On the Role of Nondeterminism and Refinement in Model-Driven Top-Down Development of Software Systems

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Heiko Schmidt

Kiel
2009

1. Gutachter	Prof. Dr. Willem-Paul de Roever
2. Gutachter	Prof. Dr. Gerald Lüttgen
Datum der mündlichen Prüfung	20.11.2009

Abstract

Large-scale software systems need to be accurately planned and designed. This includes the determination of requirements, the definition of specifications, and the development of models conforming to specifications. These models are expressed in modeling languages like process algebras, the Unified Modeling Language (UML) [Obj07], or variants of state diagrams (e.g. UML state machines [Obj07] or Harel's statecharts [Har87]). Such modeling languages are usually underspecified, since they only express certain aspects of the system to be designed, leaving out implementation details. The process of refining such abstract descriptions in a stepwise fashion, until finally the concrete, executable implementation is reached, is called (model-driven) top-down development. Finding bugs as early as possible in this process often saves considerable development costs.

This thesis considers methods for proven-to-be-correct top-down development, with specification conformance being guaranteed at all levels of abstraction, either by applying model checking [CGP01] techniques or by employing pre-defined refinement patterns that are already proven to be sound. In order to apply formal proof methods, models on all levels of abstraction, e.g. presented as process algebra [BPS01] terms or state diagrams, need to be given a precise semantics in some semantic domain, usually based on (extensions of) labeled transition systems. We call semantic domains that support underspecification *refinement settings*. One of the contributions of this thesis is a new kind of comparison of a dozen such settings proposed in the literature with respect to their expressible sets of implementations. This comparison is done by providing transformations that not only establish the implementation-based expressiveness hierarchy of the most commonly used refinement settings, but can also be employed to convert models between the settings, thus enabling tool reuse.

Some kinds of abstract models require a setting as semantic domain that not only features *resolvable nondetermism* expressing underspecification, but also *persistent nondeterminism* that is not to be resolved in refinements, as characterized by bisimulation equivalence [Par81] on labeled transition systems. We show that such a setting is needed for process algebras if they specify concurrent systems, because concurrency may introduce resolvable nondeterminism which is resolved by the scheduler of the operating system, and the choice operator, which is common to process algebras, may correspond to persistent nondeterminism. This is the first work in the literature making this observation. A simple process algebra of this kind is given an operational semantics, using the refinement setting of μ-automata, as well as a sound and complete axiomatic semantics.

Sometimes state diagrams, such as UML state machines [Obj07] or Harel's statecharts [Har87], also require a refinement setting with both kinds of nondeterminism, because (i) they are underspecified and (ii) the underlying action language may contain operators exhibiting persistent nondeterministic behavior. This thesis is the first publication presenting a state diagram semantics with both kinds of nondeterminism. In this context, existing refinement settings like μ-automata [JW95] lead to unnecessarily complex semantic models. Therefore, we develop a new and in this context more succinct refinement setting, called ν-automata, and give a semantic mapping for a simple state diagram variant, as well as a general transformation that can be applied when extending existing semantics by persistent nondeterminism. Thus, we make state diagrams accessible to persistent nondeterminism.

Support for both kinds of nondeterminism, however, does not necessarily imply the practical feasibility of top-down development in state diagrams. In existing state diagram variants, expressing resolvable nondeterminism is only possible to a certain degree, because the notations for underspecification (i) often have no precise semantics, and (ii) are not expressive enough to reflect the requirements of the top-down development process, such as starting with interface definitions and subsequent parallel development of mostly independent modules. Therefore, we develop a new variant of state diagrams that allows more explicit and more expressive modeling of underspecification than existing variants. This variant is given a semantics in a newly developed semantic setting that distinguishes between input and output events. A set of refinement patterns is then provided that enables proven-to-be-correct stepwise refinement without the need to re-check correctness after each refinement step. Consequently, we deliver the formal foundations for the development of a state-diagram-based design tool that ensures correctness at all stages of the development process.

Zusammenfassung

Große Softwaresysteme bedürfen sorgfältiger Planung und Entwicklung, einschließlich einer Anforderungsanalyse, dem Aufstellen von Spezifikationen und der Entwicklung von Modellen, die die Spezifikation einhalten. Solche Modelle werden in Modellierungssprachen wie Prozessalgebren, der Unified Modeling Language (UML) [Obj07] oder Varianten von Zustandsdiagrammen (z.B. UML state machines [Obj07] oder Harel's statecharts [Har87]) ausgedrückt. Diese Modellierungssprachen sind üblicherweise unterspezifiziert, d.h. sie beschreiben nur bestimmte Aspekte des zu entwickelten Systems und lassen Implementationsdetails weg. Der Prozess, solche abstrakten Beschreibungen schrittweise zu verfeinern, bis schließlich die konkrete, ausführbare Implementation erreicht ist, wird (modellgetriebene) Top-Down-Entwicklung genannt. Es spart oft beachtliche Entwicklungskosten, wenn Programmierfehler so früh wie möglich in diesem Prozess gefunden werden.

Die vorliegende Arbeit betrachtet Methoden für per-Konstruktion-korrekte Top-Down-Entwicklung, für die Spezifikationstreue auf allen Abstraktionsstufen gewährleistet ist, entweder durch die Anwendung von Modelchecking-Techniken [CGP01] oder durch die Verwendung von vordefinierten Verfeinerungsmustern, deren Korrektheit bereits bewiesen ist. Um formale Methoden anwenden zu können, muss den Modellen auf allen Abstraktionsstufen, ausgedrückt etwa in Prozessalgebren [BPS01] oder Zustandsdiagrammen, eine präzise Semantik gegeben werden. Solche Semantiken werden üblicherweise mittels (Erweiterungen von) Transitionssystemen ausgedrückt. Wir nennen solche semantischen Formalismen, die Unterspezifikation unterstützen, *Verfeinerungsformalismen.* Einer der Beiträge dieser Arbeit ist eine neue Art von Vergleich von einem Dutzend solcher Formalismen, in Hinblick auf ihre ausdrückbaren Mengen von Implementationen. Dieser Vergleich erfolgt durch die Angabe von Transformationen, die nicht nur die implementationsbasierte Ausdrucksstärkenhierarchie der meistbenutzten Verfeinerungsformalismen begründen, sondern auch dafür verwendet werden können, Modelle zwischen Formalismen zu konvertieren und damit die Wiederverwendung von Werkzeugen zu ermöglichen.

Einige abstrakte Modelle benötigen einen semantischen Formalismus, der nicht nur *auflösbaren Nichtdeterminismus* für das Ausdrücken von Unterspezifikation, sondern auch *persistenten Nichtdeterminismus* enthält. Letzterer soll nicht in Verfeinerungen aufgelöst werden, wie es durch Bisimulationsäquivalenz [Par81] auf Transitionssystemen charakterisiert wird. Wir zeigen, dass ein solches Modell für Prozessalgebren im Kontext nebenläufiger Systeme benötigt wird, weil Nebenläufigkeit auflösbaren Nichtdeterminismus einführen kann, der vom Scheduler des Betriebssystems aufgelöst wird, und der Choice-Operator, welcher in Prozessalgebren üblich ist, persistentem Nichtdeterminismus entsprechen kann. Dieses ist die erste publizierte Arbeit, die diese Beobachtung macht. Wir geben für eine einfache Prozessalgebra eine operationelle Semantik mittels μ-Automaten [JW95], sowie eine korrekte und vollständige axiomatische Semantik an.

Auch Zustandsdiagramme wie UML state machines [Obj07] oder Harel's statecharts [Har87] benötigen manchmal semantische Formalismen mit beiden Arten von Nichtdeterminismus, weil Zustandsdiagramme (i) unterspezifiziert sind und (ii) die zugrundeliegende Aktionssprache Operatoren enthalten kann, die persistent-nichtdeterministisches Verhalten zeigen. Die vorliegende Arbeit ist die erste, die eine Zustandsdiagramm-Semantik mit beiden Arten von Nichtdeterminismus vorstellt. In diesem Kontext würden existierende semantische Modelle wie μ-Automaten zu unnötig komplexen semantischen Modellen führen. Daher entwickeln wir einen neuen, in diesem Kontext bündigeren Verfeinerungsformalismus, nämlich ν-Automaten, und geben eine semantische Abbildung für eine einfache Zustandsdiagrammvariante, sowie eine allgemeine Transformation an, die auf existierende Semantiken, die um persistenten Nichtdeterminismus erweitert werden sollen, angewendet werden kann. Wir machen also Zustandsdiagramme im Allgemeinen für persistenten Nichtdeterminismus zugänglich.

Die Unterstützung von beiden Arten von Nichtdeterminismus impliziert jedoch nicht notwendigerweise die praktische Umsetzbarkeit von Top-Down-Entwicklung in Zustandsdiagrammen. In existierenden Zustandsdiagrammvarianten ist das Ausdrücken von auflösbarem Nichtdeterminismus nur zu einem gewissen Grade möglich, weil die Notationen für Unterspezifikation (i) oft keine präzise Semantik haben, und (ii) nicht ausdrucksstark genug sind, um die Anforderungen des Top-down-Entwicklungsprozesses widerzuspiegeln, wie das Starten mit der Definition von Schnittstellen und nachfolgende parallele Entwicklung größtenteils unabhängiger Module. Daher entwickeln wir eine neue Zustandsdiagrammvariante, die expliziteres und ausdrucksstärkeres Modellieren von Unterspezifikation als existente Varianten unterstützt. Ihre Semantik wird in einem neu entwickelten semantischen Formalismus gegeben, der zwischen Eingabe- und Ausgabeereignissen unterscheidet. Eine Kollektion von gegebenen Verfeinerungsmustern erlaubt korrekt-bewiesene schrittweise Verfeinerung, ohne dass die Korrektheit nach jedem Verfeinerungsschritt erneut bewiesen werden muss. Wir liefern also die formale Basis für die Entwicklung eines zustandsdiagrammbasierten Entwicklungswerkzeugs, welches Korrektheit in allen Stadien des Entwicklungsprozesses sicherstellt.

Acknowledgments

I would like to express my gratitude to all the people who accompanied and supported me during the creation of this thesis. First of all, I thank Harald Fecher, who first seeded my interest in this field of research by supervising my master thesis and then taught me most of the skills needed to complete this dissertation. The last years of our cooperation were characterized by locational separation, since Harald left Kiel for London and Freiburg, and I finally went to Bamberg. Willem-Paul de Roever, the leader of the working group in which I had the opportunity to work, however knew how important it is for a PhD student to see his supervisor as often as possible and always had an eye on that, making me visit him as often as possible. I thank him for that, and all those other efforts that made my stay in Kiel a very comfortable one.

Of course, my time in Kiel would have been less enjoyable without all the colleagues with whom I shared offices, scientific ideas, and "daily wisdom". In particular, I would like to thank Jens Schönborn, Andreas Grüner, Immo Grabe, Meiko Jensen, Claus Traulsen, Jan Waller and Tim Fenten for the great time I had in Kiel. I realized how much I missed their presence, when our working group slowly ended existing, with many of these guys leaving in different directions.

Finally it was also time for me to leave, with my contract in Kiel ending in April 2009. It was Gerald Lüttgen, who thankfully offered me a position at the University of Bamberg, allowing for a smooth completion of my PhD thesis. As a consequence, I spent half a year in Franconia together with my colleagues Gerald Lüttgen, Tobias Mühlberg, Tarek Nashashbi, Alexander Ditter and Alexandra Homer who made this working environment so enjoyable.

Gerald Lüttgen was also one of the colleagues I had the opportunity to collaborate with over the years. During his time at the University of York, I also enjoyed fruitful discussions with Andy Galloway, Richard Paige and Lishan Harbird. Furthermore, I benefited from e-mail discussions with Michael Huth and David de Frutos-Escrig. I thank all of these people for their kind support.

Furthermore, I would like to express my gratitude to Kirsten Kriegel for her never ending historical and botanical knowledge, which brought a refreshing change to my every-day work, to my parents Ulrike and Wolfgang for their personal and financial support during my studies, and to my friends Nadine Hauptmann and Michael Graff for their numerous emotional and intellectual inspirations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides a general introduction into the research field of this thesis, i.e. the use of refinement and abstraction in system design, and the role of nondeterminism in this context. We name the publications that form the basis of this thesis and give an outline of the thesis' content.

1.1 Refinement and abstraction in software systems

Today's technological world is increasingly made up of complex software systems, embedded in every-day electronic devices like cell phones, mobile computers, or entertainment electronics, in means of transportation like cars, trains, or planes, and much more. Control software assures that cars do not slide in emergency situations, trains do not collide, and planes do not crash. All these systems have to interact with other systems, e.g. cell phones interact with mobile radio facilities, trains interact with traffic control units, etc. Not only do software systems become more and more ubiquitous, also their degree of interconnectedness increases. Many of these systems are required to function properly, i.e. according to a given correctness specification, because failure could result in loss of large amounts of money, or even loss of life, e.g. if the systems are safety-critical parts of cars, trains or planes.

Due to these trends, the need to design proven-to-be-correct software grows steadily. Software systems must be given a formal semantics so that correctness properties can be checked. Since hardly any current software system is completely self-contained, the semantics needs to be *open*, i.e. the behavior of a system depends on the system's environment. This thesis considers different languages and techniques for open semantics of systems, focusing on refinement and abstraction.

Refinement in top-down development

Large-scale software systems need to be accurately planned and designed. This includes the determination of requirements, the definition of specifications, and the development of models conforming to these specifications, expressed in modeling languages. Such modeling languages are usually underspecified: they only consider certain aspects of the system to be designed, leaving out implementation details. In the *model-driven* design approach, these models are taken as a basis for further development, which is usually performed by different teams of developers working on different modules of the system, refining them, until finally executable code is produced.

This process is called *top-down development* because the development process starts at an abstract *top* level and then continues more and more *down* to the details of the implementation [Wir71]. There are many tools, techniques and languages that support developers with this process. The *Unified Modeling Language (UML)* [Obj07] offers a variety of visual languages for describing requirements, static class layouts, dynamic behavior, and much more. Many tools make use of these languages and simplify the development process. Models designed using these tools abstract from implementation detail, thus specifying some constraints on the software, but leaving other design choices undecided. This is what we call an *abstract* model. Every abstract model has a set of valid *concrete implementations*, i.e. executable programs, namely all those programs that conform to it.

Although this is not current practice yet, the process of *refinement* should ideally correspond to a series of models, where each narrows the set of possible implementations from the previous one, finally reaching a single concrete implementation that is consistent with all models. Each refinement step could be selected from a set of proven-to-be-correct patterns that guarantee conformance with the previous model.

However, such a framework requires well-defined semantics on all levels of refinement, e.g. for different types of UML models and for executable program code, and this has so far not been achieved in the literature. In particular, the chosen semantic domain must be able to handle refinement, i.e. it must be possible to express that certain decisions have not yet been made, and there must be a formal definition of a refinement preorder on the semantic domain. One of the contributions of this thesis is a new kind of comparison of a dozen such *refinement settings* with respect to their expressiveness. Another contribution is the illustration of their use in the context of top-down development.

A refinement-sensitive semantics of this kind has the further advantage that properties of models can be checked at all levels of refinement. Thus, property checks can accompany the top-down development process, and violations can be detected as soon as possible and when they are still cheap to eliminate. In traditional approaches where verification or testing can only be applied at implementation level, violations may instead lead to large portions of the system having to be redesigned, possibly at huge expense.

Unfortunately, this ideal of top-down development by stepwise refinement is not yet common practice. One of the reasons relates to missing or ambiguous semantics. Even for concrete programming languages, their semantics is not always clear, and for abstract models, e.g. expressed in the UML, the situation is even worse. Efforts are taken, e.g. in the UML 2 semantics project [UML], but a uniform semantic model from requirements to implementation has not yet become widely accepted.

Abstraction in verification

In the previous section, we have described the process of refinement, working from an abstract model towards a concrete implementation. This section presents applications where it is useful to take the other direction, i.e. starting with a concrete program and making it more abstract.

