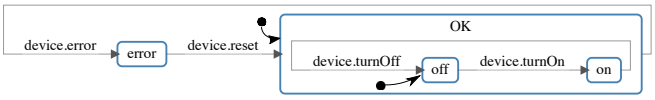

Figure 1. Light Switch Statechart

The corresponding SCXML for this Statechart diagram can be see in Figure 2. This SCXML may be run inside an SCXML execution environment.

STATE MACHINES AS A SERVICE
State Machines as a Service (SMaaS) is a new category of Platform as a Service (PaaS). PaaS provides services to deploy and run applications on cloud computing infrastructure without the need to manage the underlying infrastructure. SMaaS is a specialized PaaS that allows developers to deploy and run SCXML applications. SMaaS implements a custom API based on HTTP and REST

```
<?xml version="1.0" encoding="UTF-8"?>
<scxml
    xmlns="http://www.w3.org/2005/07/scxml"
    name="light-switch"
    datamodel="ecmascript"
    version="1.0">
  <state id="OK">
    <state id="on">
      <transition target="off"
          event="device.turnOff"/>
    </state>
    <state id="off">
      <transition target="on"
          event="device.turnOn"/>
    </state>
    <transition target="error"
          event="device.error"/>
  </state>
  <state id="error">
    <transition target="OK"
          event="device.reset"/>
  </state>
</scxml>
```

Figure 2. Light Switch SCXML

that allows developers to save an SCXML document to the SMaaS server; create state machine instances from the saved SCXML; and then send events to the instance. The instance will process sent events according to logic defined in the saved SCXML definition and the "Algorithm for SCXML Interpretation", which is described in the SCXML specification [2].

A SMaaS server can be run on distributed cloud computing infrastructure, such as Amazon EC2; or pushed to the "edge of the cloud", which is to say, run a hub on the user's local area network.

State Machines Instances as Microservices

In "Building Microservices", Sam Newman defines microservices as "small, autonomous services that work together." [10] He continues:

> ..the core concept of the service as a state machine is powerful. We've spoken before (probably ad nauseum by this point) about our services being fashioned around bounded contexts. Our customer microservice *owns* all logic associated with behavior in this context.
>
> When a consumer wants to change a customer, it sends an appropriate request to the customer service. The customer service, based on its logic, gets to decide if it accepts that request or not. Our customer service controls all lifecycle events associated with the customer itself...
>
> Having the lifecycle of key domain concepts **explicitly modeled** like this is pretty powerful. Not only do we have one place to deal with collisions of state (e.g., someone trying to update a customer that has already been removed), but we also have a place to attach behavior based on those state changes. (Emphasis added)

In SMaaS, each state machine instance is a separate microservice: it defines a bounded context, and owns all logic associated with behavior in this context. That context is mapped to a unique URL endpoint for the instance. When a state machine instance receives an event, it decides how to process that event according to logic defined in the SCXML definition. SMaaS uses SCXML as a modeling language to explicitly model the IoT domain concepts used to process events.

Each IoT device can be associated with a single state machine instance. In this way, each device is associated with a single microservice which encapsulates that device's state and models its reactive behavior.

Light Switch Microservice

Consider the example SCXML in Figure 2. An instance of this SCXML would be created on the SMaaS server for a particular WeMo device instance. This instance would be exposed via a URL endpoint, e.g. `http://home.local/api/v1/light-switch.scxml/living-room`, where "home.local" is the SMaaS server host, "light-switch.scxml" is the saved SCXML document from Figure 2, and "living-room" is the name of the state machine instance on the SMaaS server. A separate program would be run as a "connector" to listen to WeMo light switch events via the UPnP API and send SCXML events to the state machine instance via the SMaaS REST API.

This can be emulated using the cURL command-line utility. For example, to emulate an event in which the WeMo device is turned on, one would invoke curl using, e.g.

```
curl -XPOST
  -Hcontent-type:application/json
  -d {''name'':''device.turnOn''}
  http://home.local/api/v1/light-switch.scxml/living-room
```

This would send an HTTP POST request to URL `http://home.local/api/v1/light-switch.scxml/living-room` with JSON payload {''name'':''device.turnOn''}, which would dispatch an event named "device.turnOn" on the state machine instance.

Later, the device state could be queried using an HTTP GET to the the state machine instance's URL.

```
curl -XGET
  http://home.local/api/v1/light-switch.scxml/living-room
```

The response to this request would be the device "snapshot", serialized as JSON. The snapshot includes all the information required to reconstitute the state of the instance.

In this approach, the state machine instance "living-room" runs as a microservice, as it completely encapsulates its state and behaviour. It can only be controlled through a message-passing interface via its URL endpoint.

In SMaaS, the SCXML `<send>` tag is implemented in a similar fashion, such that an event encoded as JSON is sent via HTTP request to the URL of the target SCXML instance.

The full SMaaS REST API is defined using Swagger[1], a declarative markup for describing REST APIs. The SMaaS API Swagger definition can be found at:

`https://github.com/JacobeanRnD/SMaaS-swagger-spec`

Linux containers

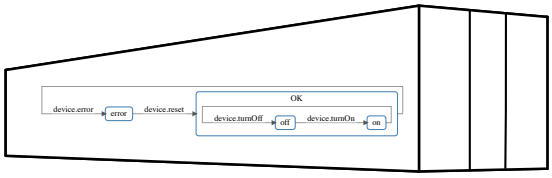

Figure 3. Light Switch Statechart running inside a Linux container

Linux containers (LXC) are an operating-system-level virtualization environment for running multiple isolated Linux systems (containers) on a single Linux control host.

The Docker open source software project provides an additional layer of abstraction on top of Linux containers. Docker is often used to implement applications based on a microservice architecture.

Linux containers can be used to run each state machine instance in isolation. The advantages of this approach are:

Speed

It is fast to create new containers. New state machine instances can typically be started in under a second. After the initial start time, the time required to process subsequent events can typically be measured in milliseconds.

Security

Each container runs in isolation, and operating system resources can be limited per container. Because SCXML contains arbitrary, user-specified JavaScript, a secure execution environment is required in order to prevent possible abuse by malicious users on a multi-tenant SMaaS cluster.

Ecosystem

There exist a number of technologies to run Docker containers across a cluster of computing resources to enable large-scale computing. These container technologies include Apache Mesos, CoreOS, and Joyent's Triton product. This reduces the effort required to create cloud computing infrastructure for large-scale IoT deployments.

JACOBEAN SMAAS IMPLEMENTATION

Jacobean Research and Development has released a number of free and open source software libraries:

- SCION, an implementation of SCXML in JavaScript.
- SCHVIZ, an automated visualization library that can render a running SCXML instance as a hierarchical graph diagram.
- SCXMLD, a Docker-based SMaaS server that can run SCION state machine instances in Docker containers across a cluster.
- SCXML-CLI, command-line tools for interacting with the SCXMLD cloud service.

Taken together, these technologies provide a feature-rich SMaaS implementation, and present unique advantages for implementing robust cloud computing services for the Internet of Things.

State Machine Snapshot

SCXMLD embeds SCION as an SCXML engine. SCION implements "state machine snapshotting", which is a technique for serializing the running state of an instance to JSON so that it can be saved to secondary storage. A snapshot is a JSON array containing: the set of basic states the state machine is in ("basic configuration"); the serialized datamodel; a boolean describing whether the state machine is in a final state; and the history states.

State machine snapshots have several advantages:

Economical

The snapshot can be saved to secondary storage, such as a relational database or document store. Secondary storage is cheaper than memory.

Crash Recovery

It is possible to restore an instance to a last known good state from a serialized snapshot, should the container crash. This is more robust than storing the instance state purely in memory.

Queryable

The state machine snapshot can be saved to a native JSON database such as MongoDB, CouchDB, ElasticSearch or the PostgreSQL JSON datatype, so that it can be queried efficiently. This facilitates "fleet management" queries. For example, one could efficiently query the database to retrieve light switch state machine instances that are in an "error" state.