For this, consider a given concrete program which is to be proven correct. The program is written in a language with a formal semantics, so theoretically we have all means needed to prove properties, e.g. via the technique of *model checking* [CGP01]. This technique works for sequential (closed) programs if they are not too large, but today's systems are not isolated closed units. Instead they may contain several threads of control, e.g. they may run on different processor cores or on different computers in a cluster, and they interact with and get influenced by their environment, e.g. by other systems or human users. Today's applications are open and concurrent, they run in parallel, and influence each other. This has a large impact on the number of reachable states of a program. This phenomenon is usually referred to as the *state explosion problem* [Val98]. For instance, a parallel program consisting of two sequential toy programs of five states each, can have up to $5^2 = 25$ states; five programs of this kind can already have $5^5 = 3.125$ states! And of course, a program of five states can hardly be anything more than a toy example; the resulting state spaces of *real* programs can be much larger, or even infinite.

The state explosion problem is typically solved using *abstraction*. The concrete program is abstracted (i.e. made more abstract), e.g. using the technique of *predicate abstraction* [GS97]. Here, all states satisfying a given predicate are united to form an abstract new state. This decreases the amount of states and can make large systems amenable to model checking. In particular, infinite state systems can sometimes be abstracted such that the state space of the abstraction is finite.

Of course, some details are lost when abstracting a system. It is possible that an abstract system satisfies neither a property nor its negation. Three-valued logics are commonly used to describe this case by a third truth value besides *true* and *false*, called *unknown*. In case a property is *unknown* with respect to a given abstract system, the system is too abstract to decide the validity of the property and should be refined, e.g. on the basis of a counter-example. This technique is called *counter-example guided abstraction-refinement (CEGAR)* [CGJ+03].

1.2 Process semantics

This section describes the formalism in which we specify (either concrete or abstract) systems or, to be more precise, the observable behavior of those systems, called *processes*. We consider systems that perform *actions*, i.e., steps of observable behavior. These actions are used for the interaction with the system's environment which may consist of other systems or human interactors. Practically, an action can correspond to a control light being turned on or off, to a button being pressed, to a (remote) method being invoked in this or another system, or to a message being sent to the considered system or issued there. *Uniform* process semantics does not distinguish between input actions (e.g. button pressed, method invoked in this system, message received) and output actions (e.g. light turned on, method invoked in another system, message sent), which allows for a very general view. However, in Chapter 6, which presents a practical application, we will differentiate between input and output actions.

Commonly, processes are described using so called *labeled transition systems*. We will simply refer to them as transition systems. A transition system is a directed graph having a set of designated initial nodes and labeled edges. The nodes are identified with the states of the described process. We typically use variables s, t, u, ... for states. Edges are labeled by a, b, c, ... and stand for transitions of the system: an edge from state s to state s' having label a, written $s \xrightarrow{a} s'$ for short, means that if a process is in state s and action a occurs, the process will evolve to state s'. Nondeterminism is allowed, i.e., there may be states s, s_1', s_2' and a

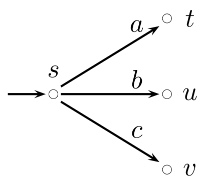


Figure 1.1: A transition system.

label a such that $s \xrightarrow{a} s'_1$ and $s \xrightarrow{a} s'_2$. In this case, action a can lead to state s'_1 or s'_2. The next section will discuss nondeterminism in transition systems in more detail.

Figure 1.1 shows a simple example of a transition system. The initial state s (in this case the only initial state) of the transition system is marked by a small arrow pointing to it. From the initial state, three outgoing transitions are possible; action a leads to state t, action b leads to state u, and action c leads to state v. We call t, u, v *successor states* of u. To denote successor states, we often use primed variables, i.e., u', v', w', $\ldots$. In illustrations, we will sometimes omit state names, because such names have no effect on the observable behavior of a transition system.

Usually, there exist many transition systems that describe the same process. Thus, in order to specify processes using transition systems, there is a need for an equivalence relation that defines which transition systems are to be identified. Then, processes can be seen as equivalence classes of transition systems with regard to some chosen equivalence relation. Several commonly employed equivalence relations are defined in Section 2.2. See [vG01] for a complete survey. All equivalence definitions have a similar structure: two systems are defined to be equivalent iff, roughly speaking, any behavior observable in the first system can also be observed in the second system, and any behavior observable in the second system can also be observed in the first system. The different equivalence definitions vary in the notion of observability.

Omitting one of the directions of such equivalence definitions yields a preorder. Such preorders are often considered as refinement relations, usually in the sense that transition system M_1 refines transition system M_2 iff all observable behavior of M_1 can also be observed in M_2. However, as we will see, not all these preorders are suitable refinement notions, because sometimes every model could be refined to the "empty" one showing no observable behavior at all, which is certainly undesired, e.g. in the software development process: an empty program should usually not be a valid development result. However, some preorders have additional constraints that solve this problem. Then, refinement corresponds, roughly speaking, to the reduction of nondeterminism.

1.3 Nondeterminism

A transition system is deterministic iff, for every label a, every state has only a single successor state reachable via a transition labeled a. The behavior of deterministic transition systems can be completely controlled by the environment. This needs not be the case for nondeterministic transition systems. Suppose, for a state s and a label a, there are several a-successors of s. Then, the behavior following an occurrence of a appears to be nondeterministic to the environment, since it is not *under the control* of the environment.

Resolvable and persistent nondeterminism

Roughly speaking, preorders that represent reduction of nondeterminism allow for the removal of a-labeled transitions originating from a common state s, as long as one a-labeled transition originating from s remains. If the preorder is considered as refinement notion, this means that reducing nondeterminism is a *refinement* or, in other words, nondeterminism is used for the expression of abstraction. Designing a system having a state s with several a-successors means explicitly allowing different behaviors following a. It is then a refinement to decide for one of these behaviors (by removing all but one a-branch). As soon as every nondeterminism in every state has been resolved in this fashion, an executable concrete system, in other words an *implementation*, has been reached. This reflects the usual approach to develop complex systems, starting with an abstract model, which is then successively refined [Wir71]. We call this kind of nondeterminism *resolvable*, because it is designed to be resolved, usually by a system designer.[1] The resolution of this nondeterminism is thus not under the control of the environment, but it is under the control of the system designer.

However, nondeterminism in system models should not always be resolvable. For example, think of a model that repeatedly makes a random decision, by assigning a random value between 1 and 6 to a variable *dice*. Such a model should be considered an executable implementation, although it still contains a kind of nondeterminism, and in fact, it would be undesirable if a modeler would refine the model to always assign value 1 to *dice*. Consequently, we have a different kind of nondeterminism which must not be resolved by a designer and which also exists in implementations; we call it *persistent nondeterminism*. Here, it is not only beyond the control of the environment how nondeterminism is resolved, but it is also beyond the control of the concrete system itself.

For random decisions, as described above, one will usually want to specify a probability distribution, e.g. that every value between 1 and 6 is assigned a probability of $\frac{1}{6}$. However, for many applications, the distribution is unknown or not of importance. For example, persistent nondeterminism can be used to model failures, e.g. in a communication channel. Failure-tolerant algorithms are required to function in a proper predefined manner also in a worst case scenario (so the property to be proved is of the form "for all resolutions of persistent nondeterminism, it holds that ..."). For this, a precise distribution, stating which faults occur with which probability, is (a) not of importance and (b) probably unknown.

A further application of persistent nondeterminism concerns *abstract actions* [Tho87]. The abstraction we have considered so far is state-based: several more concrete states can be united to form a common, more abstract state, as is done in predicate abstraction. Actions, however, have been taken to be concrete, although it often makes sense to perform abstraction also with respect to actions. Then, actions with a similar meaning can be united and given a common, abstract name. Consider a state in which two outgoing transitions are labeled with actions *send_0* and *send_1*, respectively. If we unite these two actions into an abstract action *send*, this leads to nondeterminism because then we have two outgoing transitions labeled with the common action *send*. This nondeterminism must not be resolved in refinements because it

[1] In Chapter 4 we will take the view that the decision for a scheduler in a concurrent setting also constitutes a resolution of resolvable nondeterminism. Although one usually would not regard the scheduler as part of a concurrent program, the scheduler needs to be specified in order to fully describe the actual behavior of the program. This is usually done by the operating system designer rather than the designer of the program, but does not contradict the view that *some* designer performs this resolution of nondeterminism.

just resulted from action renaming; the concrete system has to remain enabled for both *send_0* and *send_1*. Thus, persistent nondeterminism can be introduced by abstract actions.

Nondeterminism in input/output systems

Remember that the term nondeterminism refers to the environment's view of the modeled system. If the environment can completely *control* the behavior of the system, we say that the system is deterministic. We generalize this concept to a more sophisticated semantic model where input or output actions are distinguished. This model will be used in Chapter 6 as semantic domain for a statecharts [Har87] variant. Here, we require states to be of one of two kinds: input-enabled states have to react to all possible input events (this *input-enabledness* is common to statechart variants) and locally controlled states have to generate a subset of output events.

For input-enabled states, nondeterminism occurs in the familiar fashion: if an input-enabled state has a single a-successor for every input action a, the environment has complete control over the transition to be taken from this state. Nondeterminism is introduced if there is an input action a such that there are several a-successors; now the environment can no longer control which transition is taken.

For locally controlled states, as already implied by the term "locally controlled", the environment has no control over the next step if there is more than one successor. Instead, it is under the system's control which output action to generate. Thus, nondeterminism already exists if there are several successors, regardless of their labeled output actions, i.e. regardless whether these are equal or not. A resolution of nondeterminism can remove not only branches with duplicate labels, but it can remove all branches as long as one remains.

Thus a deterministic input/output system has for every input action a in an input-enabled state at most one a-successor (by the input-enabledness condition *exactly* one), and in every locally controlled state at most one successor (by the deadlock-freedom condition, which we will later impose, *exactly* one). The details, together with the actual refinement notion, are presented in Section 2.3.3.

1.4 Refinement settings

We have seen that nondeterminism can be used to describe systems abstractly, or in other words, to *underspecify* systems. Thus, *underspecification* is a special kind of specification which leaves certain parts of the modeled system open. If some of the open choices of a specification are decided, we say that the specification is refined, or that it gets more concrete. If all open choices are decided, an implementation is reached which cannot be refined further and can be considered as "executable" on a computer. This thesis considers over a dozen different formalizations of this concept, called *refinement settings*. All these consist, roughly speaking, of a set of semantic models which are (extensions of) transition systems, an embedding of implementations into the semantic models, and a refinement preorder.

The refinement settings considered here can be classified (a) by their choice of what an implementation is (arbitrary or only deterministic transition systems), and (b) by their interpretation of actions (uniform or distinguishing input and output actions). This theoretically yields the following four classes of refinement settings:

- *Refinement settings over deterministic transition systems, with uniform action interpretation.* In these settings, *deterministic* transition systems are considered to be the implementations, and nondeterminism is considered resolvable. These settings cannot express persistent nondeterminism. Prominent examples are transition systems equipped with the *simulation* [Par81] or *ready simulation* [BIM95] preorder.

- *Refinement settings over transition systems, with uniform action interpretation.* Here, (possibly nondeterministic) transition systems are considered to be the implementations. Their possible nondeterminism is seen as persistent. For additionally expressing resolvable nondeterminism, it is required to extend the syntax of transition systems, e.g. by adding a new kind of transition. Prominent examples are *modal transition systems* [LT88] and *μ-automata* [JW95]. This thesis introduces a further setting called *ν-automata* which is a variant of μ-automata.

- *Refinement settings over deterministic transition systems, distinguishing input and output actions.* We introduce one setting of this kind, called *I/O-transition systems.* It corresponds to the notion of nondeterminism in input/output systems rather than to the classical nondeterminism of uniform process semantics which does not distinguish between input and output actions. This setting can only express resolvable nondeterminism.

- *Refinement settings over transition systems, distinguishing input and output actions.* Such a setting is suitable, if both resolvable and persistent nondeterminism shall be expressed in an input/output system. It could easily be defined, e.g. by combining the concepts of μ-automata and I/O-transition systems. However, there is no application for this kind of setting in the scope of this thesis and consequently we do not give a formal definition of such a refinement setting.

All refinement settings considered in this thesis are defined in Section 2.3.

1.5 Outline and publication history

Chapter 2 establishes the necessary preliminaries. First, basic notations are declared. Then, we introduce the formal notations of process semantics, starting with (deterministic) transition systems and several preorders as well as bisimulation equivalence [Mil80, Par81]. The notions of implementation and refinement settings are formally defined, followed by all implementation and refinement settings relevant to this thesis. This includes refinement settings over deterministic transition systems, namely transition systems with the *ready pair* [OH86], *ready trace* [BBK87], *failure pair* [BHR84], *failure trace* [Phi87], *ready simulation* [BIM95] and *possible worlds* [VDN98] preorders, and *synchronously-communicating transition systems* [FG07]. We continue with settings designed for use over (possibly nondeterministic) transition systems, namely *(disjunctive) mixed/modal transition systems* [DGG97, LT88, LX90] as well as *μ-automata* [JW95] and *ν-automata*, where the latter is a setting newly developed in this thesis which serves as the semantic domain in Chapter 5. Finally, we present another new setting that distinguishes between input and output actions. This will be used as semantic model for state diagrams in Chapter 6.

Chapters 3 to 6 present the main results of this thesis: Chapter 3 introduces a new theoretical approach to compare refinement settings with respect to their expressiveness, based on their sets of expressible implementations. This comparison is performed for a dozen refinement settings over deterministic transition systems by providing transformations between the settings, which not only establish an expressiveness hierarchy but can also be used to convert between the settings, thus enabling tool reuse.

Chapter 4 is the first work to argue that a combination of resolvable and persistent nondeterminism occurs in process algebra when interpreted in a concurrent setting, and gives a semantics in terms of a refinement setting over (possibly nondeterministic) transition systems.

The two kinds of nondeterminism in the context of state diagram semantics are considered in Chapter 5. In particular, this is the first work giving a state diagram semantics with both kinds of nondeterminism. Since existing refinement settings would lead to unnecessarily complex semantic models, we develop a new and in this context more succinct refinement setting, called ν-automata. We also give a semantic mapping for a simple state diagram variant, as well as a general transformation that can be applied to existing semantics if extended by persistent nondeterminism.

Chapter 6 deals with the integration of underspecification techniques into the model-based design approach, in particular into state diagrams. To achieve this, we develop a novel variant of state diagrams that allows more explicit and more expressive modeling of underspecification than existing variants. It is given a semantics in a newly developed semantic setting, which distinguishes between input and output actions. A set of given refinement patterns enables proven-to-be-correct stepwise refinement without the need to re-check correctness after each refinement step. Consequently we deliver the formal foundations for the development of a state-diagram-based design tool that ensures correctness at all stages of the development process.

Finally, Chapter 7 concludes and points out directions for future work. Abstracts for each of the main chapters follow, together with bibliographic information on the publication of the results.

Chapter 3: Expressiveness of refinement settings

Based on [FdFELS09]: Harald Fecher, David de Frutos-Escrig, Gerald Lüttgen, and Heiko Schmidt. On the expressiveness of refinement settings. In *Proceedings of the 3rd International Conference on Fundamentals of Software Engineering (FSEN 2009).* To appear in *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag.

An extended version of this paper has been submitted to Elsevier's *Science of Computer Programming (SCP)* journal.

This chapter compares popular refinement settings over deterministic transition systems regarding the sets of implementations they are able to express. The main result is an expressiveness hierarchy as well as language-preserving transformations between various settings. In addition to system designers, the main beneficiaries of the work presented here are tool builders who wish to reuse refinement checkers or model checkers across different settings.

Chapter 4: Nondeterminism in process algebra

Based on [FS07b]: Harald Fecher and Heiko Schmidt. Process algebra having inherent choice: Revised semantics for concurrent systems. In *Proceedings of the 4th Workshop on Structural Operational Semantics (SOS 2007)*, volume 192, issue 1 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 45–60. Elsevier, 2007.

This chapter gives both an operational and an axiomatic semantics to a process algebra with (i) a parallel operator interpreted in a concurrent setting, implying resolvable nondeterminism, and (ii) a persistent choice operator, which expresses persistent nondeterminism if applied to equal actions. To handle the different kinds of nondeterminism, the operational semantics uses *μ-automata* [JW95] as underlying semantic model. Soundness and completeness of our axiomatic semantics with respect to the operational semantics are proved.

Chapter 5: Nondeterminism in state diagrams

Based on [FHSS09]: Harald Fecher, Michael Huth, Heiko Schmidt, and Jens Schönborn. Refinement sensitive formal semantics of state machines with persistent choice. In *Proceedings of the Seventh International Workshop on Automated Verification of Critical Systems (AVoCS 2007)*, volume 250, issue 1 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 71–86. Elsevier, 2009.