Visualization

SCXMLD includes SCHVIZ, a tool for automatically visualizing SCXML as a hierarchical graph diagram. The

diagram is a "live" visualization, meaning that it is subscribed to state change events on the SCXMLD server, such that the diagram updates automatically to highlight the current state when the instance receives a new event. This provides excellent visibility into the state of a running system.

For example, consider the "living-room" light switch state machine instance from earlier. The light switch instance would start in state "off". In this case, the "off" state would be highlighted, as can be seen in Figure 4. After sending the event "device.turnOn", the state machine instance would transition to state "on", in which case the visualization would be automatically updated such that state "on" would be highlighted. This can be see in Figure 5.

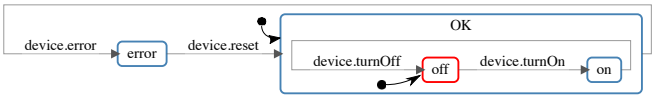

Figure 4. Running Instance Statechart Visualization

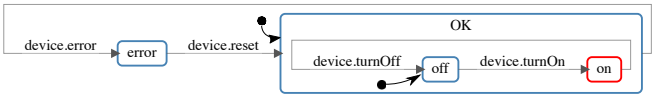

Figure 5. After sending event "device.turnOn"

SCHVIZ can be invoked using the SCXML-CLI command-line tools. For example, the command to visualize the light switch state machine instance would be:

```
scxml viz light-switch.scxml/living-room
```

This would open a window on the user's desktop that would render the visualization.

Automated visualization has a number of useful applications. First, it can be used in development to debug a system interactively, by sending events and observing the visual change in system state. Next, it can be used in operations for monitoring the running state of a system in production. Finally, it can be used to produce technical documentation, and communicate business requirements between non-technical domain experts and developers.

Event Log

SCXMLD stores a log of every event that is processed, and the resulting state machine instance state.

This can be used to provide a full audit of the state of the system over time, which is useful for debugging. For example, if a particular state machine instance enters an error state, the Event Log would allow the developer to step backwards through time to review the sequence of events that led to that error state.

OTHER APPLICATIONS

SMaaS has applications to domains outside of the Internet of Things. One example of this is automated algorithmic trading.

An example of a trading algorithm implemented as a state machine can be seen in Figure 6.

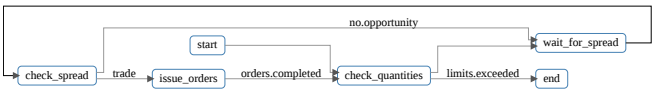

Figure 6. Trading Algorithm

Figure 6 is derived from an example listed in "Algorithmic Trading & DMA" by Barry Johnson [8]. Johnson states that:

> The Event Modeler provides a state-driven approach... The arrows show the progression from one state to another: So after starting and checking the quantities, the scenario effectively loops between the "wait for spread" and "check spread" states. Only when the spread is favourable will it shift to issue orders.

Johnson states that the server software executing the trading algorithm must be resilient, manageable and scalable. As SCXMLD includes support for snapshotting, event logging, and can be scaled across cloud infrastructure, it is well-suited to meet the requirements of an algorithmic trading server.

In general, SMaaS can be applied to the scalable, reliable execution of complex, event-driven systems. Other domains where SMaaS could be applied include telephony (for describing flows through interactive voice response menu), web applications (for describing navigation between web pages), and business process modeling.

CONCLUSION

This paper introduced SMaaS, an SCXML microservices platform built on Linux containers to control the Internet of Things. An example IoT application was described in which the SMaaS REST API was used to control an IoT light switch and monitor the device's state. Next, this paper introduced SCXMLD, an open source implementation of SMaaS with novel features including snapshots, logging, and automated visualization. Finally, other applications of SMaaS and SCXMLD were discussed, including an application to algorithmic trading.

REFERENCES

1. Swagger: The world's most popular framework for apis.

2. Jim Barnett, Rahul Akolkar, RJ Auburn, Michael Bodell, Daniel Burnett, Jerry Carter, Scott McGlashan, Torbjörn Lager, Mark Helbing, Rafah Hosn, T.V. Raman, Klaus Reifenrath, and No'am Rosenthal. State Chart XML (SCXML): State Machine Notation for Control Abstraction. *W3C Working Draft*, 2010.

3. O.A. Basir and W.B. Miners. Multi-participant, mixed-initiative voice interaction system, October 1 2009. US Patent App. 12/410,864.

4. Jacob Beard. Developing rich, web-based user interfaces with the statecharts interpretation and optimization engine. 2013.

5. Marcello Bonfe and Cesare Fantuzzi. Design and verification of industrial logic controllers with UML and statecharts. In *Control Applications, 2003. CCA 2003. Proceedings of 2003 IEEE Conference on*, volume 2, pages 1029–1034. IEEE, 2003.

6. D. Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231274, 1987.

7. Ian Horrocks. *Constructing the user interface with statecharts*. Addison-Wesley Longman Publishing Co., Inc., 1999.

8. Barry Johnson. *Algorithmic Trading & DMA: An introduction to direct access trading strategies*, volume 200. 4Myeloma Press, 2010.

9. Silvia Mur Blanch. *Statecharts modelling of a robots behavior*. Projecte fi de carrera, Escola Tcnica Superior DEnginyera, 2008.

10. Sam Newman. *Building Microservices*. "O'Reilly Media, Inc.", 2015.

11. Alvaro Siguenza, David D Pardo, Jose Luis Blanco, Jess Bernat, Mercedes Garijo, and Luis A Hernandez. Bridging the semantic sensor web and multimodal human-machine interaction using SCXML. *International Journal of Sensors Wireless Communications and Control*, 2(1):27–43, 2012.

12. Inc. The MathWorks. MATLAB and Simulink Help Toyota Design for the Future - MathWorks User Stories, 2011.

Energized State Charts with *PauWare*

Franck Barbier
Univ. of Pau, France
BP 1155
64013 Pau CEDEX
Franck.Barbier@FranckBarbier.com

Olivier Le Goaer
Univ. of Pau, France
BP 1155
64013 Pau CEDEX
olivier.legoaer@univ-pau.fr

Eric Cariou
Univ. of Pau, France
BP 1155
64013 Pau CEDEX
Eric.Cariou@univ-pau.fr

ABSTRACT
Persuading software engineers to systematically use on a large scale, a modeling language like SCXML greatly depends upon suited tools. At the very end, only financial concerns prevail: productivity increases due to modeling. Otherwise, modeling stops. This paper comments on *PauWare*, a Java technology that aims at ameliorating the daily practice of State Chart modeling. Beyond design, *PauWare* is based on models@runtime to continuously benefit from models when applications are in production.

Author Keywords
Model-Driven Development; State Charts; Executability.

ACM Classification Keywords
D. Software; D.2 SOFTWARE ENGINEERING; D.2.2 Design Tools and Techniques.

General Terms
Design.

INTRODUCTION
Since the takeoff and development of *Model-Driven Development* (MDD) in the spirit of the *Unified Modeling Language* (UML), modeling take-up remains fairly low. From experience in software industry, mental blocks persist. Developers rather prefer coding than modeling. Being graphical and/or textual, the situation is the same for all kinds of models; accordingly, modeling techniques are still often considered as supports for only producing software documentation.

The reason is "abstraction". Even though abstraction allows sound design principles like "separation of concerns", "incrementality" or "early design error detection", it is also "far from the processor". Latest software tuning is often incompatible with "idealistic worlds" in models. Over time, models and code diverge, leading developers to throw models overboard as soon as possible.

As a modeling language, SCXML spreading may stumble over these well-known "hurdles". The quality of surrounding well-integrated tools (editors, checkers, simulators, code generators…) plays then a crucial role for the success of a modeling language. For example, *Eclipse Modeling Framework* (EMF) [1] has made UML manageable in XML (declarative aspects) and Java (imperative statements as model transformations). In another style, Yakindu (statecharts.org) for State Charts allows model simulation and code generation. Both tools are actual proofs about moving models one step beyond: models benefit from being executable (or "interpretable"). Nonetheless, this idea is not new. In [2] or [3], the intention to offer an executable UML or a definitive virtual machine for the overall UML does not result in something tangible at this time.

This paper presents the *PauWare engine* Java library (PauWare.com) to design ordinary software applications from executable State Charts. From the origin, this library obeys to the execution semantics of UML (with safe homemade corrections), which is, in our opinion, very close to that of SCXML. Regarding theoretical concerns, *PauWare engine* is a research prototype mainly used for carrying out experiences in software adaptation [4]. Otherwise, the two key industrial realizations from *PauWare engine* are the implementation of a service mediator in the ReMiCS project (remics.eu) and a model debugger in the BLU AGE® MDD tool suite (bluage.com).

This paper discusses long experience and practice in State Chart modeling with concise consideration on associated tools, industrial usages, feelings and feedbacks as well on the high necessity of models with greater attractiveness and power of conviction.

REVISITING MDD
Over years, despite a certain know-how engraved in *PauWare engine*, it is still difficult to convince people to switch from prehistoric coding practices to relevant standards like SCXML. Open proven *Computer-Aided Software Engineering* (CASE) tools are important to guarantee progresses. In this context, code generation from SCXML models to *PauWare engine* API continues to raise a squaring-the-circle problem: the gaining of SCXML models is above all an often sizeable modeling effort, especially when requirements are numerous and complex, leading to labyrinthine State Charts. In other words, CASE tools cannot be substituted for human intelligence.

EICS'15 SCXML workshop, June 23, 2015, Duisburg, Germany.

State Chart execution with *PauWare engine* or direct interpretation with tools like Commons SCXML (commons.apache.org/scxml) supposes the prior nontrivial design of complete and ready-to-use SCXML models. Similar to code writing, modeling is error-prone with limited possibilities of testing intermediate designs.

To address these issues, the key idea is to give more latitude to software engineers in seamlessly navigating between models and code. Namely, "hiding" modeling activities can be a sound design principle. Concretely, once *PauWare engine* API under control, software engineers can express State Charts in Java with a very reduced set of classes/interfaces that easily and straightforwardly manage compound/leaf states, state machines and any kind of structuring: state nesting, state exclusiveness, state orthogonality, transitions, guards and actions. Other constructs of *PauWare engine* API are linked functions ("fires" and "run_to_completion" essentially).

In fact, there is no great distinction about dealing with SCXML or *PauWare engine*. Code generation may produce *PauWare engine* API code from SCXML source. SCXML models may also be derived from *PauWare engine* API code.

For example, here is a SCXML source sample extracted from the reference *Barbados Crisis Management System* case study (franckbarbier.com/PauWare/BCMS). States are in blue while events are in red:

```
<state id="Route_for_fire_trucks_development"
initial="Route_for_fire_trucks_to_be_proposed">

<final id="End_of_route_for_fire_trucks_development"/>

<state id="Route_for_fire_trucks_approved"/>

<state id="Route_for_fire_trucks_to_be_proposed">

<transition event="route_for_fire_trucks"
target="Route_for_fire_trucks_fixed"/>

</state>

<state id="Route_for_fire_trucks_fixed">

<transition event="FSC_agrees_about_fire_truck_route"
cond="!In('Route_for_police_vehicles_approved')"
target="End_of_route_for_fire_trucks_development"/>

<transition event="FSC_agrees_about_fire_truck_route" cond="!
In('Route_for_police_vehicles_approved')"
target="Route_for_fire_trucks_approved"/>

<transition event="FSC_disagrees_about_fire_truck_route"
target="Route_for_fire_trucks_to_be_proposed"/>

</state>

</state>
```

The corresponding *PauWare engine* code is as follows:

```
state_machine.fires(route_for_fire_trucks,
Route_for_fire_trucks_to_be_proposed, Route_for_fire_trucks_fixed);

state_machine.fires(FSC_disagrees_about_fire_truck_route,
Route_for_fire_trucks_fixed, Route_for_fire_trucks_to_be_proposed);

state_machine.fires(FSC_agrees_about_fire_truck_route,
Route_for_fire_trucks_fixed, End_of_route_for_fire_trucks_development,
this, "in_Route_for_police_vehicles_approved");

state_machine.fires(FSC_agrees_about_fire_truck_route,
Route_for_fire_trucks_fixed, Route_for_fire_trucks_approved, this,
"not_in_Route_for_police_vehicles_approved");
```

In this Java code, transitions are simply connected to source and target states. Events are later processed as follows:

```
public void route_for_fire_trucks() throws Statechart_exception {

state_machine.run_to_completion(route_for_fire_trucks);

} // Etc. Other events here…
```

As for SCXML conditions:

```
public boolean in_Route_for_police_vehicles_approved() throws
Statechart_exception {

return
state_machine.in_state(Route_for_police_vehicles_approved.name());

}
```

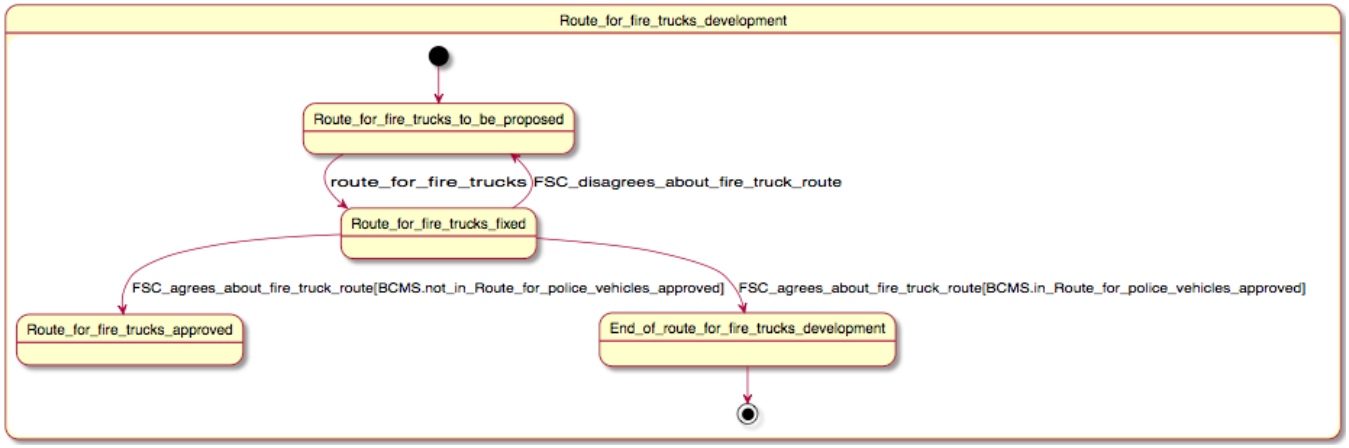

Figure 1. *PauWare view* look & feel (extract from *Barbados Crisis Management System*).

Testing through simulation at design time in particular relies on a third-party tool: *PauWare view*. *PauWare view* is an addon for *PauWare engine*. *PauWare view* dynamically generates one or more instances of State Charts in SVG

format by taking advantage of the PlantUML Java library (plantuml.sourceforge.net). *PauWare view* displays and simulates instances of State Charts in Web browsers in an asynchronous way (Figure 1). Any *PauWare engine* application communicates through Web sockets the discretized status of some or all of its running state machines. This logically results from the processing of event occurrences in run-to-completion cycles. Since applications have their own event processing frequency (for instance, a highly interactive application may be "bombed" by event occurrences), *PauWare view* acts as a buffer for displaying these occurrences in a human readable manner (refreshes are adjustable between 1 sec. and 5 sec.).

MODELS@RUNTIME

Even though *PauWare view* can be rightly viewed as a model testing tool at design time, its main purpose is run-time observation, even control in case of adaptation. The animation of State Charts by means of *PauWare view* in Web browsers is more than the simulation of models in the sense that these models are abstract software artifacts. Here, "abstract" precisely means that models mimic the grand characteristics of the final software, but all low-level details are not yet presented.

Instead, *PauWare view* is plugged in the application **in production** with, often, end-users being the source of event occurrences through GUIs. Running state machines may possibly be embedded in devices with system-oriented events (*e.g.*, battery events in an Android application [5]) or they can power *Enterprise JavaBeans* (EJBs) in large-scale SOA applications.