The semantics of state diagrams contains resolvable nondeterminism, because models are usually underspecified. In this chapter, we give a formal operational semantics to state diagrams enriched with an operator expressing persistent nondeterminism, thus requiring a semantic domain supporting both resolvable and persistent nondeterminism. A new such refinement setting is introduced – a variant of μ-automata with a novel refinement relation –, and a sound three-valued satisfaction relation for properties expressed in the μ-calculus is given. We also show how existing state machine semantics can be adapted to support persistent nondeterminism.

Chapter 6: Top-down development of state diagrams

This chapter contains the youngest results. They have been submitted to *Fundamental Approaches to Software Engineering (FASE 2010)*, a member conference of the *European Joint Conferences on Theory and Practice of Software (ETAPS 2010)*.

This chapter defines a variant of state diagrams which has (i) a restricted set of features in order to avoid semantic ambiguities and (ii) an extended syntax in order to support top-down development. This variant is given a precise operational semantics using a refinement setting with input and output actions. A set of refinement patterns is established which can be used to perform correct-by-construction refinement steps along the development process. The chapter's results pave the way for the development and implementation of a tool supporting refinement in state diagrams.

Chapter 2

Preliminaries and Refinement Settings

This chapter first defines necessary notations used throughout the thesis. It formalizes processes in terms of transition systems and introduces preorders and equivalences relevant to this thesis. It also defines a precise notion of refinement setting, which to our knowledge has not been formulated in this fashion before. Finally, all refinement settings used in this thesis are introduced. Note that it is not necessary to work through all definitions in Subsections 2.3.1, 2.3.2, and 2.3.3 at this point; they can also be read "on demand" at the time of their use in the following chapters.

2.1 Basic notations

For a set M, let $|M|$ denote the cardinality of M and $\mathcal{P}(M)$ its power set. Special sets used in this thesis are $\mathbb{N} \overset{\text{def}}{=} \{0, 1, 2, 3, ...\}$ and $\mathcal{L}$, which is a fixed *finite* set of action labels. In Chapter 6, we refine the notion of labels by distinguishing between input and output actions, which in this context are called events and denoted by $\underline{\mathsf{Ev}}$ and $\overline{\mathsf{Ev}}$, respectively. Both $\underline{\mathsf{Ev}}$ and $\overline{\mathsf{Ev}}$ are fixed globally and required to be finite.

M^* denotes the set of finite sequences over M. We notate sequences by $\langle \ldots \rangle$; thus, $\langle\rangle$ is the empty sequence, and $\langle abc \rangle$ is the sequence of a, b, and c (in this order). Sequences are concatenated using operator $\cdot$, e.g. $\langle abc \rangle \cdot \langle def \rangle$ yields $\langle abcdef \rangle$.

Depending on the context, a function $f : M_1 \to M_2$ is also interpreted as a higher order function from $\mathcal{P}(M_1)$ to $\mathcal{P}(M_2)$ with $f(X_1) = \{f(m_1) \mid m_1 \in X_1\}$. For a function $f : A \to B$, let $f(A) = \bigcup_{a \in A} f(a)$ and let $f[a \mapsto b]$ denote function f except that a is mapped to b, i.e. $(f \setminus \{(a, f(a))\}) \cup \{(a, b)\}$ if $a \in A$, and $f \cup \{(a, b)\}$ otherwise.

For a binary relation $R \subseteq M_1 \times M_2$, we write $x_1 R x_2$ if $(x_1, x_2) \in R$ and let $R^{-1} = \{(x_2, x_1) \mid (x_1, x_2) \in R\}$. If $X_1 \subseteq M_1$, $X_2 \subseteq M_2$, we let $X_1.R = \{x_2 \in M_2 \mid \exists x_1 \in X_1 : (x_1, x_2) \in R\}$ and $R.X_2 = \{x_1 \in M_1 \mid \exists x_2 \in X_2 : (x_1, x_2) \in R\}$. Instead of $\{x\}.R$ and $R.\{x\}$, we shortly write $x.R$ and $R.x$, respectively. These notations are also employed for ternary relations $R \subseteq M_1 \times M_2 \times M_3$ that are then understood as binary relations $R \subseteq (M_1 \times M_2) \times M_3$ or $R \subseteq M_1 \times (M_2 \times M_3)$, depending on the context.

The above notations will be used extensively for transition relations, e.g. $\longrightarrow \subseteq S \times \mathcal{L} \times S$. Here, we write $s \xrightarrow{a} s'$ instead of $(s, a, s') \in \longrightarrow$, which yields notations $s. \xrightarrow{a}$ for the a-successors of s, and $\xrightarrow{a} .s'$ for the a-predecessors of s'. We write $s \overset{a}{\not\longrightarrow}$ iff $\{s\}. \xrightarrow{a} = \emptyset$ and we define the set of outgoing labels $\mathbf{O}_{\longrightarrow}(s)$ of $s \in S$ by $\{a \in \mathcal{L} \mid \exists s' \in S : s \xrightarrow{a} s'\}$. The outgoing labels of a state are sometimes also called the *enabled* labels of s.

In this thesis, proofs are written in a special proof environment introduced by "*Proof.*" and ended by symbol □. In case the proof contains a subproof, the end of the subproof is denoted by symbol ◇.

2.2 Process semantics

We describe processes in terms of *transition systems* which are essentially directed graphs with a set of designated initial nodes and labeled edges. The nodes stand for states of the process, and the edges denote state transitions that can be performed in case the action that corresponds to the label of the transition occurs.

Definition 2.1 (Transition system). *A* transition system (TS) *with respect to label set $\mathcal{L}$ is a tuple $(S, S^{\mathrm{i}}, \longrightarrow)$, where S is a set of states, $\emptyset \neq S^{\mathrm{i}} \subseteq S$ is a non-empty subset of initial states, and $\longrightarrow \subseteq S \times \mathcal{L} \times S$ is a transition relation. A transition system is called finite iff its set of states is finite. The set of transition systems is denoted by* TS.

A transition system is deterministic iff its initial state set is a singleton set and, for every state s and every label a, there is only a single successor from s via a.

Definition 2.2 (Deterministic TS). *A transition system $(S, S^{\mathrm{i}}, \longrightarrow)$ is deterministic iff $|S^{\mathrm{i}}| = 1$ and $\forall s \in S, a \in \mathcal{L} : |s. \xrightarrow{a}| \leq 1$. The set of deterministic transition systems is denoted by* $\mathsf{TS}^{\mathsf{det}}$.

Consequently, nondeterminism can occur in two ways: either through several initial states (then it is uncontrollable which initial state is actually chosen for an execution), or through several transition from one state with equal labels (then it is uncontrollable which transition is actually taken in case the action corresponding to the common label occurs).

Trace-based semantics

Usually there exist many transition systems that describe the same process. Thus in order to specify processes using transition systems, there is a need for an equivalence relation that defines which transition systems are to be identified. Then, processes can be seen as equivalence classes of transition systems with regard to the chosen equivalence relation.

For closed systems, this equivalence relation is usually based on the possible *traces* that can occur if the transition system is executed. Here, a trace is defined to be the sequence of labels attached to transitions taken. The corresponding execution must start in an initial state and then may move along transitions, stopping at an arbitrary moment or continuing until there is no further outgoing transition. Note, however, that we do not consider infinite traces. Two transition systems are considered equivalent iff they have the same sets of possible traces.

Definition 2.3 (Trace). *Let M be a* TS. *Define $\Theta^M_{\mathsf{T}} \subseteq S \times \mathcal{L}^*$ to be the smallest set satisfying*

(i) $\forall s \in S : (s, \langle\rangle) \in \Theta^M_{\mathsf{T}}$

(ii) $(s', \vartheta) \in \Theta^M_{\mathsf{T}} \wedge s \xrightarrow{a} s' \Rightarrow (s, \langle a \rangle \cdot \vartheta) \in \Theta^M_{\mathsf{T}}$

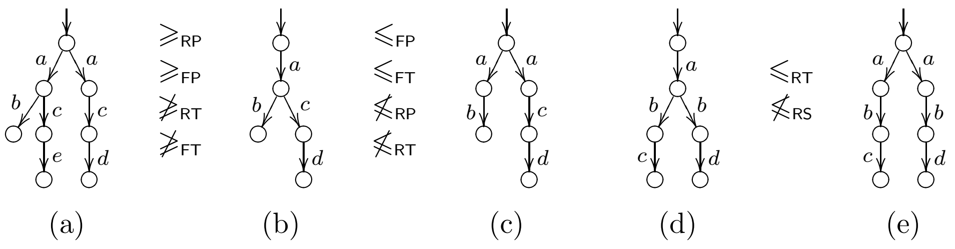

$\geqslant_{\mathsf{RP}}$
$\geqslant_{\mathsf{FP}}$
$\not\geqslant_{\mathsf{RT}}$
$\not\geqslant_{\mathsf{FT}}$
$\leqslant_{\mathsf{FP}}$
$\leqslant_{\mathsf{FT}}$
$\not\leqslant_{\mathsf{RP}}$
$\not\leqslant_{\mathsf{RT}}$
$\leqslant_{\mathsf{RT}}$
$\not\leqslant_{\mathsf{RS}}$
(a)
(b)
(c)
(d)
(e)

Figure 2.1: Illustration of the refinement preorders on transition systems. These examples are derived from Counterexamples 5, 6, and 8 of [vG01].

Then, $\vartheta \in \mathcal{L}^*$ is a trace *of M iff there is some $s \in S^{\mathrm{i}}$ such that $(s, \vartheta) \in \Theta^M_{\mathsf{T}}$.*

M_1 and M_2 are trace-equivalent iff the set of traces of M_1 equals the set of traces of M_2.

The so-called *completed traces* are those traces that correspond to an execution "of maximal length", i.e., for the execution's last state, say s, it holds that $\mathbf{O}_{\longrightarrow}(s) = \emptyset$.

Definition 2.4 (Completed trace). *Let M be a* TS. *Define $\Theta^M_{\mathsf{CT}} \subseteq S \times \mathcal{L}^*$ to be the smallest set satisfying*

(i) $\forall s \in S : \mathbf{O}_{\longrightarrow}(s) = \emptyset \Rightarrow (s, \langle\rangle) \in \Theta^M_{\mathsf{CT}}$

(ii) $(s', \vartheta) \in \Theta^M_{\mathsf{CT}} \land s \xrightarrow{a} s' \Rightarrow (s, \langle a\rangle \cdot \vartheta) \in \Theta^M_{\mathsf{CT}}$

Then, $\vartheta \in \mathcal{L}^*$ is a completed trace *of M iff there is $s \in S^{\mathrm{i}}$ such that $(s, \vartheta) \in \Theta^M_{\mathsf{CT}}$.*

M_1 and M_2 are completed trace-equivalent iff the set of completed traces of M_1 equals the set of completed traces of M_2.

Semantics based on (completed) traces are also called linear-time semantics because they do not at all consider the branching structure of a transition system. These semantics are suitable for closed systems where traces are the implementations. Abstract models in this context are Kripke structures [CGP01] or automata describing a set of traces, known as the abstract model's *language*. However, we are focusing on open systems here, and therefore we need to consider branching behavior which linear-time semantics based on traces cannot do. Consider, e.g., transition systems (b) and (c) in Figure 2.1. These systems are trace-equivalent although in system (b) the environment can control (by providing b or c after action a) whether d is executable afterwards or not, whereas for system (c), the environment has no means of such control (this system is nondeterministic). Therefore, linear-time semantics is not suitable for open systems in our context of refinement.

Consequently, the corresponding preorder, which relates two models iff the first has a set of traces that is a subset of the set of traces of the second, cannot be a suitable refinement notion. First, as seen in the abovementioned example, even equivalent systems can have different characteristics concerning controllability by the environment. Second, deterministic transition systems could be refined further because arbitrary transitions can be omitted, which can only restrict the set of possible traces. However, deterministic transition systems are considered implementations and as such are not refineable. Thus, suitable refinement notions may only remove transitions as long as another transition from the same state remains with the same label, which corresponds to reducing nondeterminism. In the following, several preorders are presented which are suitable as refinement notion because they only reduce nondeterminism.

The first four preorders are again based on traces, but they also take into account the sets of enabled labels (i.e. the sets of labels attached to all possible next transition steps). We start with the preorder based on *readiness semantics* [OH86], which we call *ready pair* refinement. A ready pair is a pair of a trace and the set of enabled labels in the trace's last state, and those are compared with respect to inclusion. If one now removes a (reachable) transition without another transition existing with the same label, one would get a new ready pair which has one label less in its set of enabled actions and, thus, this is no refinement, as desired. Instead, one may only remove transitions as long as this does not change the sets of enabled labels.

Definition 2.5 (Ready pair). *Let M be a* TS*. Define $\Theta^M_{\mathsf{RP}} \subseteq S \times (\mathcal{L}^* \cdot \mathcal{P}(\mathcal{L}))$ to be the smallest set satisfying*

(i) $\forall s \in S : (s, \langle \mathbf{O}_{\longrightarrow}(s)\rangle) \in \Theta^M_{\mathsf{RP}}$

(ii) $(s', \vartheta) \in \Theta^M_{\mathsf{RP}} \wedge s \xrightarrow{a} s' \Rightarrow (s, \langle a\rangle \cdot \vartheta) \in \Theta^M_{\mathsf{RP}}$

Then, $\vartheta \in \mathcal{L}^* \cdot \mathcal{P}(\mathcal{L})$ is a ready pair *of M iff there is $s \in S^{\mathrm{i}}$ such that $(s, \vartheta) \in \Theta^M_{\mathsf{RP}}$.*

M_1 ready pair-refines M_2, written $M_1 \leqslant_{\mathsf{RP}} M_2$, iff every ready pair of M_1 is a ready pair of M_2.

Ready trace semantics [BBK87] not only considers the set of enabled labels in the last state of an execution but also in every intermediate state. The corresponding preorder is presented in the following. A difference between ready pair refinement and ready trace refinement is illustrated by the systems shown in Figures 2.1(a) and (b).

Definition 2.6 (Ready trace). *Let M be a* TS*. Define $\Theta^M_{\mathsf{RT}} \subseteq S \times ((\mathcal{P}(\mathcal{L}) \cdot \mathcal{L})^* \cdot \mathcal{P}(\mathcal{L}))$ to be the smallest set satisfying*

(i) $\forall s \in S : (s, \langle \mathbf{O}_{\longrightarrow}(s)\rangle) \in \Theta^M_{\mathsf{RT}}$

(ii) $(s', \vartheta) \in \Theta^M_{\mathsf{RT}} \wedge s \xrightarrow{a} s' \Rightarrow (s, \langle(\mathbf{O}_{\longrightarrow}(s)\rangle \cdot \langle a\rangle \cdot \vartheta) \in \Theta^M_{\mathsf{RT}}$

Then, $\vartheta \in (\mathcal{P}(\mathcal{L})\cdot\mathcal{L})^*\cdot\mathcal{P}(\mathcal{L})$ is a ready trace *of M iff there is some $s \in S^{\mathrm{i}}$ such that $(s, \vartheta) \in \Theta^M_{\mathsf{RT}}$.*

M_1 ready trace-refines M_2, written $M_1 \leqslant_{\mathsf{RT}} M_2$, iff every ready trace of M_1 is a ready trace of M_2.

Failures semantics [BHR84] considers pairs of traces and label sets that correspond to the trace's last states, as for readiness semantics. However, the label sets do not contain all enabled labels in the trace's last state, but rather consist of labels which are *not* enabled (e.g., if the language of labels is $\{a, b, c\}$ and a transition system M has a ready pair $(\langle a\rangle\{a\})$, then $(\langle a\rangle\emptyset)$, $(\langle a\rangle\{b\})$, $(\langle a\rangle\{c\})$, and $(\langle a\rangle\{b, c\})$ are failure pairs of M. The corresponding preorder, which we call failure pair refinement, is presented in the following. A difference between ready pair refinement and failure pair refinement is illustrated by the systems shown in Figures 2.1(b) and (c).