Keeping or not *PauWare view* at run time is a question of application administration in the spirit of the Java console. The latter aims at tracing, even controlling, operating applications. In all cases, cutting *PauWare view* off from *PauWare engine* is no effort. Performance issues for example may justify such a cutting even though *PauWare view* may run on other machines thanks to Web sockets.

Models@runtime [6] is the major source of inspiration for *PauWare* technology. No significant distinction is made between coding and modeling. Modeling is just disciplined coding to create higher intelligibility in the code by means of persisting models. Consequence is higher software quality, but nothing new under the sun: these are just software engineering entrails, *i.e.*, maintenability, reusability and reliability naturally increase.

WEAKNESSES

- With the exception of Java, there is no devoted mechanism in *PauWare engine* to write the bodies of actions launched in reaction to events or as entry/exit actions of states. The same applies for guards that are embodied by Boolean Java methods (see above). SCXML has a rich and relevant language-neutral approach with ECMAScript or, instead, by offering varied supports for different programming languages. Actions in *PauWare engine* stress data transformations in avoiding any control flow, which, in essence, is under the aegis of State Charts.
- PlantUML has drawing restrictions in the sense that it is not able to manage arrows (*i.e.*, transitions) that cross, from inside or outside, container states. *PauWare engine* does not have this embarrassing limitation, which confines *PauWare view* to specific forms of State Charts only. As an illustration, Figure 2 shows a model, which cannot be simulated at design time (and, mechanically, controlled at run time).

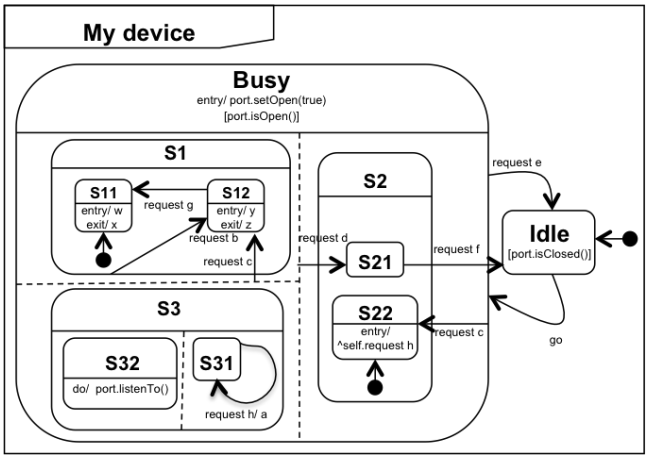

Figure 2. State Chart with numerous factorized transitions from/to superstates to/from substates.

The model in Figure 2 is simply and directly expressible in SCXML apart from proprietary UML constructs: "do/" UML notation for activities and state invariants between brackets (both are supported by *PauWare engine*). At run time, *PauWare engine* seamlessly executes the model in Figure 2, but, again, no behavior visualization is possible through *PauWare view*.

This problem can be bypassed with alternative PlantUML-compliant models having the same business semantics, but such models tend to accidentally become more complicated. Beyond, such an approach is dubious because tools serve modeling. It would simply be erroneous to envisage anything else.

- *PauWare* prompts software engineers to become model supporters with a kind of "extreme modeling" style. Indeed, "Write little matter-Compile-Test" is the rule in extreme programming: in short, tests drive development. However, such an approach is not unanimously known as a proven productive software development method when several stakeholders

are involved. A debatable question is the fact that MDD is recognized (or not?) as disruptive with respect to "ordinary" software development practices. Breaking requirements and specifications into modular pieces is normally favored by modeling. State Charts have intrinsic characteristics for being these pieces. This debate is outside the scope of this paper, but it is interesting to point out that State Chart expression is systematically preceded by an upstream significant modeling activity that is not readily aligned with *PauWare* design style.

STRENGTHS

- Distribution through Web sockets allows the remote run-time observation, even control (or self-control in case of self-adaptation [4]) of *PauWare engine* applications everywhere. Fruitful experiences relate to the Java Embedded technology. Running state machines are embeddable as a *System on Chip* (SoC) using, for instance, the Raspberry PI hardware. State Chart behavior visualization then becomes extremely informative for electronic/software engineers who have experience in only having "physical" perceptions of the SoC's behavior in a given real-word context. With reasonable effort, hardware-oriented events can be "mounted" on models animated in Web browsers.

- Without escape routes, crowded State Charts (due to challenging requirements) are both natural and difficult to read (to understand accordingly). *PauWare view* efficiently addresses combinatory issues. There is a kind of roundtrip engineering between code and (visualized) State Charts that are two distinguished viewpoints of the same thing. Typically, the suppression of useless model complexity often leads to code compaction/rationalization. Practice shows that the divergence between code and models in "traditional" MDD does not occur here.

- As already mentioned, models@runtime constitute an underlying appropriate support to keep control on running applications. For example, "runtime mutation" is recognized useful in [7] for debugging State Charts. Usually, code arises from specifications. Here, State Charts may derive from Java code and vice-versa. There is no effective upstream/downstream dependency between the two. *PauWare view* for example is able to take snapshots of (at rest or active) State Charts for software documentation production: a kind of "upside down software engineering".

CONCLUSION

In our opinion, compared to UML, SCXML succeeded in only keeping the true substance of the original Harel's Statecharts. In this paper, we defend the idea that a bi-layer approach is wrong. We mean: the classical MDD cycle in which code comes into being from models and code is, **later on**, enhanced with implementation details (*i.e.*, platform-specific information) that, in essence, do not belong to models because of their abstract nature, is a strong factor of MDD weakening and consequential rejection. A renewed MDD is possible if and only if models and code share a better articulation as offered by *PauWare*.

In this scope, the evolution of Commons SCXML is quite sound with the principle of an "expression language engine" in charge of parsing and evaluating imperative statements (typically, action bodies between the <*script*> and </*script*> tags). The possibility of using Groovy for instance as this expression language allows the controlled mixing of code and models as done in EMF, Commons SCXML, *PauWare* and, probably, forthcoming modeling environments.

ACKNOWLEDGMENTS

PauWare has been partly funded by the European Commission. All authors gratefully acknowledge the grant from the European Commission through the ReMiCS project (remics.eu), contract number 257793, within the 7th Framework Program.

REFERENCES

1. Steinberg, D., Budinsky, F., Paternostro M. and Merks, E. *EMF - Eclipse Modeling Framework, Second Edition*. Addison-Wesley, 2008.
2. Mellor, S. and Balcer, S. *Executable UML – A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.
3. Riehle, D., Fraleigh, S., Bucka-Lassen, D. and Omorogbe, N. The Architecture of a UML Virtual Machine. *Proc. 2001 Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM Press (2001), 327-341.
4. Barbier, F., Cariou, E., Le Goaer, O. and Pierre, S. Software adaptation: classification and case study with State Chart XML. *IEEE Software*, in press (2015).
5. Le Goaer, O., Barbier, F., Cariou, E. and Pierre, S. Android Executable Modeling: Beyond Android Programming. *Proc. 2014 International Workshop on Mobile Applications* (2014).
6. Blair, G., Bencomo, N. and France, R. Models@run.time. *IEEE Computer 42*, 10 (2009).
7. Junger, D. Transforming a State Chart at Runtime. *Proc. Engineering Interactive Systems with SCXML Workshop* (2014).

Extending SCXML by a Feature for Creating Dynamic State Instances

Peter Forbrig, Anke Dittmar, Mathias Kühn
University of Rostock
Albert-Einstein-Str. 22
D-18051 Rostock, Germany
{peter.forbrig | anke.dittmar | mathias.kuehn}
@uni-rostock.de

ABSTRACT
Statecharts have been demonstrated as an appropriate way for specifying the behavior of technical systems. In recent time they have been applied for specifying the behavior of navigation models of interactive systems. However, there is certain behavior of interactive systems that is difficult to specify with state charts. The creation of states that are concurrent to existing instances is such an example.

The development of SCXML might be the chance to introduce such a feature for navigation models based on statecharts of GUIs. The paper discusses an approach for extending statecharts and SCXML in this direction. It allows specifying the dynamic behavior of several instances of a certain window. Additionally, a specific characterization of states as modal is suggested. For both extensions examples are discussed that are intended to convince the reader that the extensions to SCXML are useful.

Author Keywords
Statechart, state machine, navigation model, graphical user interface specification, SCXML, dynamic creation of parallel hierarchical states, modal states.

ACM Classification Keywords
H.5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

INTRODUCTION
Statecharts and more specifically SCXML have been using quite successful for specifying the behavior of interactive systems. Some papers focus on web-based dialog systems (e.g. [8], [9]). Other papers discuss a more general approach (e.g. [5], [7]).

However, to the best of our knowledge there was no attempt to specify the behavior of several instances of the same window, which is necessary in a lot of applications.

We would like to refer to the domain of mailing systems. Users want sometimes to switch from one mail that is written at the moment to another one that is written at the same time. The same is true for reading mails. It is very common that users open several mails at the same time in different windows. Figure 1 provides the visual support for

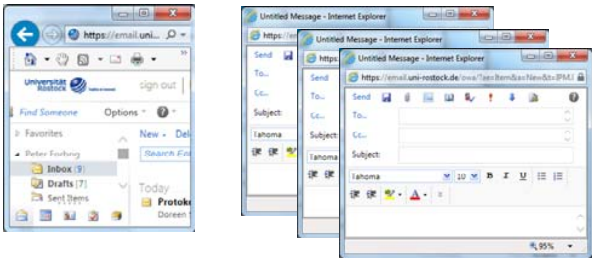

Figure 1. The Email system with two instances of the same window

Statecharts are able to specify such kind of behavior only for a fixed number of instances by a complex state with parallel sub-states. We will provide a proposal for an extension of statecharts and SCXML that allows specifying the described behavior in a good readable way.

The rest of the paper is structured in such a way that we will introduce or extension by example. After that we will discuss related work and finally there will be some conclusions.

PROPOSED EXTENSION TO STATECHARTS AND SCXML
The following section introduces an extension of statecharts and SCXML that allows specifying the creation of different instances of a complex state in a simple way. This feature is necessary for specifying dialog models for systems that are convenient and usable for users.

It is our idea to extend complex states by a feature that allows the creation of new complex sub-states that run in

parallel. For the creation process a new kind of transition is introduced that connects a source state with a destination state in such a way that a copy (new instance) of the complex state is activated in parallel. Additionally, control is given back to the source state. Initially there might be a complex state with no parallel active sub-state. Figure 2 gives an impression of how this might look like for our email example.

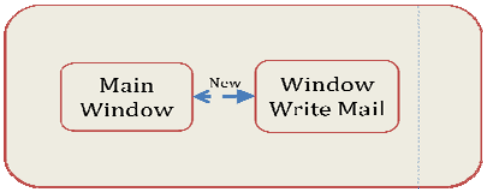

Figure 2. Statechart for creating instances of "Window Write Mail"

Figure 2 presents a possible notation for creating instances of a complex state and activating them. The dotted line might not be necessary. However, it characterizes the complex state as ready for accepting further parallel running sub-states.

The transition in Figure 2 has the two different arrowheads to express the semantics of creating a new instance of the destination state but still activating the source state after this action.

Assuming that "Main Window" is active and "New" activates the transition the situation of Figure 3 would be the result.

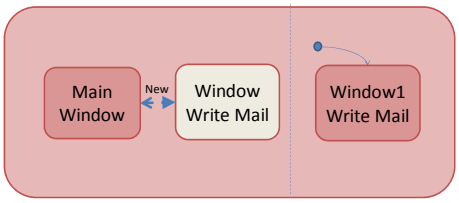

Figure 3. "Animated" statechart after creating a new instance of "Window Write Mail"

After creating s new instance for "Window Write Mail" the new window and the main window are active. Figure 3 tries to present as snapshot of a visualization of the runtime situation of the statechart specification. State "Window Write Mail" will never become active. It is a kind of pseudo state.

In the example of Figure 2 for states complex states are assumed even that only simple states were drawn. This was done for readability reasons only. However, one can imagine that these states contain some sub-states.

Additionally, it is assumed that a parallel state instance disappears after it is ended.

In this way the situation of Figure 3 can be followed by the situation of Figure 4 and after closing window2 and window3 again the situation of Figure 3 is reached. This assumes of course that in "Window Write Mail" the final state is reached with closing.

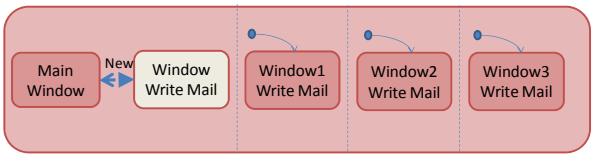

Figure 4. "Animated" statechart after creating three new instance of "Window Write Mail"

The consequences of the suggested extension of statecharts influence the interpreter/compiler of SCXML more than the syntax of the language. For the language it might be enough to introduce a new attribute for transitions. Let us call this attribute new_instance.

For Figure 2 the SCXML representation of the transition could look like.

```
<transition>
         event  = "New"
         target = "Window Write Mail"
         new_instance
</transition>
```

As already mentioned the extension to SCXML is simple but changes in supporting tools are more complicated.

Another suggestion of extending statecharts and SCXML is a characterization of states as modal. This means that only transitions of this state can be executed while it is active. In case the state "Window Write Mail" was modal there would be no chance to create a second instance of this state because state "Main Window" (and all other states) would be blocked.

Sometimes it makes sense to specify a dialog as modal to force the user to finalize the related process. Sometimes registration processes are specified as modal.

Modal dialogs are a common and quite old concept in HCI [1]. It would be good to have support in SCXML for such kind of modeling.

RELATED WORK

We have been working on model-based design of interactive systems for several years. Our tool support has been based on task models of the users. For the specification of the navigation dialog the concept of dialogue graphs was introduced.

Dialog graphs consist of a set of nodes, which are called views and a set of transitions. There are 5 types of views: single, multi, modal, complex, and end views. A single view is an abstraction of a single sub-dialog of the user interface that has to be described. A multi view serves to specify a set of instances of a sub-dialog. A modal view specifies a sub-dialog, which has to be finished in order to continue other sub-dialogs of the system. Complex views allow hierarchical descriptions of a user interface model. Nodes can be specified in this way by graphs. End views characterize the end of a dialog. Figure 5 presents the graphical notation for all views of a dialog graph.

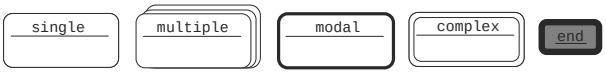

Figure 5. Types of views in dialog graphs

Single and complex views can be compared to complex states in state charts. The end view is similar to the end state. Multiple views were still missing in state charts but can be realized as suggested in this paper.

Modal states do not exist in statecharts yet. However, it might be a good idea to have this feature in SCXML as well. Again a simple new attribute for states like modal would be sufficient for the language. However, tools have to ensure that no transitions can be activated in other states until the modal state is deactivated.

A transition in dialog graphs is a directed relation between an element of a view and a view. Transitions reflect navigational aspects of user interfaces. It is distinguished between sequential and concurrent transitions. A sequential transition from view v1 to view v2 closes the sub-dialog described by v1 and activates the sub-dialog, which corresponds to v2. In contrast, v1 remains open while v2 is activated if v1 and v2 are connected by a concurrent transition.

Figure 6 presents the graphical notation for the different types of transitions and Figure 7 provides an example of a dialog graph with elements of related tasks.

Figure 6. Types of transitions

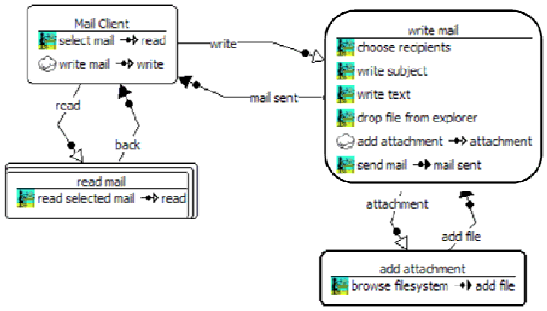

Figure 7. Dialog graph specification

The main window provides two options. One can read or write an email. In case of writing a mail the corresponding view opens and is active in parallel to the main window. Certain tasks can be performed within the new view. In case one activates the task add attachment a new view becomes visible. It disappears after a file was added. The view write m ail disappears after the mail was sent. With this specification only one mail can be written at the same time. However, several mails can be read simultaneously.