Definition 2.7 (Failure pair). *Let M be a* TS*. Define $\Theta^M_{\mathsf{FP}} \subseteq S \times (\mathcal{L}^* \cdot \mathcal{P}(\mathcal{L}))$ to be the smallest set satisfying* *(i)* $\forall s \in S, F \subseteq \mathcal{L} \setminus \mathbf{O}_{\longrightarrow}(s) : (s, \langle F \rangle) \in \Theta^M_{\mathsf{FP}}$

(ii) $(s', \vartheta) \in \Theta^M_{\mathsf{FP}} \wedge s \xrightarrow{a} s' \Rightarrow (s, \langle a \rangle \cdot \vartheta) \in \Theta^M_{\mathsf{FP}}$

Then, $\vartheta \in \mathcal{L}^* \cdot \mathcal{P}(\mathcal{L})$ *is a* failure pair *of* M *iff there is some* $s \in S^{\mathrm{i}}$ *such that* $(s, \vartheta) \in \Theta^M_{\mathsf{FP}}$*.*

M_1 *failure pair-refines* M_2*, written* $M_1 \leqslant_{\mathsf{FP}} M_2$*, iff every failure pair of* M_1 *is a failure pair of* M_2*.*

Refusal, or *failure trace semantics* [Phi87] extends failures semantics in an analogous fashion like ready trace semantics extends readiness semantics. Again, not only are the failure sets (i.e. sets of not enabled labels) of the last state of an execution considered, but also the failure sets of all intermediate states. The corresponding preorder is presented in the following.

Definition 2.8 (Failure trace). *Let* M *be a* TS*. Define* $\Theta^M_{\mathsf{FT}} \subseteq S \times ((\mathcal{P}(\mathcal{L}) \cdot \mathcal{L})^* \cdot \mathcal{P}(\mathcal{L}))$ *to be the smallest set satisfying*

(i) $\forall s \in S, F \subseteq \mathcal{L} \setminus \mathbf{O}_{\longrightarrow}(s) : (s, \langle F \rangle) \in \Theta^M_{\mathsf{FT}}$

(ii) $(s', \vartheta) \in \Theta^M_{\mathsf{FT}} \wedge s \xrightarrow{a} s' \wedge F \subseteq \mathcal{L} \setminus \mathbf{O}_{\longrightarrow}(s') \Rightarrow (s, \langle F \rangle \cdot \langle a \rangle \cdot \vartheta) \in \Theta^M_{\mathsf{FT}}$

Then, $\vartheta \in (\mathcal{P}(\mathcal{L}) \cdot \mathcal{L})^* \cdot \mathcal{P}(\mathcal{L})$ *is a* failure trace *of* M *iff there is some* $s \in S^{\mathrm{i}}$ *such that* $(s, \vartheta) \in \Theta^M_{\mathsf{FT}}$*.*

M_1 *failure trace-refines* M_2*, written* $M_1 \leqslant_{\mathsf{FT}} M_2$*, iff every failure trace of* M_1 *is a failure trace of* M_2*.*

Simulation-based semantics

Semantics based on *simulations* lead to finer equivalences, when compared to the semantics presented so far. A simulation is a relation between the state sets of two transition systems M_1 and M_2. A pair (s_1, s_2) in the relation expresses that s_2 simulates s_1, i.e. all transitions that can be taken in s_1 can also be taken in s_2, and this should also hold after this transition step, so the "matching" successor states of s_1 and s_2 should again be related. Furthermore, every initial state of M_1 is required to be simulated by an initial state of M_2. Formally:

Definition 2.9 (Simulation). *Let* M_1*,* M_2 *be* TS*s. A* simulation *between* M_1 *and* M_2 *is a relation* $R \subseteq S_1 \times S_2$ *such that the following properties hold:*

(i) $\forall s_1 \in S^{\mathrm{i}}_1 : \exists s_2 \in S^{\mathrm{i}}_2 : (s_1, s_2) \in R$

(ii) For all $(s_1, s_2) \in R$ *and* $a \in \mathcal{L}$ *we have*

$$\forall s'_1 \in (s_1. \xrightarrow{a}_1) : \exists s'_2 \in (s_2. \xrightarrow{a}_2) : (s'_1, s'_2) \in R$$

M_1 *is simulated by* M_2*, written* $M_1 \leqslant_{\mathsf{S}} M_2$*, iff there is a simulation between* M_1 *and* M_2*.*

Relation $\leqslant_{\mathsf{S}}$ ("being simulated") is no suitable refinement notion, because, as for trace inclusion, the enabledness of actions is not preserved in refinement steps; in fact, the empty system with no transitions whatsoever is trivially simulated by any transition system. However, one can simply add the requirement that related states must have equal sets of enabled actions, which yields the refinement notion known as *ready simulation* [BIM95].

Definition 2.10 (Ready simulation). *Let M_1, M_2 be* TS*s. A* ready simulation *between M_1 and M_2 is a simulation between M_1 and M_2 satisfying $\forall(s_1, s_2) \in R : \mathbf{O}_{\longrightarrow}(s_1) = \mathbf{O}_{\longrightarrow}(s_2)$.*

M_1 is ready-simulated by (or ready simulation-refines) M_2, written $M_1 \leqslant_{\mathsf{RS}} M_2$, iff there is a ready simulation between M_1 and M_2.

We close this section with the definition of yet another refinement preorder called *possible worlds-refinement* [VDN98], which assigns to every nondeterministic transition system their sets of deterministic ready-simulation refinements (i.e. implementations) and compares those with respect to inclusion.

Definition 2.11 (Possible worlds). *Let M be a* TS*. $N \in \mathsf{TS}^{\mathsf{det}}$ is a* possible world *of M iff $N \leqslant_{\mathsf{RS}} M$.*

M_1 possible worlds-refines M_2, written $M_1 \leqslant_{\mathsf{PW}} M_2$, iff every possible world of M_1 is a possible world of M_2.

We have presented several preorders based on simulation but so far had no word about equivalences yet. We only present *bisimulation equivalence* [Par81] here, which not only requires simulations in both directions but also requires the same relation to be simulations in both directions.

Definition 2.12 (Bisimulation). *Let M_1, M_2 be* TS*s. A* bisimulation *between M_1 and M_2 is a simulation $R \subseteq S_1 \times S_2$ such that R^{-1} is a simulation between M_2 and M_1.*

M_1 and M_2 are bisimilar, written $M_1 \approx M_2$, iff there is a bisimulation between M_1 and M_2.

When arguing about deterministic transition systems only (e.g. when those are considered implementations and we only need a notion of equivalence on those), it makes no difference whether bisimulation equivalence or one of the other commonly used equivalences [vG01] is taken, because it has been shown by Park [Par81], and further examined by Engelfriet [Eng85], that all preorders collapse on deterministic transition systems.

2.3 Implementation and refinement settings

In the previous section, we formally presented several preorders on transition systems and one equivalence, called *bisimulation equivalence.* This makes it possible to consider deterministic transition systems, equipped with bisimulation equivalence, as implementations and using a preorder, e.g. ready simulation, as refinement preorder on all (i.e. nondeterministic and deterministic) transition systems.

Now we generally define the notions of *implementation setting* and *refinement setting* and what those have to satisfy. First, an implementation setting is a pair of (concrete) models and an equivalence relation on those models.

Definition 2.13 (Implementation setting). *An* implementation setting *is a tuple $(\mathsf{X}, \sim)$, where X is a set of models and $\sim$ is an equivalence relation on X.*

The approach discussed so far uses deterministic transition systems, together with bisimulation equivalence, as implementation setting. Note again that bisimulation equivalence could also be replaced by any other commonly used equivalence [vG01], because it has been shown by Park [Par81], and further examined by Engelfriet [Eng85], that all preorders collapse on deterministic transition systems.

Definition 2.14. *The implementation setting of deterministic transition systems is defined as* $(\mathsf{TS}^{\mathsf{det}}, \approx)$.

Different from the approach discussed so far, it is also possible to consider the set of *all* transition systems as implementations (i.e. not only the deterministic ones), together with an equivalence, e.g. bisimulation equivalence. Then, any occurring nondeterminism in implementations is interpreted as persistent.

Definition 2.15. *The implementation setting of transition systems is defined as* $(\mathsf{TS}, \approx)$.

Refinement settings are defined over a given implementation setting. Refinement settings over $(\mathsf{TS}^{\mathsf{det}}, \approx)$ can express resolvable nondeterminism only (e.g. the refinement setting based on ready simulation), whereas refinement settings over $(\mathsf{TS}, \approx)$ can describe both persistent and resolvable nondeterminism (e.g. μ-automata, to be introduced later).

Formally, a refinement setting over a given implementation setting consists of (i) a set of models, (ii) a distinguished subset of finite models, which for all refinement settings in this thesis will be defined to be those models with a finite state set, (iii) a preorder on the models and (iv) an embedding of the implementations into the models such that equivalence on implementations coincides with the refinement preorder on the refinement setting's models. Often, this embedding is the identity function, e.g. if the implementations are deterministic transition systems and the models are (general) transition systems. Non-trivial but usually straightforward embeddings are employed if the models are not transition systems but some kind of extension of transition systems. Such kinds of extensions are introduced in Sections 2.3.1 and 2.3.2.

Definition 2.16 (Refinement setting). *A* refinement setting *over the implementation setting* $(\mathsf{X}, \sim)$ *is a tuple* $\mathcal{X} = (\mathsf{Y}, \mathsf{Y}^{\mathsf{f}}, \leqslant, h)$*, where* Y *is a set of models,* Y^{f} *is a distinguished subset of so-called finite models*[1]*,* $\leqslant$ *is a preorder on* Y*, called* refinement*, and* $h : \mathsf{X} \to \mathsf{Y}$ *is an embedding, i.e.* $\forall M_1, M_2 \in \mathsf{X} : M_1 \sim M_2 \Leftrightarrow h(M_1) \leqslant h(M_2)$.

Though this is not required by the definition, first think of $h(\mathsf{X})$ as the bottom elements of the refinement preorder $\leqslant$. They correspond to the implementations. Then, equivalence on deterministic transition systems $(M_1 \sim M_2)$ must imply "refinement equivalence", i.e. $h(M_1) \leqslant h(M_2)$ and, directly implied by equivalence of $\sim$, $h(M_2) \leqslant h(M_1)$ on these bottom elements. For the other direction, refinement between models on the implementation level must be enough to establish equivalence on deterministic transition systems, which makes sure that every implementation can be specified by itself, without any other, non-equivalent refining implementations.

[1]For all refinement settings considered in this thesis, Y^{f} is the subset of Y with finite state set. In general, other definitions may be reasonable, too, e.g. finite branching properties. Furthermore, the definitions so far allow arbitrary mathematical objects to be "models"; thus, a definition of a subset of finite models cannot be directly derived. These are the reasons why we have to include component Y^{f} in the definition of a refinement setting, although this may not seem elegant.

In fact, the definition also allows non-implementations below implementation level, i.e. below elements from $h(\mathsf{X})$. These do not appear in the refinement settings corresponding to the preorders presented so far, but they exist in other settings presented later (e.g. mixed transition systems) and allow for the description of an *empty* set of implementations; in other words, they allow for *unsatisfiable* descriptions.

In the following, we present all implementation and refinement settings occurring in this thesis. It is easily checked that all presented refinement settings are indeed refinement settings.

2.3.1 Refinement settings over deterministic transition systems

We start with the presentation of refinement settings over deterministic transition systems. First, we give the formal definition of the settings based on those already presented preorders that are suitable as refinement notion.

Definition 2.17. *The refinement settings over* $(\mathsf{TS}^{\mathsf{det}}, \approx)$ *of* ready pair inclusion $\mathbb{T}\$_{\mathsf{RP}}$*,* ready trace inclusion $\mathbb{T}\$_{\mathsf{RT}}$*,* failure pair inclusion $\mathbb{T}\$_{\mathsf{FP}}$*,* failure trace inclusion $\mathbb{T}\$_{\mathsf{FT}}$*,* ready simulation $\mathbb{T}\$_{\mathsf{RS}}$*, and* possible worlds inclusion $\mathbb{T}\$_{\mathsf{PW}}$ *are defined as* $(\mathsf{TS}, \mathsf{TS}_{\mathsf{f}}, \leqslant, h)$*, where* TS_{f} *is the set of* TS*s with a finite state set,* h *is the identity embedding of* $\mathsf{TS}^{\mathsf{det}}$ *into* TS*, and* $\leqslant$ *is defined as* $\leqslant_{\mathsf{RP}}$*,* $\leqslant_{\mathsf{RT}}$*,* $\leqslant_{\mathsf{FP}}$*,* $\leqslant_{\mathsf{FT}}$*,* $\leqslant_{\mathsf{RS}}$*, and* $\leqslant_{\mathsf{PW}}$*, respectively.*

As already discussed, trace inclusion and simulation do not yield proper refinement settings over deterministic transition systems. This is because the requirement $h(M_1) \leqslant h(M_2) \Rightarrow M_1 \sim M_2$ of a refinement setting is not always satisfied, e.g., if M_1 is the empty system without transitions and M_2 has (reachable) transitions.

Synchronously-communicating transition systems

A commonly used refinement setting over deterministic transition systems is ready simulation ($\mathbb{T}\$_{\mathsf{RS}}$), as introduced in Definition 2.17. Here, every enabled action needs to remain enabled in every refinement. However, sometimes one may want to express that in certain states it may also be allowed to remove the enabledness with respect to certain actions. We now introduce a refinement setting which has the full expressiveness of ready simulation, but also allows to remove the enabledness of certain actions in certain states. In particular, this enables the modeler to define an abstract state that allows *any* subsequent behaviour, including "deadlock", which is not possible in $\mathbb{T}\$_{\mathsf{RS}}$.

This is the first time that we present a more sophisticated refinement setting where the implementations are still deterministic transition systems, but for the abstract models, an extension of transition systems is used, namely *synchronously-communicating transition systems* [FG07]. In fact, these are not more or less "synchronous" than ordinary transition systems, but since they have been introduced under that name, we will stick to it and prefer the abbreviation STS. Also, compared to [FG07], we consider a simplified variant of STSs without fairness. Our variant of STSs extends ordinary transition systems by so-called existence predicates on states, i.e. every state is assigned a set of labels. The intuition is that if a label is in the existence predicate of a state, then the enabledness of this label needs to be preserved in refinements, as in ready simulation. Accordingly, if a label is *not* in the existence predicate of a state, then the enabledness of this label may be removed in refinements, as in simulation.

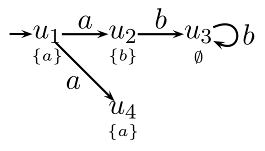
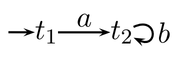
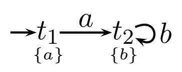
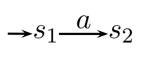
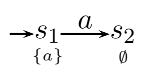
TS	STS
	M 
	$M_{\leqslant}$ 
	$M_{\not\leqslant}$ 

Figure 2.2: STS examples. Here, $\mathcal{L} = \{a, b\}$.

Following this intuition, one expects that a state may not have non-enabled labels in its existence predicate. Such systems are introduced as *must-saturated* STSs, and denoted by STS_+. However, for general STSs we do not make this restriction which introduces unsatisfiable models, i.e., models with an empty set of refining implementations (or shortly, with an empty language).

Definition 2.18 (Synchronously-communicating transition system [FG07]). *A* synchronously-communicating transition system (STS) *with respect to label set $\mathcal{L}$ is a tuple $(S, S^{\mathrm{i}}, \longrightarrow, e)$, where S is a set of states, $\emptyset \neq S^{\mathrm{i}} \subseteq S$ is a non-empty subset of initial states, $\longrightarrow \subseteq S \times \mathcal{L} \times S$ is a transition relation, and $e : S \to \mathcal{P}(\mathcal{L})$ is an existence predicate. An* STS *is called* must-saturated *iff $\forall s \in S : e(s) \subseteq \mathbf{O}_{\longrightarrow}(s)$.*

The set of STS*s is denoted by* STS*, and the set of must-saturated* STS*s is denoted by* STS_+*.*

The definition of STS-refinement extends simulation by the requirement that a more abstract state, compared to a more concrete state, must have a subset (or an equal set) of existence predicate labels. This allows one to express that, on an abstract level, the enabledness with respect to certain labels can be optional, whereas in more concrete systems, those decisions only get less (or equally) optional, with no such options remaining in implementations.