Our experience in specifying such kind of models inspired us to extend statecharts with features that seem to be important for the specification of the behavior of user interfaces.

Roxendahl [8] studies in detail the management of Web-based dialogue systems using SCXML. He provides a lot of examples but does not mention several instances of one window. His examples do not contain such a feature.

First Web-based mailing system did not have this feature as well. One can live without this feature but usability increases a lot if it is implemented.

The approach of Kistner et al. [5] follow a more general approach. They use SCXML for the general development of user interfaces. They especially focus on the glue between interface logic and presentation and driving the presentation from states. They argue for interpreting SCXML during runtime and report from successful applications with customers.

Winckler et al. [9] discuss the usage SWC and SCXML for the modeling of Web applications. They use the concept of a dynamic state. However, this is not related to the problem we mentioned. A dynamic state provides dynamic content in their examples. We are sure that their approach can be

combined with our extensions to SCXML and can benefit from them.

Dynamic changes in statechart specifications are discussed by Junger [4]. This paper does not directly refer to our problems but provides a lot of ideas for dynamic changes of specifications and discusses the consequences. Our approach can be considered as a further manipulation of statechart specifications during runtime. Such a runtime manipulation can be considered as one way of implementing it. This might be a workaround to implement the problem solution as manipulation at runtime but do not change the existing interpreter.

SUMMARY

Based on our experiences with model-based development of interactive systems two possible extensions of statecharts and SCXML were identified. We tried to argue for these extensions by examples and by referring to the literature.

These suggested extensions are:

1. Introducing a specific transition in hierarchical states that result while executed in a new instance of an activated complex sub-state that runs in parallel. When the sub-state is in its end state it is eliminated.

2. States can be characterized as modal and in case they are activated only their transitions can be executed.

In this way dialogs can be specified by state charts in a way that is common in the domain of HCI. Dialog graph specifications as discussed in [9] and [10] could be replaced by SCXML specifications. This would allow new approaches for platform independent development of interactive systems.

For the extensions certain notations were provided. However, this notation is not important. If it can be replaced by some other representations this would be fine for us. If the discussions during the workshop provide an improved result this would be great.

REFERENCES

1. Bonneau, P.: Edit control memory management; making modal dialogs modeless. Windows/DOS Dev. Journal 4, 7 (July 1993), 77-86.
2. Brusk, J., Lager, B., Hjalmarsson, A., and Wik, P.: DEAL – Dialogue Management in SCXML for Believable Game Characters *In Proceedings of the 2007 conference on Future Play (Future Play '07).* ACM, New York, NY, USA, 137-144.
3. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. In *Science of Computer Programming 8*, 3 (1987), 231-274.
4. Junger, D.: Transforming a State Chart at Runtime, *EICS 2014 Workshop on Engineering Interactive Systems with SCXML,* http://scxmlworkshop.de/eics2014/.
5. Kistner, G., and Nuernberger, C.: Developing User Interfaces using SCXML Statecharts. In *Proceedings of the 1st EICS Workshop on Engineering Interactive Computer Systems with SCXML* (2014), 5-11.
6. Kronlid, F., and Lager, T.: Implementing the Information-State Update Approach to Dialogue Management in a Slightly Extended SCXML. In *Proceedings of the 11th Workshop on the Semantics and Pragmatics of Dialogue* (2007), 99-106.
7. Lager, T.: Statecharts and SCXML for Dialogue Management. In *Proceedings of the 16th International Conference on Text, Speech, and Dialogue* (2013), Springer Berlin Heidelberg, 35.
8. Roxendahl, J.: Managing Web Based Dialog Systems Using StateChart XML, *Thesis University of Gothenburg*, 2010.
9. Schlungbaum, E., and Elwert, T.: Automatic user interface generation from declarative models, In Proceedings CADUI 1996, June 5-9, Namur Belgium, p. 3-18.
10. Schlungbaum, E., and Elwert, T.: Dialogue graphs: a formal and visual specification technique for dialogue modelling, In Proceedings of the 1996 BCS-FACS conference on Formal Aspects of the Human Computer Interface (FAC-FA'96), C. R. Roast and J. I. Siddiqi (Eds.). British Computer Society, Swinton, UK, UK, 13-13.
11. Winckler, M., Charrere, C., and Barboni, E.: From SWC to SCXML: using a statecharts-based markup language to model navigation of Web applications. *EICS 2014 Workshop on Engineering Interactive Systems with SCXML,* http://scxmlworkshop.de/eics2014/.
12. State Chart XML: http://www.w3.org/TR/scxml/

Formal Verification of Selected Game-Logic Specifications

Stefan Radomski
TU Darmstadt
Telecooperation Group
radomski@tk.informatik.tu-darmstadt.de

Tim Neubacher
TU Darmstadt
neubacher@cs.tu-darmstadt.de

ABSTRACT

Despite production budgets approaching the gross national income of small countries, many AAA game titles are still distributed with irritating, even game-breaking bugs in their quest and dialog structures. This severely damages a publishers reputation among its customers and entails considerable costs in the form of patches and support. By modeling these structures as state-charts, approaches for formal verification become applicable. In this paper, we will model a selection of bugs from a popular AAA title and present an approach to formally validate some of their soundness properties.

Author Keywords

SCXML; Harel State-Chart; Formal Verification; Languages

ACM Classification Keywords

D.2.4. Software/Program Verification: Model checking

INTRODUCTION

When Fallout 3 was released in 2008 by Bethesda Game Studios, over 200 bugs of various severities were identified by the community before the first patch was released one month later [1]. Some of the bugs were so severe that players had to revert to earlier save-games in order to progress with no way of knowing which of the previous decisions caused the problem in the respective quest or dialog and losing all progress. This problem is not exclusive to any publisher or developer but can be experienced in a wide selection of game titles. Many of the bugs are related to a game's quest structure or dialog behavior, e.g. a dialog partner required to progress a quest is no longer available, a critical item was sold or misplaced or a chance encounter leading to an undefined state and unfinishable quests.

In this paper we will model a selection of two bugs found in quests and dialogs in Fallout 3 using state-charts via SCXML and employ the formalisms of model-checking via Spin/PROMELA [7] to formally proof certain soundness properties. We will rely on a construction we described earlier [4] to transform a huge subset of SCXML onto PROMELA[2] as an input language for the Spin model-checker.

[1] `http://fallout.wikia.com/wiki/Fallout_3_bugs?oldid=52621` (accessed April 22nd, 2015)

EICS'15 Workshop, Engineering Interactive Systems with SCXML, June 23, 2015, Duisburg, Germany

RELATED WORK

Various approaches for enhancing the video game development process in general and the modeling of dialogs and quests in particular have been introduced over the last years.

Related General Approaches

In [8] the authors present a taxonomy of possible failures in a video game in order to help developers to identify various bugs during the development. The resulting taxonomy, focusses on implementation failures, omitting incorrect game specifications with regard to quests and dialogs. It can be used by human testers to classify the feedback to the developers.

Approaches using design patterns during the game development process have been introduced e.g. by [2] and [1]. While those papers mainly aim for reducing the maintenance effort and improving the communication between the different stake-holders during a game development process, the use of design patterns may also prevent the emergence of failures and increase the acceptance within the players community.

Also, mainly aiming for the improvement of collaboration tasks between the different participants during a video game development process, Moreno-Ger et al. have introduced a documental approach in [10]. They suggest a domain-specific markup language <e-Game> based on XML to create the video game storyboard. The executable video game is automatically produced by processing the resulting storyboard. As stated above, this structured development process may also lead to less bugs in to-be-released video games.

Brusk et al. have introduced the idea of a dialogue manager in State Chart XML (SCXML) in order to increase the believability of non-player characters (NPCs) in video games by improving their ability to communicate in natural language. Based i.a. on the idea of game patterns as stated in [2], the authors have chosen trading as a good example for the use of natural language in video games. Therefore they developed a trading model in SCXML allowing multimodal conversations as in human to human dialogs.