Definition 2.19 (STS-refinement [FG07]). *Let M_1, M_2 be* STS*s. An* STS-refinement *between M_1 and M_2 is a relation $R \subseteq S_1 \times S_2$ such that the following properties hold:*

(i) $\forall s_1 \in S_1^{\mathrm{i}} : \exists s_2 \in S_2^{\mathrm{i}} : (s_1, s_2) \in R$

(ii) For all $(s_1, s_2) \in R$ and $a \in \mathcal{L}$ we have

a) $\forall s_1' \in (s_1.\xrightarrow{a}_1) : \exists s_2' \in (s_2.\xrightarrow{a}_2) : (s_1', s_2') \in R$

b) $e_2(s_2) \subseteq e_1(s_1)$

M_1 STS-refines M_2, written $M_1 \leqslant_{\mathsf{STS}} M_2$, iff there is an STS-refinement between M_1 and M_2.

Although STSs allow for a simulation-like refinement, and simulation is no proper refinement setting, STS can be used in a proper refinement setting. This is achieved by mapping the implementations, i.e. deterministic transition systems, to those STSs where in every state the existence predicate corresponds exactly to the enabled labels in this state. Consequently, models from $h(\mathsf{TS}^{\mathsf{det}})$ cannot be refined further to non-equivalent models.

Definition 2.20 (STS-settings). *Let $h : \mathsf{TS}^{\mathsf{det}} \to \mathsf{STS}$ such that $h(S, S^{\mathrm{i}}, \longrightarrow) = (S, S^{\mathrm{i}}, \longrightarrow, \mathbf{O}_{\longrightarrow})$.*

(i) The refinement setting $\mathbb{S}\mathbb{T}\mathbb{S}$ over $(\mathsf{TS}^{\mathsf{det}}, \approx)$ is defined as $(\mathsf{STS}, \mathsf{STS}^{\mathrm{f}}, \leqslant_{\mathsf{STS}}, h)$, where $\mathsf{STS}^{\mathrm{f}}$ is the set of STSs having a finite state set.

(ii) The refinement setting $\mathbb{S}\mathbb{T}\mathbb{S}_+$ over $(\mathsf{TS}^{\mathsf{det}}, \approx)$ is defined as $(\mathsf{STS}_+, \mathsf{STS}_+^{\mathrm{f}}, \leqslant_{\mathsf{STS}}, h)$, where $\mathsf{STS}_+^{\mathrm{f}}$ is the set of STS_+s having a finite state set.

Example 2.21. *Figure 2.2 illustrates the embedding of two deterministic transition systems into STSs in the lower two rows, resulting in the concrete STSs $M_{\leqslant}$ and $M_{\not\leqslant}$, and presents in the uppermost row an abstract STS M, which is refined by model $M_{\leqslant}$, but is not refined by model $M_{\not\leqslant}$. In the graphical representation, existence predicates are drawn close to their corresponding states.*

Roughly spoken, M describes those implementations with completed traces $\langle ab \rangle$, $\langle abb \rangle$, $\langle abbb \rangle$, ... Deterministic transition system $M_{\not\leqslant}$, which has only the completed trace $\langle a \rangle$, is no refinement of M, as argued in the following. Suppose that in the definition of STS-refinement, we relate s_2 with u_2. Then we violate the required preservedness of existence predicates, because $\{b\} \not\subseteq \emptyset$. Consequently we need to relate s_2 with u_4, but this is not possible either, because $\{a\} \not\subseteq \emptyset$, contradiction. In fact, state u_4 is unsatisfiable (and M not must-saturated) because it requires by its existence predicate a following a-step, although no transition from u_4 permits it.

2.3.2 Refinement settings over transition systems

Now we present refinement settings which were originally designed for use with (general) transition systems as implementations. Nevertheless, for some of them we also define variants over deterministic transition systems, which leads to new, more succinct modeling possibilities.

All settings over (general) transition systems have in common that they express both persistent nondeterminism and resolvable nondeterminism. Persistent nondeterminism exists in implementations because those are not required to be deterministic (and for all settings introduced here, persistent nondeterminism may also exist in non-implementations). Resolvable nondeterminism is characterized by the refinement preorder, i.e. relation $\leqslant$.

Mixed/modal transition systems

Mixed [DGG97] and modal [LT88] transition systems extend ordinary transition systems by an additional transition relation, enabling the expression of so-called *must transitions* and *may transitions*. Must transitions describe behavior that must be present in an implementation, wheras may transitions describe behavior that is allowed to be present in an implementation; in other words, behavior *not* described by may transitions is forbidden.

Thus, persistent nondeterminism is expressed by several outgoing must transitions with the same label. All these outgoing transitions have to exist in any implementation. Resolvable nondeterminism is expressed by may transitions that are not must transitions: It is resolvable nondeterminism whether such may transitions do or do not occur in an implementation.

In contrast to mixed transition systems, modal transition systems have the additional requirement that every must-transition must also occur as may-transition, which excludes, similar to must-saturated STSs, unsatisfiable models.

Definition 2.22 (Mixed/modal transition system [DGG97, LT88]). *A* mixed transition system (MTS) *with respect to label set $\mathcal{L}$ is a tuple $(S, S^{\mathrm{i}}, \dashrightarrow, \longrightarrow)$, where S is a set of states, $\emptyset \neq S^{\mathrm{i}} \subseteq S$ is a non-empty subset of initial states, $\dashrightarrow \subseteq S \times \mathcal{L} \times S$ is a may transition relation, and $\longrightarrow \subseteq S \times \mathcal{L} \times S$ is a must transition relation. An* MTS *is called* modal transition system *iff $\longrightarrow \subseteq \dashrightarrow$.*

The set of mixed transition systems is denoted by MTS*, and the set of modal transition systems is denoted by* MTS_+*.*

The definition of refinement for MTSs and MTS_+s combine a simulation-like approach for the may transition relation with a "reverse" simulation-like approach for the must transition relation, thus allowing refinements with less allowed and more required behavior.

Definition 2.23 (MTS-refinement [DGG97, LT88]). *Let M_1, M_2 be* MTS*s. An* MTS-refinement *between M_1 and M_2 is a relation $R \subseteq S_1 \times S_2$ such that the following properties hold:*

(i) $\forall s_1 \in S_1^{\mathrm{i}} : \exists s_2 \in S_2^{\mathrm{i}} : (s_1, s_2) \in R$

(ii) For all $(s_1, s_2) \in R$ and $a \in \mathcal{L}$ we have

  a) $\forall s_1' \in (s_1 . \overset{a}{\dashrightarrow}_1) : \exists s_2' \in (s_2 . \overset{a}{\dashrightarrow}_2) : (s_1', s_2') \in R$

  b) $\forall s_2' \in (s_2 . \xrightarrow{a}_2) : \exists s_1' \in (s_1 . \xrightarrow{a}_1) : (s_1', s_2') \in R$

M_1 MTS-refines M_2, written $M_1 \leqslant_{\mathsf{MTS}} M_2$, iff there is an MTS-refinement between M_1 and M_2.

For implementations, allowed and required behavior coincides. Consequently, transition systems are embedded by defining both the may and must transition relation to be exactly the transition relation of the transition system.

Definition 2.24 (MTS-settings). *Let $h : \mathsf{TS} \to \mathsf{MTS}$ such that $h(S, S^{\mathrm{i}}, \longrightarrow) = (S, S^{\mathrm{i}}, \longrightarrow, \longrightarrow)$, and h^{det} its restriction to $\mathsf{TS}^{\mathsf{det}}$. Then the refinement settings $\mathbb{MTS}^{\mathsf{gen}}$, $\mathbb{MTS}_+^{\mathsf{gen}}$ over $(\mathsf{TS}, \approx)$, and $\mathbb{MTS}^{\mathsf{det}}$, $\mathbb{MTS}_+^{\mathsf{det}}$ over $(\mathsf{TS}^{\mathsf{det}}, \approx)$ are defined as $(\mathsf{MTS}, \mathsf{MTS}^{\mathrm{f}}, \leqslant_{\mathsf{MTS}}, h)$, $(\mathsf{MTS}_+, \mathsf{MTS}_+^{\mathrm{f}}, \leqslant_{\mathsf{MTS}}, h)$, $(\mathsf{MTS}, \mathsf{MTS}^{\mathrm{f}}, \leqslant_{\mathsf{MTS}}, h^{\mathsf{det}})$, and $(\mathsf{MTS}_+, \mathsf{MTS}_+^{\mathrm{f}}, \leqslant_{\mathsf{MTS}}, h^{\mathsf{det}})$, respectively, where X^{f} denotes the subset of models of X having a finite state set.*

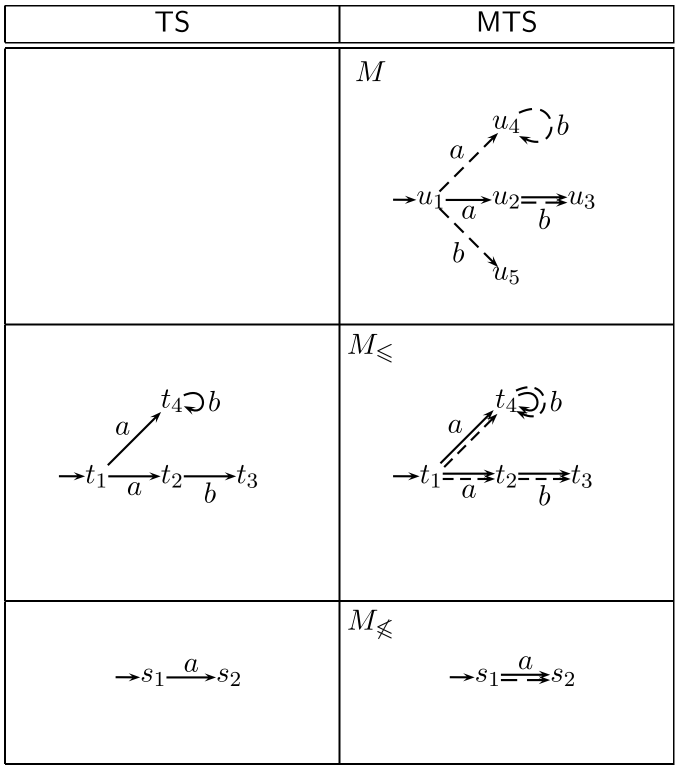


Figure 2.3: MTS examples. Here, $\mathcal{L} = \{a, b\}$.

Example 2.25. *Figure 2.3 illustrates the embedding of two transition systems into* MTS*s in the lower two rows, resulting in the concrete* MTS*s $M_{\leqslant}$ and $M_{\nleqslant}$, and presents in the uppermost row an abstract* MTS *M, which is refined by model $M_{\leqslant}$ but is not refined by model $M_{\nleqslant}$. In the graphical representation, may transitions are drawn as dashed arrows, whereas must transitions are drawn as solid arrows.*

M is a mixed, but not a modal transition system, because must transition $u_1 \xrightarrow{a} u_2$ has no corresponding may transition. Nevertheless, model $M_{\leqslant}$ is a refinement of M, because may transition $t_1 \overset{a}{\dashrightarrow} t_2$ can be matched by $u_1 \overset{a}{\dashrightarrow} u_4$ if we include pairs (t_2, u_4) and (t_3, u_4) in the refinement relation. Model $M_{\nleqslant}$ is no refinement of M, because must transition $u_2 \xrightarrow{b} u_3$ cannot be matched in $M_{\nleqslant}$.

Disjunctive mixed/modal transition systems

Disjunctive modal (and mixed) transition systems, introduced in [LX90] using a notation that slightly differs from ours, generalize modal (mixed) transition systems by modifying the must transition relation in that must transitions not only have a single target states, but multiple target states. These special kinds of transition are called *hypertransitions*. The intuition is that must hypertransitions correspond to a disjunctive choice, i.e. an implementation has to implement the behavior corresponding to (at least) one of its targets. Note that this intuition cannot directly be expressed in mixed/modal transition systems.

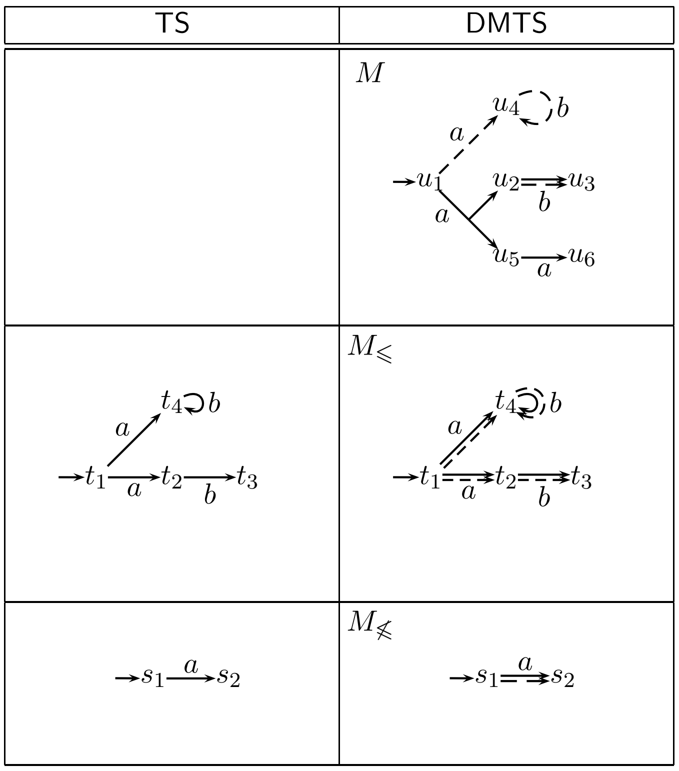


Figure 2.4: DMTS examples. Here, $\mathcal{L} = \{a, b\}$.

Again, the modal variant ensures that every model is satisfiable by requiring that every target set of a must hypertransition, say labeled by a, is non-empty and contains only targets reachable via a-labeled may transitions. The mixed variant does not make this restriction.

Definition 2.26 (Disjunctive mixed/modal transition system [LX90]). *A* disjunctive mixed transition system (DMTS) *with respect to label set $\mathcal{L}$ is a tuple $(S, S^{\mathrm{i}}, \dashrightarrow, \longrightarrow)$, where S is a set of states, $\emptyset \neq S^{\mathrm{i}} \subseteq S$ is a non-empty subset of initial states, $\dashrightarrow \subseteq S \times \mathcal{L} \times S$ is a may transition relation, and $\longrightarrow \subseteq S \times \mathcal{L} \times \mathcal{P}(S)$ is a must hypertransition relation. A* DMTS *is called* disjunctive modal transition system *iff all must hypertransition target sets are non-empty and only have elements that are also targets of may-transitions, i.e.*

$$\forall s \in S, a \in \mathcal{L}, \ddot{S} \in (s.\xrightarrow{a}) : \emptyset \neq \ddot{S} \subseteq (s.\overset{a}{\dashrightarrow})$$

The set of disjunctive mixed transition systems is denoted by DMTS*, and the set of disjunctive modal transition systems is denoted by* DMTS_+.

For may transitions, the refinement definition is as for modal/mixed transition systems. For must hypertransitions, we require for every target set in the more abstract system a target set in the more concrete system such that for all targets in the more concrete target set, we find a corresponding target in the more abstract target set.

Definition 2.27 (DMTS-refinement [LX90]). *Let M_1, M_2 be* DMTS*s. A* DMTS-refinement *between M_1 and M_2 is a relation $R \subseteq S_1 \times S_2$ such that the following properties hold:*

(i) $\forall s_1 \in S_1^{\mathrm{i}} : \exists s_2 \in S_2^{\mathrm{i}} : (s_1, s_2) \in R$

(ii) For all $(s_1, s_2) \in R$ *and* $a \in \mathcal{L}$ *we have*

a) $\forall s_1' \in (s_1.\overset{a}{\dashrightarrow}_1) : \exists s_2' \in (s_2.\overset{a}{\dashrightarrow}_2) : (s_1', s_2') \in R$

b) $\forall \ddot{S}_2 \in (s_2.\xrightarrow{a}_2) : \exists \ddot{S}_1 \in (s_1.\xrightarrow{a}_1) : \forall s_1' \in \ddot{S}_1 : \exists s_2' \in \ddot{S}_2 : (s_1', s_2') \in R$

M_1 DMTS*-refines* M_2*, written* $M_1 \leqslant_{\mathsf{DMTS}} M_2$*, iff there is a* DMTS*-refinement between* M_1 *and* M_2*.*

Transition systems are embedded by defining the may transition to be exactly the transition system relation, as for modal/mixed transition systems, and by giving every must hypertransition exactly one target, which then corresponds exactly to a transition in the transition system.