[2] `http://spinroot.com/spin/Man/promela.html` (accessed April 22nd, 2015)

Other Formal Approaches

A runtime-based approach for verifying game properties during the implementation process of a video game has been introduced in [12]. Varvaressos et al. create XML events aligned with the "game loop" via a XML template engine. The result is validated against constraints formulated in LTL-FO$^+$, a first order extension to Linear Temporal Logic (LTL) introduced in [5].

In [9] the authors introduced a model-checking approach for adventure video games based on a domain-specific language <e-Adventure>, an XML dialect. They developed a *Verification Model Generator* transferring the specification given in an <e-Adventure> document for "point & click" adventures to a verification model used as an input for the NuSMV [3] model-checker. This allows for the verification of such game specifications via Computation Tree Logic (CTL) expressions.

Shafiei and van Breugel show in [11] that bugs such as uncaught exceptions can be found via the use of Java PathFinder (JPF) [6], an explicit state software model-checker for Java^{TM} byte code. The basic possibilities of JPF can be extended by the use of several extension. It is e.g. possible to additionally verify properties defined in LTL. The authors also addressed the state space explosion problem and problems resulting through native calls and how to handle them.

Despite missing insights into the detailed development process of video games by the authors of this paper, it seems like the actual modeling of game-logic, especially with regards to dialogs and quests, is still a predominantly manual process with little or no established conceptualizations, much less formally verifiable. This is evidenced by the plethora of bugs and glitches found even in the most ambitious game titles.

FORMAL VERIFICATION OF STATE-CHARTS

In earlier work, we already described an approach to flatten SCXML state-charts into semantically equivalent state-machines notated in slightly extended SCXML[4]. This transformation is agnostic of the employed datamodel and results in an SCXML document wherein only a single state is ever active. Together with a subsequent transformation onto PROMELA programs, this allows for the application of the Spin model-checker (see figure 1).

We will outline the approach here again, as the original paper was missing the evaluation described below. For a more detailed description of the actual transformation process, we refer to our earlier paper[4].

The generality of the transformation onto SCXML state-machines can be evaluated by considering the subset of the 232 tests from the SCXML implementation and report plan (IRP)[3] that still pass after the transformation. And, indeed, they all pass if we introduce three slight adaptations to an SCXML interpreter:

[3] `http://www.w3.org/Voice/2013/scxml-irp/`

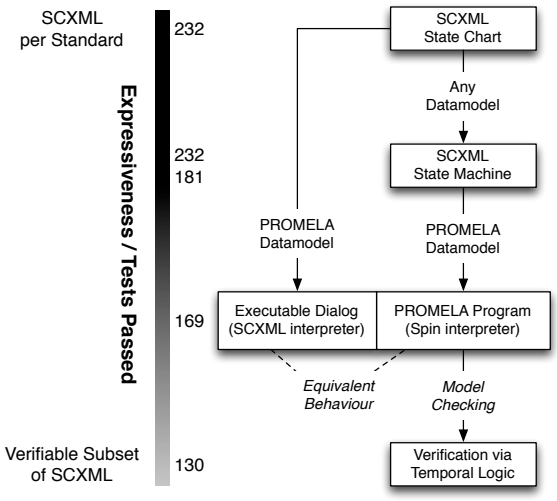

Figure 1. SCXML with the PROMELA datamodel allows (i) actual interpretation at runtime and (ii) verification via linear temporal logic.

1. Accounting for flattened state-names in SCXML's `In` predicate.
2. Making components instantiated by `<invoke>` persistent and introducing `<uninvoke>`.
3. Transforming `<donedata>` to `<raise>` and allowing for embedded content as with `<send>`.

With these changes to SCXML, a completely automated and semantically equivalent transformation from SCXML state-charts onto SCXML state-machines is possible regardless of the employed datamodel.

In the same paper we also introduced the PROMELA datamodel, allowing SCXML document authors to employ the syntax and semantics of the PROMELA language as an embedded scripting language. Here, the expressiveness can be evaluated by instantiating the 181 datamodel-agnostic IRP tests for the new datamodel and count the tests passed (51 of the IRP tests are specific to the `xpath` and `ecmascript` datamodel already specified in the SCXML standard). Our PROMELA datamodel will currently pass 169 tests with the reasons for failing the remaining 12 tests given in table 1.

Cause	#
String operations required	8
Declared vs. defined	1
Shallow copies in foreach	1
XML DOM node in variable	1

Table 1. Reasons for failing tests with the PROMELA datamodel.

Those two techniques allow us to create PROMELA programs as input files for the SPIN model-checker for just about any SCXML document employing the PROMELA datamodel. However, whereas flattened SCXML documents with the PROMELA datamodel are interpreted by an (slightly adapted) SCXML interpreter, PROMELA programs will be interpreted by the SPIN interpreter and we have to make sure that the respective semantics are preserved.

Cause	#
Relies on errors raised by platform	27
String operations required	7
Assumes open HTTP socket	4
Inexpressible event structure	3
Assigning to system variables	3
Identity for compounds	2
Late data binding	1
Dynamic URL for nested machine	1
Syntax error evaluates to `false`	1
Shallow copies in foreach	1
XML DOM node in variable	1

Table 2. Reasons for failing formal verification for PROMELA programs transformed from SCXML tests with PROMELA datamodel.

This can, again, be evaluated via the IRP tests by taking the 169 tests expressible and passed with the PROMELA datamodel, transform them into actual PROMELA programs and proof with an LTL expression that they will all eventually enter the `pass` state:

```
ltl { eventually (s == PASS)}
```

This is still possible for 130 of the 181 datamodel agnostic IRP tests with the reasons for failing the rest given in table 2.

The largest class of unverifiable tests are due to SCXML error semantics at runtime. It is plainly unpractical to raise these errors and the benefit is minimal as most of these situations would never occur if an SCXML author would have been more careful while writing the SCXML document. Other tests are inexpressible due to missing language support in PROMELA, e.g. there is no notion of strings and we emulate some of their semantics by enumerating string literals as integers to support the identity relation. However, more elaborate relations such as `contains` or `startsWith` are very difficult to model using only the integer arithmetics available in PROMELA.

Nevertheless, if an author is somewhat careful with regard to SCXML features employed (compare table 2) and choses the PROMELA datamodel, we can expose a respective document to the formalisms of model-checking via Linear Temporal Logic (LTL) expressions available in Spin/PROMELA.

Expressiveness of Linear Temporal Logic

To get an idea about the expressiveness of LTL to verify temporal properties / constraints of a state-chart, and to avoid a formal introduction of the syntax and semantics of LTL, it is helpful to have a look at some of the patterns and example distributed with Spin[4]:

Let `P`, `Q` and `R` be any property of a SCXML state-chart at runtime (i.e. the assignment of a variable, the current state or whether an entity is currently invoked), the following are valid LTL expressions:

- P is false between Q and R:
  `always ((Q and !R and eventually R) -> (!P U R))`
- P occurs at most twice:
  `(!P W (P W (!P W (P W always !P))))`
- P becomes true after Q until R:
  `always (Q and !R -> (!R U (P and !R)))`

[4]from `Examples/LTL/patterns.pml` in the Spin distribution

SELECTED GAME-LOGIC BUGS IN SCXML

The general approach of using state-charts or at least state-transition systems to model game logic is well established, e.g. in the form of the Unreal Engine's "Blueprints for Visual Scripting" [5] or the "Flow Graph Editor" of CryEngine[6].

Despite the simplification for the design of game-logic via the use of such systems during the development of video games, the complexity of modern video games and the sheer amount of in-game interactions still provide ample opportunities for unforeseen failures, even quite basic ones. In order to motivate how to avoid some of those bugs, we approximated two examples, found in the first release of Fallout 3, in SCXML. The datamodel employed in both cases is PROMELA to provide access to the declarations, expressions and a subset of statements of the PROMELA language as employed by the Spin model-checker.

Example 1: The Dead But Unforgotten Companion

In Fallout 3 the player can recruit NPC companions. While some of the companions will accompany the player only for the duration of a quest, some may follow the players character permanently until dismissed, helping the player e.g. in combat.
One of the companions available is Charon, a ghoul bodyguard, who can be signed by the player through either absolving a quest or paying for a contract. If Charon is under contract with the player's character and dies, the player cannot dismiss him nor drop the contract, leaving the player permanently unable to acquire another companion. Clearly, this game behavior was not intended by the developers.

To approximate the bug, something similar to the following SCXML state-chart might be employed. Keep in mind, that the purpose of this document is ultimately not only a basis for formal verification but also to be interpreted by an SCXML interpreter embedded in the game engine to actually define the game-logic.

```
1  <scxml datamodel="promela">
2    <datamodel>
3      <data>
4          mtype = { NONE, CHARON, FAWKES, .. }
5          typedef companion_t {
6            byte health;
7            int name;
8          };
9      </data>
10     <data>
11         companion_t party[1];
12     </data>
13     <data>
14       party[0].name = NONE;
```

[5]**https://docs.unrealengine.com/latest/INT/Engine/Blueprints/index.html** (accessed April 22nd, 2015)
[6]**http://docs.cryengine.com/display/SDKDOC2/Flow+Graph+Editor** (accessed April 22nd, 2015)

```
      party[0].health = 100;
    </data>
  </datamodel>
  <parallel id="root">
    <state id="party">
      <transition event="companion.new.charon"
                  cond="party[0] == NONE">
        <script>
          party[0].name = CHARON;
          party[0].health = 100;
        </script>
      </transition>
      <transition event="companion.dismiss"
                  cond="party[0] != NONE">
        <script>party[0].name = NONE;</script>
      </transition>
      <transition cond="party[0].health == 0">
        <script>party[0].name = NONE;</script>
      </transition>
    </state>
    <state id="health">
      <!-- automatically drain health to trigger bug,
           this would not exist in an actual game -->
      <transition target="health"
                  cond="party[0].health > 0">
        party[0].health = party[0].health - 1
      </transition>
    </state>
  </parallel>
<scxml>
```

Example 2: Endless Discussion

In many video games, experience points (XP) are a measurement of character progress and can be achieved in different ways, depending on the specific game logic, e.g. through absolving quests or using specific abilities of the players character.

In Fallout 3 one of the skills which can occasionally be used in conversations with NPCs is *Speech*, the proficiency at persuading other characters in the game. Such a dialog option can be seen as a challenge and a player, depending on the current skill level of the players character, may succeed or fail the challenge. By using this ability successfully during an in-game conversation, the players character automatically gains XP.

For example talking to the NPC *Bittercup* in *Big Town* enables an additional dialogue option, based on the *Speech* skill, with *Pappy*, another NPC. In the first release of Fallout 3, this dialogue option was always available, allowing the player to gather an infinite amount of XP by repeatedly succeeding the *Speech* challenge. As this has been fixed with one of the first patches for the game, this behavior clearly also was not intended. In most cases such XP increasing options should be available only until the player succeeded once.

This can easily be expressed in SCXML via the following snippet.

```
<scxml datamodel="promela">
  <parallel id="main">
    <state id="pappy.dialog">
      <state id="pappy.dialog.initial">
        <transition event="pappy.speech.pass"
                    target="pappy.speech.pass" />
      </state>
      <state id="pappy.dialog.speech.pass" />
    </state>

  </parallel>
</scxml>
```

The actual availability of the speech challenge would depend on the state of the dialog with *Bittercup*, but this will have to suffice as an example. Both examples are obviously very simplified, the main point is to provide a convincing argument that such game logic, indeed all game-logic, of certain games can potentially be expressed in SCXML with the PROMELA datamodel.

FORMAL VERIFICATION VIA LTL EXPRESSIONS

For the first example from above, we need to verify that a dead companion is removed from the party. We can do this via the following LTL expression:

```
always(
  (party[0].health == 0)
    -> eventually(party[0].name == NONE)
)
```

This expression will cause the Spin interpreter to verify that the first (and for Fallout 3 sole) companion in the party is removed if his health drops to zero. It can be read as: “whenever the companions health is 0, all subsequent computation path will eventually pass a state where the party is empty”.

In the second example, we need to proof that a given dialog option can only be selected once as soon as the associated challenge is passed. This can be achieved via the following LTL expression:

```
always(
  (_x.states[PAPPY_DIALOG_SPEECH_PASS])
    -> always(! _x.states[PAPPY_DIALOG_INITIAL])
)
```

In our transformation, the set of active states from the original state-machine for a given configuration remain available in the PROMELA structure `_x.states[STATE]`, allowing LTL expressions regarding the original state names and their sequences. The expression above would read as “whenever `pappy.dialog.speech.pass` was observed in the state-charts configuration, it will never again assume the initial state of the dialog with Pappy”.

Both expressions can be verified by using Spin with the PROMELA representation of the transformed state-charts listed above and will indeed hold true.

CONCLUSION

In this paper we outlined an approach to formally verify properties of game-logic for computer games to detect and avoid game-breaking bugs and glitches. By modeling these structures as state-charts with a formally provable datamodel in SCXML and transforming them via state-machines onto input files for a model-checker, we enabled the verification of expression given in Linear Temporal Logic (LTL).

A major drawback of the approach is the fact that most properties to be proven formally only become obvious in hindsight, after being detected by the player community in the initial release. However, much of the game-logic within a given genre is shared among titles, allowing to avoid at least the most common pitfalls for similar structures in future games. Given the glaring obviousness of some of the bugs, e.g. in Fallout 3, this would still be a considerable benefit.

Another point of critique might be the rather simplified nature of modeling these structures in the SCXML examples in this paper. However, the approach was successfully employed for state-charts wherein the intermediate state-machine representation had well over 10.000 states. If one were able to compartmentalize game-logic into distinct, mutually side-effect free sub-logic, this approach could scale very well for the game-logic of complete game titles.

One important point for future work could be the projection onto other model-checkers providing other classes of temporal logic (e.g. CTL with the NuSMV model-checker).

Given the potential benefits of formally proofing game-logic via temporal logic and the pervasiveness of related bugs even in titles with huge production budgets, a closer investigation of the approach seems very promising.

REFERENCES

1. Ampatzoglou, A., and Chatzigeorgiou, A. Evaluation of object-oriented design patterns in game development. *Inf. Softw. Technol. 49*, 5 (May 2007), 445–454.
2. Bjrk, S., Lundgren, S., and Holopainen, J. Game design patterns. In *in Level Up: Digital Games Research Conference 2003* (2003), 4–6.
3. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, Springer (2002), 359–364.
4. for blind review, R. Removed for blind review. In *Removed for blind review* (June 2014).
5. Hallé, S., and Villemaire, R. Runtime enforcement of web service message contracts with data. *IEEE T. Services Computing 5*, 2 (2012), 192–206.
6. Havelund, K., and Pressburger, T. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer 2*, 4 (2000), 366–381.
7. Holzmann, G. J. The model checker spin. *IEEE Transactions on software engineering 23*, 5 (1997), 279–295.
8. Lewis, C., Whitehead, J., and Wardrip-Fruin, N. What went wrong: A taxonomy of video game bugs. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, FDG '10, ACM (New York, NY, USA, 2010), 108–115.
9. Moreno-Ger, P., Fuentes-Fernández, R., Sierra-Rodríguez, J. L., and Fernández-Manjón, B. Model-checking for adventure videogames. *Information & Software Technology 51*, 3 (2009), 564–580.
10. Moreno-Ger, P., Sierra, J. L., Martínez-Ortiz, I., and Fernández-Manjón, B. A documental approach to adventure game development. *Sci. Comput. Program. 67*, 1 (June 2007), 3–31.
11. Shafiei, N., and van Breugel, F. Towards model checking of computer games with java pathfinder. In *Proceedings of the 3rd International Workshop on Games and Software Engineering: Engineering Computer Games to Enable Positive, Progressive Change*, GAS '13, IEEE Press (Piscataway, NJ, USA, 2013), 15–21.
12. Varvaressos, S., Lavoie, K., Massé, A. B., Gaboury, S., and Hallé, S. Automated bug finding in video games: A case study for runtime monitoring. In *IEEE Seventh International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA* (2014), 143–152.