Definition 2.28 (DMTS-settings). *Let* $h : \mathsf{TS} \to \mathsf{DMTS}$ *such that* $h(S, S^{\mathrm{i}}, \longrightarrow) = (S, S^{\mathrm{i}}, \longrightarrow, \{(s, a, \{s'\}) \mid s \xrightarrow{a} s'\})$*, and* h^{det} *its restriction to* $\mathsf{TS}^{\mathsf{det}}$*. Then, the refinement settings* $\mathbb{DMTS}^{\mathsf{gen}}$*,* $\mathbb{DMTS}_{+}^{\mathsf{gen}}$ *over* $(\mathsf{TS}, \approx)$*, and* $\mathbb{DMTS}^{\mathsf{det}}$*,* $\mathbb{DMTS}_{+}^{\mathsf{det}}$ *over* $(\mathsf{TS}^{\mathsf{det}}, \approx)$ *are defined as* $(\mathsf{DMTS}, \mathsf{DMTS}^{\mathrm{f}}, \leqslant_{\mathsf{DMTS}}, h)$*,* $(\mathsf{DMTS}_{+}, \mathsf{DMTS}_{+}^{\mathrm{f}}, \leqslant_{\mathsf{DMTS}}, h)$*,* $(\mathsf{DMTS}, \mathsf{DMTS}^{\mathrm{f}}, \leqslant_{\mathsf{DMTS}}, h^{\mathsf{det}})$*, and* $(\mathsf{DMTS}_{+}, \mathsf{DMTS}_{+}^{\mathrm{f}}, \leqslant_{\mathsf{DMTS}}, h^{\mathsf{det}})$*, respectively, where* X^{f} *denotes the subset of models of* X *having a finite state set.*

Example 2.29. *Figure 2.4 illustrates the embedding of two transition systems into* DMTS*s in the lower two rows, resulting in the concrete* DMTS*s* $M_{\leqslant}$ *and* $M_{\nleqslant}$*, and presents in the uppermost row an abstract* STS M*, which is refined by model* $M_{\leqslant}$ *but is not refined by model* $M_{\nleqslant}$*. May transitions are drawn as dashed arrows, whereas must hypertransitions are drawn as solid arrows which may have several targets.*

M *is a disjunctive mixed transition system, but not a disjunctive modal transition system, because must hypertransition* $u_1 \xrightarrow{a} \{u_2, u_5\}$ *is lacking corresponding may transitions. The same is true for* $u_5 \xrightarrow{a} \{u_6\}$*. Nevertheless, model* $M_{\leqslant}$ *is a refinement of* M*, because the may transition* $t_1 \overset{a}{\dashrightarrow} t_2$ *can be matched by* $u_1 \overset{a}{\dashrightarrow} u_4$*, if we include pairs* (t_2, u_4) *and* (t_3, u_4) *in the refinement relation. Furthermore, we do not relate unsatisfiable state* u_5 *to any state in* $M_{\leqslant}$*. Model* $M_{\nleqslant}$ *is no refinement of* M*, because due to* $u_1 \xrightarrow{a} \{u_2, u_5\}$*, we need to relate either* u_2 *or* u_5 *with* s_2*, but* u_2 *requires a following* b*,* u_5 *requires a following* a*, and* s_2 *cannot provide either of those.*

The original definition of disjunctive modal transition systems [LX90] allows hypertransitions with different action labels for each target, i.e. the must hypertransition relation contains elements from $S \times \mathcal{P}(\mathcal{L} \times S)$ instead of $S \times \mathcal{L} \times \mathcal{P}(S)$. However, disjunctive modal transition systems in the notation of [LX90] can easily be transformed into our notation by using state copies, each employing such transitions with a common label.

At this point we illustrate why refinement settings originally designed for use over (general) transition systems, like $\mathbb{DMTS}^{\mathsf{gen}}$, can lead to new, more succinct modeling techniques when considering deterministic transition systems as implementations, like $\mathbb{DMTS}^{\mathsf{det}}$.

Example 2.30. *Consider the disjunctive mixed transition system of Figure 2.5(a), which describes a part of an abstract model of a tea and coffee vending machine. If regarded as an abstract model in* $\mathbb{DMTS}^{\mathsf{det}}$*, it expresses all implementations that offer, after an initial init action, a combination of tea and coffee variants, however, requiring that at least one kind of*

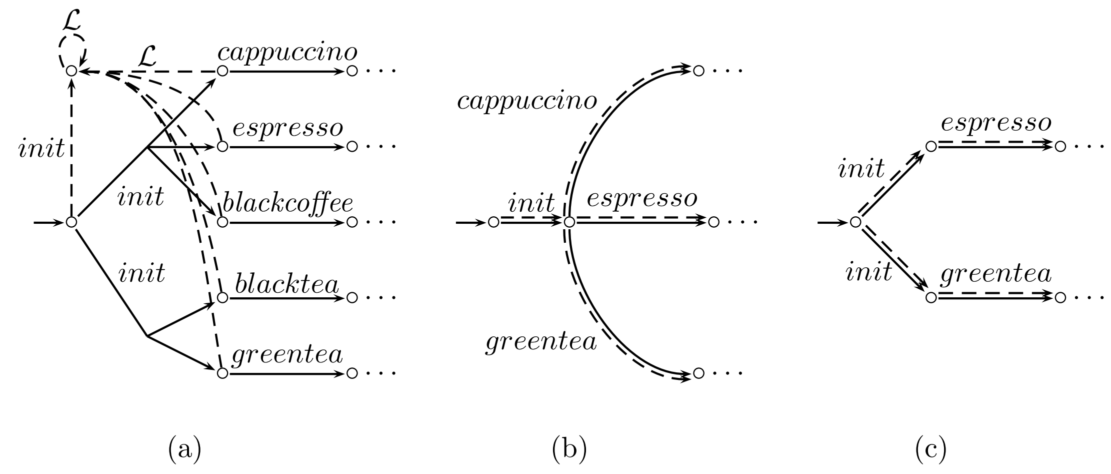


Figure 2.5: Illustration of new, succinct modeling possibilities with $\mathbb{DMTS}^{\mathsf{det}}$ (compared to $\mathbb{DMTS}^{\mathsf{gen}}$). Systems (b) and (c) are refinements of (a), but the nondeterministic implementation (c) is undesired.

coffee (cappuccino, espresso, or blackcoffee) and one kind of tea (blacktea or greentea) is offered. For instance, system (b) is a valid implementation. If regarded as an abstract model in $\mathbb{DMTS}^{\mathsf{gen}}$, though, there are undesired further implementations, e.g. system (c), where it is (persistently) nondeterministic, whether espresso or greentea is offered (this persistent nondeterminism is, of course, ruled out if we define implementations to be deterministic). Eliminating these undesired implementations in $\mathbb{DMTS}^{\mathsf{gen}}$ cannot be done as succinctly as (a) does with respect to $\mathbb{DMTS}^{\mathsf{det}}$; every combination of coffee and tea would have to be made explicit.

μ-automata

μ-automata do not take the approach of may and must transitions, as modal transition systems and their variants do. However, they can express resolvably nondeterministic choices, similar to the hypertransitions in disjunctive mixed/modal transition systems. This is made explicit by employing two different kinds of transitions, one for resolvable nondeterminism (called **or**-transitions) and one for execution steps (called **branch**-transitions, which also express persistent nondeterminism).

More precisely, a μ-automaton [JW95], written here in the notation of [DN05] (but omitting fairness and propositions), has two kinds of state, called **or**-states and **branch**-states, and two kinds of transition, namely unlabeled **or**-transitions from **or**- to **branch**-states and labeled **branch**-transitions from **branch**- to **or**-states. An equivalent notation would be to consider every **branch**-transition together with the following **or**-transitions as a single hypertransition, where its label is defined by the label of the **branch**-transition and its targets are defined by all succeeding **or**-transitions.

Definition 2.31 (μ-automaton [JW95, DN05]). *A* μ-automaton MUA *with respect to label set* $\mathcal{L}$ *is a tuple* $(S, \widetilde{S}, S^{\mathrm{i}}, \rightrightarrows, \longrightarrow)$*, where* S *is a set of so-called* ***or****-states,* $\widetilde{S}$ *is a set of so-called* ***branch****-states (disjoint from* S*),* $\emptyset \neq S^{\mathrm{i}} \subseteq S$ *is a non-empty subset of initial* ***or****-states,* $\rightrightarrows \subseteq S \times \widetilde{S}$ *is a so-called* ***or****-transition relation, and* $\longrightarrow \subseteq \widetilde{S} \times \mathcal{L} \times S$ *is a so-called* ***branch****-transition relation.*

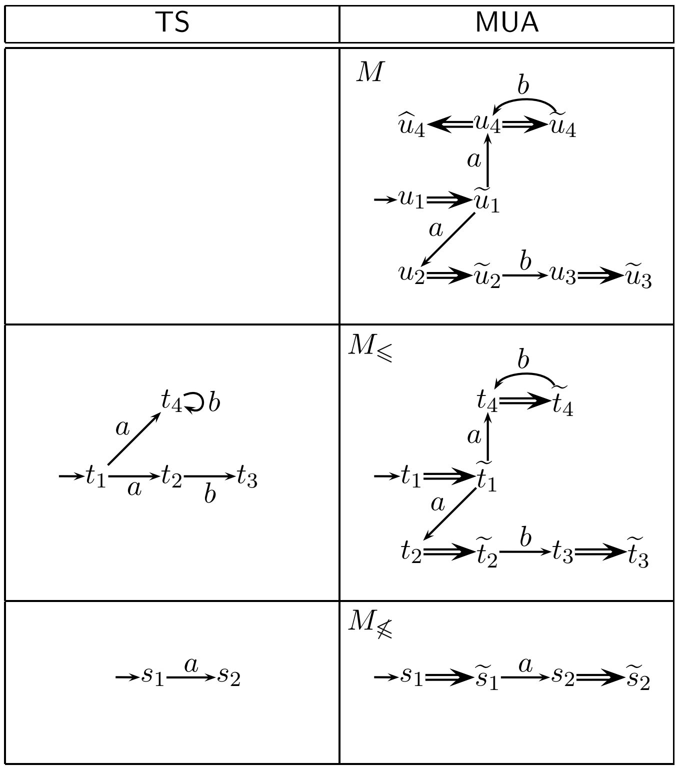


Figure 2.6: μ-automaton examples. Here, $\mathcal{L} = \{a, b\}$.

The set of μ-automata is denoted by MUA.

In μ-automata, persistent nondeterminism exists only in **branch**-states and is expressed by several outgoing **branch**-transitions with the same label. Resolvable nondeterminism exists only in **or**-states, because **or**-transitions can be removed in refinements, while **branch**-transitions cannot. Deciding for one **or**-transition resolves the nondeterminism and leads execution back to a concrete **branch**-state. For this reason, the **or**-transition relation is also called *concretization relation*. This is made precise by the standard refinement notion of μ-automata:

Definition 2.32 (μ-refinement [JW95, DN05])**.** *Let M_1, M_2 be* MUA*s. A μ-refinement between M_1 and M_2 is a relation $R \subseteq (S_1 \times S_2) \cup (\widetilde{S}_1 \times \widetilde{S}_2)$ such that the following properties hold:*

(i) $\forall s_1 \in S_1^{\mathrm{i}} : \exists s_2 \in S_2^{\mathrm{i}} : (s_1, s_2) \in R$

(ii) $\forall (s_1, s_2) \in R, \widetilde{s}_1 \in (s_1. \rightrightarrows) : \exists \widetilde{s}_2 \in (s_2. \rightrightarrows) : (\widetilde{s}_1, \widetilde{s}_2) \in R$

(iii) $\forall (\widetilde{s}_1, \widetilde{s}_2) \in R, a \in \mathcal{L}, s_1 \in (\widetilde{s}_1. \xrightarrow{a}) : \exists s_2 \in (\widetilde{s}_2. \xrightarrow{a}) : (s_1, s_2) \in R$

(iv) $\forall (\widetilde{s}_1, \widetilde{s}_2) \in R, a \in \mathcal{L}, s_2 \in (\widetilde{s}_2. \xrightarrow{a}) : \exists s_1 \in (\widetilde{s}_1. \xrightarrow{a}) : (s_1, s_2) \in R$

M_1 μ-refines M_2*, written $M_1 \leqslant_\mu M_2$, iff there is a* MUA*-refinement between M_1 and M_2.*

Transition systems are embedded into μ-automata by using **or**-states having exactly one outgoing **or**-transition.

Definition 2.33 (MUA-setting). *Let $h : \mathsf{TS} \rightarrow \mathsf{MUA}$ such that $h(S, S^{\mathrm{i}}, \longrightarrow) = (S, \widetilde{S}, S^{\mathrm{i}}, \{(s, \iota(s)) \mid s \in S\}, \{(\iota(s), a, s') \mid (s, a, s') \in \longrightarrow\})$, where $\widetilde{S} \cap S = \emptyset$ and $\iota : S \rightarrow \widetilde{S}$ is bijective. The refinement setting $\mathbb{MUA}$ is defined as $(\mathsf{MUA}, \mathsf{MUA}^{\mathrm{f}}, \leqslant_{\mu}, h)$, where $\mathsf{MUA}^{\mathrm{f}}$ is the set of MUAs having a finite state set.*

Example 2.34. *Figure 2.6 illustrates the embedding of two transition systems into μ-automata in the lower two rows, resulting in the concrete MUAs $M_{\leqslant}$ and $M_{\not\leqslant}$, and presents in the uppermost row an abstract MUA M, which is refined by model $M_{\leqslant}$ but is not refined by model $M_{\not\leqslant}$. In the graphical representation, the **or**-transitions are drawn as double-lined arrows, whereas the **branch**-transitions are drawn as single-lined arrows.*

μ-automaton M describes, roughly spoken, a persistent choice between either the completed trace $\langle ab \rangle$ or an initial a-step, followed by the unresolved choice to either take no further actions or to take a b-step, in which case we again have an unresolved choice to take no further actions, or to take a b-step again, in which case we again have the same unresolved choice, etc. $M_{\leqslant}$ is a refinement of M, which always resolves the resolvable nondeterminism to take a further b-step. Furthermore, it also provides completed trace $\langle ab \rangle$, which is required, because the nondeterminism in $\widetilde{u}_1$ is persistent. Since $M_{\not\leqslant}$ does not provide $\langle ab \rangle$, it is no refinement of M.

ν-automata

ν-automata are newly introduced in [FHSS09], on which Chapter 5 of this thesis is based. The motivation is that ν-automata, like μ-automata, fully support the combination of persistent and resolvable nondeterminism, but allow for a more succinct representation, especially in the context of state diagram semantics. The improved succinctness is discussed and illustrated by an example in Section 5.3.2.

Definition 2.35 (ν-automaton). *A ν-automaton* NUA *with respect to label set $\mathcal{L}$ is a tuple $(S, S^{\mathrm{i}}, \longrightarrow)$, where S is a set of states, $\emptyset \neq S^{\mathrm{i}} \subseteq S$ is a non-empty subset of initial states, and $\longrightarrow \subseteq S \times \mathcal{L} \times \mathcal{P}(S)$ is a hypertransition relation.*

The set of ν-automata is denoted by NUA.

The syntax of ν-automata is very close to that of μ-automata, although it does not use two transition relations but one hypertransition relation. However, if the junction points of hypertransitions are regarded as a special kind of state, we get an analogous syntax by identifying the **branch**-states of μ-automata with the ν-automaton states (both provide the menu of enabled actions at this point), and identifying the **or**-states of μ-automata with the junction points in ν-automata.

The significant difference between μ- and ν-automata shows when comparing the refinement notions and the embeddings of transition systems. ν-refinement allows for the removal of hypertransitions, but compares hypertransition targets with respect to bisimulation. Thus, resolvable nondeterminism is expressed by multiple hypertransitions from the same state with the same label, whereas multiple hypertransition targets express persistent nondeterminism. This is dual to the interpretation of μ-automata, since here multiple **branch**-transitions (corresponding to multiple hypertransitions in ν-automata) express persistent nondeterminism

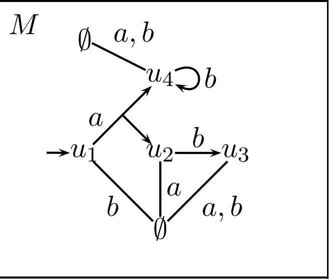
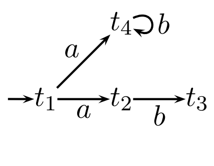
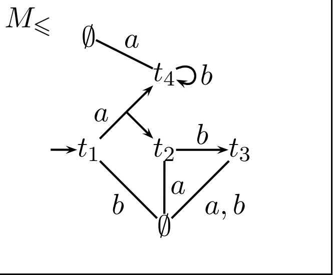
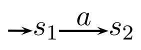
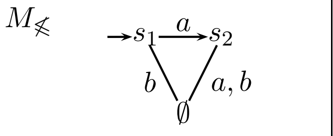
TS	NUA
	
	
	

Figure 2.7: ν-automaton examples. Here, $\mathcal{L} = \{a, b\}$. Transitions labeled by several actions are a short notation for several transitions, with one of the annotated actions each.

and multiple **or**-transitions (corresponding to multiple hypertransition targets in ν-automata) express resolvable nondeterminism.

Thus, ν-automata provide (per label) a resolvable choice of several persistent choices, whereas μ-automata provide (per label) a persistent choice of several resolvable choices. This makes ν-automata more succinct, e.g., in the context of state diagram semantics, as discussed in Section 5.3.2.

The definition of ν-refinement reflects the exchanged order of resolvable and persistent nondeterminism. For every more concrete hypertransition, we need to find a more abstract hypertransition such that their targets correspond in both directions.

Definition 2.36 (ν-refinement). *Let M_1, M_2 be* NUA*s. A ν-refinement between M_1 and M_2 is a relation $R \subseteq S_1 \times S_2$ such that the following properties hold:*

(i) $\forall s_1 \in S_1^{\mathrm{i}} : \exists s_2 \in S_2^{\mathrm{i}} : (s_1, s_2) \in R$

(ii) For all $(s_1, s_2) \in R$, $a \in \mathcal{L}$, and $\ddot{S}_1 \in (s_1. \xrightarrow{a}_1)$ there is $\ddot{S}_2 \in (s_2. \xrightarrow{a}_2)$ such that

a) $\forall s_1' \in \ddot{S}_1 : \exists s_2' \in \ddot{S}_2 : (s_1', s_2') \in R$

b) $\forall s_2' \in \ddot{S}_2 : \exists s_1' \in \ddot{S}_1 : (s_1', s_2') \in R$

*M_1 ν-*refines *M_2, written $M_1 \leqslant_\nu M_2$, iff there is a* NUA*-refinement between M_1 and M_2.*

Note that the definition of ν-refinement does not explicitly require that enabled actions are preserved, although it has been discussed that this is compulsory for a suitable refinement setting. We do not need an explicit formulation of this requirement here, because we embed transition systems in a way that defines concrete ν-automata to be enabled with respect to each action in each state. Thus, if one refines in such a way that a state is no longer enabled with respect to each action, this state becomes unsatisfiable, i.e., it has no refining implementations.

Transition systems are embedded into ν-automata by defining for each state s and each label a exactly one hypertransition, with the targets being defined by all a-successors of s in the transition system. This way, the branching structure of the transition system is moved into the hypertransition targets of a single hypertransition. Note that this can also lead to an empty target set. After this transformation, further hypertransitions with the same label can be used to express resolvable nondeterminism.

Definition 2.37 (NUA-setting). *Let $h : \mathsf{TS} \to \mathsf{MUA}$ such that $h(S, S^{\mathrm{i}}, \longrightarrow) = (S, S^{\mathrm{i}}, \{(s, a, (s.\xrightarrow{a})) \mid s \in S \wedge a \in \mathcal{L}\})$. The refinement setting $\mathbb{NUA}$ is defined as $(\mathsf{NUA}, \mathsf{NUA}^{\mathrm{f}}, \leqslant_{\nu}, h)$, where $\mathsf{NUA}^{\mathrm{f}}$ is the set of NUAs having a finite state set.*

Example 2.38. *Figure 2.7 illustrates the embedding of two transition systems into ν-automata in the lower two rows, resulting in the concrete NUAs $M_{\leqslant}$ and $M_{\nleqslant}$, and presents in the uppermost row an abstract NUA M, which is refined by model $M_{\leqslant}$ but is not refined by model $M_{\nleqslant}$. Hypertransitions with an empty target set are drawn as lines connected to symbol $\emptyset$. If hypertransitions are labeled with several, comma-seperated labels, this is a notation for several transitions with the same source and the same targets, each labeled with one of the labels.*

ν-automaton M describes the same behavior as μ-automaton M from Figure 2.6 (Example 2.34). Analogous arguments as to why $M_{\leqslant}$ is a refinement of M, but $M_{\nleqslant}$ is not, can be applied here.

2.3.3 A refinement setting with input and output actions

In Chapters 5 and 6 we will give a semantics to statecharts variants. In this context, actions (i.e., observable steps of behavior) are usually called *events*. In contrast to actions which are interpreted purely uniformly, we distinguish two kinds of events: *input* and *output* events, where input events (from a set $\underline{\mathsf{Ev}}$) are controlled by the environment and the modeled system only reacts on it, and output events (from a set $\overline{\mathsf{Ev}}$) are controlled by the modeled system.

As has already been discussed in Section 1.3, this differentiation leads to a different notion of nondeterminism. Several outgoing transitions with different input events are considered deterministic, because the environment can control which transition is taken, by selecting the corresponding event. Several outgoing transitions with different output events, however, are considered nondeterministic, because the environment cannot control which transition is taken; this is completely under the modeled system's control.

The interpretation of nondeterminism for a state which has both outgoing transitions with input and output events is unclear. One possible solution would be to give priority to either input or output events, but then the transitions with less priority could be left out anyway.

We avoid these problems by requiring every state to either only generate output events (these state are called *locally controlled states*) or only react to input events (these state are called *input-enabled states*). Furthermore, we demand that every input-enabled state is enabled with respect to *each* input label, i.e., the set of enabled events in input-enabled states must be $\underline{\mathsf{Ev}}$. We will see that these restrictions fit state diagrams, while at the same time keeping the semantic model as simple as possible.

Now, we formally define the implementation and refinement setting of so-called *I/O-transition systems.*

Definition 2.39 (I/O-transition system). *An* I/O-transition system (IOTS) *with respect to input event set* $\underline{\mathsf{Ev}}$ *and output event set* $\overline{\mathsf{Ev}}$ *is a tuple* $(S, S^{\mathrm{i}}, \longrightarrow)$*, where* S *is a set of states,* $\emptyset \neq S^{\mathrm{i}} \subseteq S$ *is a subset of initial states, and* $\longrightarrow \subseteq S \times (\underline{\mathsf{Ev}} \cup \overline{\mathsf{Ev}}) \times S$ *is a transition relation, such that every configuration either handles all input events (and no output events), or at least one output event (and no input events), i.e.* $\forall s \in S : (\forall \underline{e} \in \underline{\mathsf{Ev}} : (s.\xrightarrow{\underline{e}}) \neq \emptyset \wedge \forall \overline{e} \in \overline{\mathsf{Ev}} : (s.\xrightarrow{\overline{e}}) = \emptyset) \vee (\exists \overline{e} \in \overline{\mathsf{Ev}} : (s.\xrightarrow{\overline{e}}) \neq \emptyset \wedge \forall \underline{e} \in \underline{\mathsf{Ev}} : (s.\xrightarrow{\underline{e}}) = \emptyset)$.

Thus I/O-transition systems are simply a subset of transition systems. However, we equip them with a different notion of determinism.

Definition 2.40 (Deterministic IOTS). *An I/O-transition system* $(S, S^{\mathrm{i}}, \longrightarrow)$ *is deterministic iff every input-enabled configuration has only a single successor per label and every locally controlled configuration has only a single successor via a single output event, i.e.* $\forall s \in S : (\forall \underline{e} \in \underline{\mathsf{Ev}} : |s.\overset{\underline{e}}{\longmapsto}| = 1) \vee (\sum_{\overline{e} \in \overline{\mathsf{Ev}}} |s.\overset{\overline{e}}{\longmapsto}| = 1)$*. The set of deterministic I/O-transition systems is denoted by* $\mathsf{IOTS}^{\mathsf{det}}$.

We consider the implementation setting of I/O-transition systems together with bisimulation equivalence.

Definition 2.41. *The implementation setting of deterministic I/O-transition systems is defined as* $(\mathsf{IOTS}^{\mathsf{det}}, \approx)$.

The refinement setting of I/O-transition systems over $(\mathsf{IOTS}^{\mathsf{det}}, \approx)$ uses the preorder of simulation and the identity embedding.

Definition 2.42 (IOTS-setting). *The refinement setting of* $\mathbb{IOTS}$ *over* $(\mathsf{IOTS}, \approx)$ *is defined as* $(\mathsf{IOTS}, \mathsf{IOTS}^{\mathrm{f}}, \leqslant_{\mathrm{s}}, h)$*, where* h *is the identity embedding of* $\mathsf{IOTS}^{\mathsf{det}}$ *into* IOTS*, and* $\mathsf{IOTS}^{\mathrm{f}}$ *is the set of* IOTS*s with a finite state set.*

It may seem surprising that simulation is used here as refinement notion, since after Definition 2.9, where simulation is defined, we have argued that simulation does not yield a suitable refinement notion because it does not preserve the enabledness of actions. However, this problem does not arise in the context of IOTSs: Input events remain enabled, since we require any input-enabled state to be enabled with respect to each event, so the set of enabled events is always $\underline{\mathsf{Ev}}$ and can never be changed in a refinement. Concerning output events, we have to be aware of the different notion of determinism for I/O-transition systems. Downsizing the set of enabled output events, as long as one output event is left, now corresponds to reducing nondeterminism; the restrictions of IOTSs make sure that there is at least one enabled output event in *any* locally controlled state. Consequently, when applied to IOTSs, simulation *does* yield a suitable refinement notion.

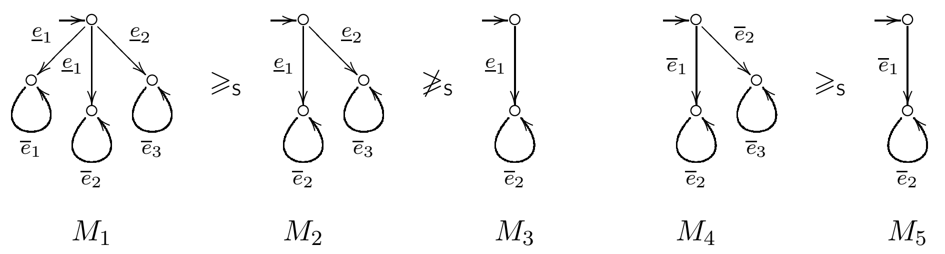


Figure 2.8: I/O-transition systems illustrating refinement. Here, $\underline{\mathsf{Ev}} = \{\underline{e}_1, \underline{e}_2\}$ and $\overline{\mathsf{Ev}} = \{\overline{e}_1, \overline{e}_2, \overline{e}_2\}$.

Example 2.43. *Figure 2.8 illustrates the refinement setting $\mathbb{IOTS}$. M_1 has nondeterminism in its input-enabled initial state: it is a refinement to remove one of the branches reacting on $\underline{e}_1$, which may yield, e.g., M_2. This makes sense, because it is not under the environment's control which of the branches labeled $\underline{e}_1$ is taken, provided $\underline{e}_1$ occurs. M_2 is deterministic, i.e. no further branch can be removed, because every refinement needs to remain enabled for each input event. In this situation, the environment can decide which transition to take by sending the corresponding event. Thus, M_3 is no refinement of M_2.*

For locally controlled states, there is a different interpretation: M_4 is nondeterministic, and M_5 is an refinement of it, because it is enough to generate only one event in the refinement. Therefore, the other branch can be removed. This makes sense, because here it is not under the environment's control which transition is taken.

Chapter 3

Expressiveness of Refinement Settings

This chapter compares popular refinement settings over deterministic transition systems with respect to the sets of implementations that they are able to express. As the main result, we establish and prove an expressiveness hierarchy, as well as language-preserving transformations between the various settings. In addition to system designers, the main beneficiaries of the work presented here are tool builders who wish to reuse refinement checkers or model checkers across different settings.

3.1 Introduction

Many of today's embedded systems employ control software that runs on specialized computer chips, performing dedicated tasks often without the need of an operating system. System designers typically specify such software using notations based on labeled transition systems: a possibly nondeterministic specification allows for a set of deterministic implementations, amenable to quality checks via testing or model checking. Verifiers benefit from the reduced state space in possibly nondeterministic abstractions from deterministic implementations. Choosing a suitable *refinement setting* for a given application depends on various aspects, e.g., expressiveness, conciseness and verification support.

In the concurrency-theory literature many refinement settings have been studied, with a focus on compositionality and full abstraction of, and logical characterizations and decision procedures for, the underlying refinement preorders. Less attention has been paid to questions of expressiveness. In the context of top-down development, where sets of allowed implementations are specified at different design levels, it is of special interest to characterize the expressible sets of implementations. In general, the more implementation sets a formalism can describe, the more expressive it is and the more flexibility a system designer has.

We perform the expressiveness comparison using language-preserving transformations, where the *language* of a refinement setting is its expressible set of deterministic implementations. This is analogous to trace-based languages, where language-preserving transformations have been developed between automata that differ in their notions of fairness (Büchi, Muller, Rabin, Streett, parity automata).

Language-preserving transformations are also valuable in the context of model checking, where abstract models reduce the size of the state space, while at the same time staying amenable to quality checks: if a property is model checked for an abstract model, then it is guaranteed to hold for each of its implementations. Therefore, a model checking tool over a refinement

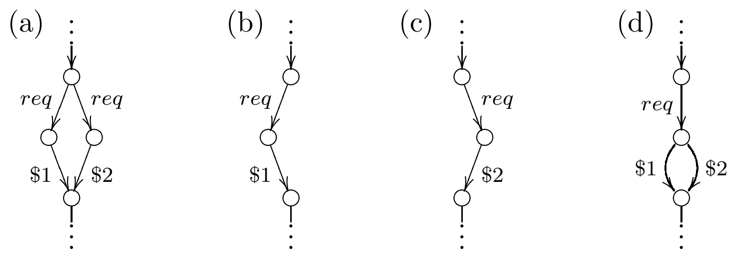


Figure 3.1: Vending machine example.

setting $(\mathsf{X}_1, \mathsf{X}_1^{\mathsf{f}}, \leqslant_1, h_1)$ can be reused for another setting $(\mathsf{X}_2, \mathsf{X}_2^{\mathsf{f}}, \leqslant_2, h_2)$ if every model in X_2 can be converted into an equivalent model in X_1 that defines the same language.

This chapter studies and compares the expressiveness of a dozen refinement settings designed for deterministic transition systems, namely $\mathbb{TS}_{\mathsf{RP}}$, $\mathbb{TS}_{\mathsf{FP}}$, $\mathbb{TS}_{\mathsf{RT}}$, $\mathbb{TS}_{\mathsf{FT}}$, $\mathbb{TS}_{\mathsf{RS}}$, $\mathbb{TS}_{\mathsf{PW}}$, $\mathbb{STS}$, $\mathbb{STS}_+$, $\mathbb{MTS}^{\mathsf{det}}$, $\mathbb{MTS}_+^{\mathsf{det}}$, $\mathbb{DMTS}^{\mathsf{det}}$, and $\mathbb{DMTS}_+^{\mathsf{det}}$. To do so, several intricate language-preserving transformations are developed and proven correct. We also show how algorithms for checking a specification's consistency, i.e. for checking non-emptyness of its language, and for checking refinement can be derived from our transformations. While the expressiveness hierarchy is valuable for informing system designers on their choice of refinement setting, our transformations allow tool builders to reuse their refinement checkers or model checking algorithms across different settings.

Outline. Section 3.2 defines the notion of a *language-preserving transformation*, on which our comparison is based. The comparison is then performed in Section 3.3. Section 3.4 presents some interesting related results. Finally, we discuss related work in Section 3.5 and conclude in Section 3.6.

3.2 Language-based expressiveness

This chapter focuses on refinement settings over deterministic transition systems. An informal discussion on why deterministic transition systems are the natural model for implementations in the context of open systems, has already been given in Section 2.2. We call the set of refining deterministic transition systems (i.e., implementations) of a model its *language*. We compare refinement settings with respect to their expressible languages. An example of a situation where only one of two settings is expressive enough for a given application in being able to describe a desired language, follows:

Example 3.1. *Consider a part of a vending machine specification, as shown in Figure 3.1(a). The transition system is nondeterministic since following a req action it can either ask for \$1 or for \$2. This nondeterminism is desired because the specification should be refinable to either a "cheap" machine (requesting \$1, shown in Figure 3.1(b)) or an "expensive" machine (requesting \$2, shown in Figure 3.1(c)). However, it depends on the used refinement setting, whether these two implementations can be modeled without also modeling the undesired implementation shown in Figure 3.1(d), which gives the user the choice whether to pay \$1 or \$2. For instance, with respect to failure pair inclusion, model (a) has the undesired implementation (d), whereas with respect to ready pair inclusion, it has not.*

We formalize our notion of expressiveness. One setting $\mathcal{X}_1$ should be considered at least as expressive as another setting $\mathcal{X}_2$ iff, for every language described by a model in $\mathcal{X}_2$, we find a model in $\mathcal{X}_1$ with the same language. Such a mapping from the models in $\mathcal{X}_2$ to the models in $\mathcal{X}_1$ is called *language-preserving transformation*. The following definition makes our notion of expressiveness more precise:

Definition 3.2 (Language-preserving transformation)**.** *Let $\mathcal{X} = (\mathsf{X}, \mathsf{X}^{\mathrm{f}}, \leqslant, h)$ be a refinement setting. The* language $\mathcal{X}(M)$ *of $M \in \mathsf{X}$ (also called "possible worlds") is $\{N \in \mathsf{TS}^{\mathsf{det}} \mid h(N) \leqslant M\}$. A* language-preserving transformation *from $\mathcal{X}_1 = (\mathsf{X}_1, \mathsf{X}^{\mathrm{f}}_1, \leqslant_1, h_1)$ to $\mathcal{X}_2 = (\mathsf{X}_2, \mathsf{X}^{\mathrm{f}}_2, \leqslant_2, h_2)$ is a total function $f : \mathsf{X}^{\mathrm{f}}_1 \to \mathsf{X}^{\mathrm{f}}_2$ such that $\mathcal{X}_1(M) = \mathcal{X}_2(f(M))$ for all $M \in \mathsf{X}^{\mathrm{f}}_1$. We say that $\mathcal{X}_1$ is* at least as expressive as *$\mathcal{X}_2$ if there is a language-preserving transformation from $\mathcal{X}_2$ to $\mathcal{X}_1$.*

By the definition, language-preserving transformations are mapping *finite* models to *finite* models[1] (though implementations may be infinite). We are especially interested in such mappings because they preserve the (direct) amenability to applications of tools such as model checking. Furthermore, language-based expressiveness results in our sense are trivial for *infinite* models, because infinite initial state sets can be used to describe any desired language (by introducing one initial state for each desired implementation).

Example 3.3. *Reconsider Example 3.1 where we claimed that model (a) expresses, with respect to ready pair inclusion, implementations (b) and (c), whereas it also has the further implementation (d) with respect to failure pair semantics. We can use this as a starting point to prove that ready pair inclusion ($\mathbb{T}\$_{\mathsf{RP}}$) is more expressive than failure pair inclusion ($\mathbb{T}\$_{\mathsf{FP}}$). Two properties need to be shown:*

(i) There is a language-preserving transformation from $\mathbb{T}\$_{\mathsf{FP}}$ to $\mathbb{T}\$_{\mathsf{RP}}$, i.e., every language expressible in $\mathbb{T}\$_{\mathsf{FP}}$ can also be expressed in $\mathbb{T}\$_{\mathsf{RP}}$.

(ii) There is no language-preserving transformation from $\mathbb{T}\$_{\mathsf{RP}}$ to $\mathbb{T}\$_{\mathsf{FP}}$, i.e., there is an expressible set of implementations with respect to $\mathbb{T}\$_{\mathsf{RP}}$, which cannot be expressed in $\mathbb{T}\$_{\mathsf{FP}}$.

For proving the first property, a language-preserving transformation should be given. This is done in Transformation 3.6. For the second property, it would be sufficient to prove that all *specifications that express (b) and (c) in $\mathbb{T}\$_{\mathsf{FP}}$ also express (d), since $\mathbb{T}\$_{\mathsf{RP}}$ can express (b) and (c) without (d). This kind of argumentation is adopted in the proof of Lemma 3.7.*

3.3 Comparison

This section establishes the expressiveness hierarchy presented in Figure 3.2. This is done constructively by developing language-preserving transformations or showing their non-existence by counter-example. In particular, we have paid attention to *simple* transformations and *small-sized* transformed models. All transformations also work for infinite-state systems; however, it is only for finite models that their transformed models are guaranteed to be finite.

[1]This makes language-preserving transformations trivial for refinement settings with an empty set of finite models. We will only consider the refinement settings introduced in Chapter 2, all of which have a non-empty set of finite models.

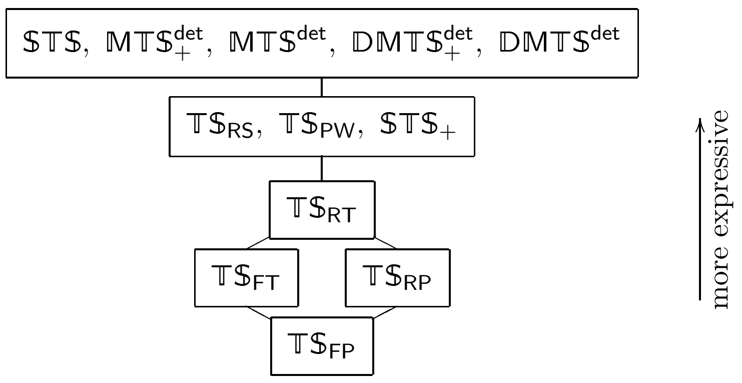

$\mathbb{STS}$, $\mathbb{MTS}^{\mathsf{det}}_+$, $\mathbb{MTS}^{\mathsf{det}}$, $\mathbb{DMTS}^{\mathsf{det}}_+$, $\mathbb{DMTS}^{\mathsf{det}}$
$\mathbb{TS}_{\mathsf{RS}}$, $\mathbb{TS}_{\mathsf{PW}}$, $\mathbb{STS}_+$
$\mathbb{TS}_{\mathsf{RT}}$
$\mathbb{TS}_{\mathsf{FT}}$
$\mathbb{TS}_{\mathsf{RP}}$
$\mathbb{TS}_{\mathsf{FP}}$
more expressive

Figure 3.2: Expressiveness hierarchy of refinement settings over deterministic transition systems.

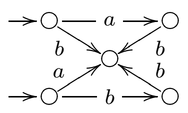


Figure 3.3: Transition system $M_{\mathrm{rs,rt}}$.

To begin with, the identity function is obviously a transformation from $\mathbb{STS}_+$ to $\mathbb{STS}$; from $\mathbb{MTS}^{\mathsf{det}}_+$ to $\mathbb{MTS}^{\mathsf{det}}$; and from $\mathbb{DMTS}^{\mathsf{det}}_+$ to $\mathbb{DMTS}^{\mathsf{det}}$. It follows directly from the definitions of the settings that the identy function is also a transformation from $\mathbb{TS}_{\mathsf{PW}}$ to $\mathbb{TS}_{\mathsf{RS}}$ and vice versa.

3.3.1 Trace inclusions

Due to the inductive definition of simulation, checking refinement in simulation-based settings only depends on what remains to be considered in the future, e.g. $M \overset{\text{def}}{=}$
$$\begin{tikzcd} \arrow[r] & \circ \arrow[r, "a"] \arrow[r, "b"'] & \circ \end{tikzcd}$$
is no implementation of $M_{\mathrm{rs,rt}}$ of Figure 3.3 in simulation-like approaches, because the refinement relation has to decide which one of the two initial states of $M_{\mathrm{rs,rt}}$ should be mapped to the initial state of M, and this will require in the implementation a succeeding b step, either after the initial a step or the initial b step. This is different in trace-based approaches, where at any time it is possible to go back in a trace and resolve nondeterminism differently, as long as the traces still coincide. Consequently, M *is* a refinement of $M_{\mathrm{rs,rt}}$ in trace-like settings.

The transformations from trace approaches thus have in common that they use power set states to collect all possible states reachable via common traces (i.e., ready traces, failure traces, ready pairs, failure pairs). We call these power set states *merged states* and its elements *elementary states*.

For ready traces, all states with the same sets of enabled labels are collected in a merged state. Given a collection of states $\widehat{S}$, $\Psi^t_=(\widehat{S})$, as defined formally in Transformation 3.4, describes the set of these merged states, only considering states from $\widehat{S}$ and excluding the empty set.

The corresponding notion of merged states for failure traces is $\Psi^t_\subseteq(\widehat{S})$ which, given $\widehat{S}$, describes the set of those merged states, for which there is a label set that is a super-set of all enabled

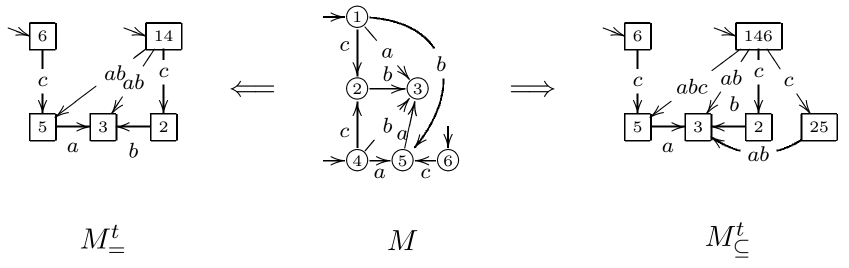


Figure 3.4: Examples of the transformations from $\mathbb{T}\$_{\mathsf{RT}}$ and $\mathbb{T}\$_{\mathsf{FT}}$. The first of these transforms the system in the middle (M) to the system on the left ($M^t_=$), the second one transforms the system in the middle (M) to the system on the right ($M^t_\subseteq$). Transitions having a set as label (denoted here shortly by a sequence of labels) indicate a set of transitions, one for each label. The numbers of merged state names in the transformed systems ($M^t_=$ and $M^t_\subseteq$) correspond to their elementary states.

labels in the elementary states. This corresponds to the coarser notion of observability in failure trace semantics.

Then, the transition relation is defined on these merged states in a straightforward fashion, for every step, say via a, taking into account all a-transitions of all elementary states. Figure 3.4 gives examples for both transformations. In fact, these are not only transformations to ready simulation, but also to further settings, including themselves.

Transformation 3.4. *For any transition system $M = (S, S^{\mathrm{i}}, \longrightarrow)$, we have*

(i) $\mathbb{T}\$_{\mathsf{RT}}(M) = \mathbb{T}\$_{\mathsf{RS}}(M^t_=) = \mathbb{T}\$_{\mathsf{RT}}(M^t_=)$*, and*

(ii) $\mathbb{T}\$_{\mathsf{FT}}(M) = \mathbb{T}\$_{\mathsf{RS}}(M^t_\subseteq) = \mathbb{T}\$_{\mathsf{RT}}(M^t_\subseteq) = \mathbb{T}\$_{\mathsf{FT}}(M^t_\subseteq)$*,*

with $M^t_\lhd = (\mathcal{P}(S), \Psi^t_\lhd(S^{\mathrm{i}}), \{(\ddot{S}, a, \ddot{S}') \mid \ddot{S}' \in \Psi^t_\lhd(\ddot{S}.\xrightarrow{a})\})$*, where* $\lhd\ \in \{=, \subseteq\}$ *and* $\Psi^t_\lhd(\widehat{S}) = \{\{s \in \widehat{S} \mid \mathbf{O}_{\longrightarrow}(s) \lhd L\} \mid L \subseteq \mathcal{L}\} \setminus \{\emptyset\}$.

Proof of Transformation 3.3. First we recall the following statement, taken from [vG01], which compares the refinement preorders on transition systems with respect to their coarseness (i.e., inclusion of sets of pairs in the relation).

Lemma 3.5 ([vG01]). *The coarseness hierarchy of the preorders of ready simulation, ready trace inclusion, failure trace inclusion, ready pair inclusion and failure pair inclusion is as illustrated in Figure 3.5.*

Now we prove the soundness of Transformation 3.4. It is easily seen that $M^t_=$ and $M^t_\subseteq$ are transition systems. From Lemma 3.5 we obtain $\mathbb{T}\$_{\mathsf{RS}}(M^t_=) \subseteq \mathbb{T}\$_{\mathsf{RT}}(M^t_=)$ and $\mathbb{T}\$_{\mathsf{RS}}(M^t_\subseteq) \subseteq \mathbb{T}\$_{\mathsf{RT}}(M^t_\subseteq) \subseteq \mathbb{T}\$_{\mathsf{FT}}(M^t_\subseteq)$. Therefore, in order to complete a circle of inclusions, the following four cases remain to be shown:

$\mathbb{T}\$_{\mathsf{RT}}(M) \subseteq \mathbb{T}\$_{\mathsf{RS}}(M^t_=)$: Let M^d be a deterministic transition system that is a ready trace implementation of M. Then $\{(s^d, \ddot{S}) \mid (s^d.\Theta^{M^d}_{\mathsf{RT}}) \subseteq (\ddot{S}.\Theta^M_{\mathsf{RT}}) \wedge \forall s \in \ddot{S} : \mathbf{O}_{\longrightarrow}(s) = \mathbf{O}_{\longrightarrow^d}(s^d)\}$, with Θ^M_{RT} defined as in Definition 2.6, yields a ready simulation that witnesses that M^d is a refinement of $M^t_=$. This is seen as follows:

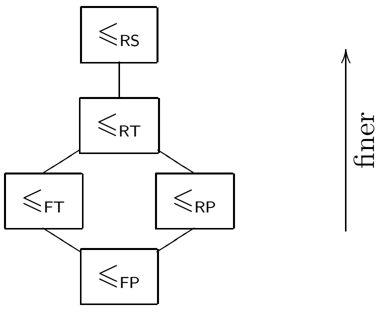

$\leqslant_{\mathsf{RS}}$
$\leqslant_{\mathsf{RT}}$
$\leqslant_{\mathsf{FT}}$
$\leqslant_{\mathsf{RP}}$
$\leqslant_{\mathsf{FP}}$
finer

Figure 3.5: Coarseness hierarchy of preorders on transition systems, taken from [vG01].

Suppose s_1^d and $\ddot{S}_1$ are related. Then, it follows that $\mathbf{O}_{M^t_=}(\ddot{S}_1) = \mathbf{O}_{\longrightarrow^d}(s_1^d)$. Now suppose $s_2^d \in (s_1^d.\xrightarrow{a}{}^d)$. Let $\ddot{S}_2 = \{s_2 \in (\ddot{S}_1.\xrightarrow{a}) \mid \mathbf{O}_{\longrightarrow}(s_2) = \mathbf{O}_{\longrightarrow^d}(s_2^d)\}$. From $(s_1^d.\Theta^{M^d}_{\mathsf{RT}}) \subseteq (\ddot{S}_1.\Theta^{M}_{\mathsf{RT}})$ we get (i) $\ddot{S}_2 \neq \emptyset$, which implies $\ddot{S}_1 \xrightarrow{a}{}^t_= \ddot{S}_2$, and (ii) $(s_2^d.\Theta^{M^d}_{\mathsf{RT}}) \subseteq (\ddot{S}_2.\Theta^{M}_{\mathsf{RT}})$, which implies that s_2^d and $\ddot{S}_2$ are related, as required. It is easily seen that the unique initial element of M^d and $\{s_2 \in S^{\mathrm{i}} \mid \mathbf{O}_{\longrightarrow}(s_2) = \mathbf{O}_{\longrightarrow^d}(s_2^d)\}$ are related.

$\mathbb{T}\$_{\mathsf{RT}}(M^t_=) \subseteq \mathbb{T}\$_{\mathsf{RT}}(M)$: Let M^d be a deterministic transition system that is a ready trace implementation of $M^t_=$. Let $\langle R_0 a_0 R_1 a_1 \ldots a_{n-1} R_n\rangle$ be a ready trace of M^d. Then this ready trace also exists in $M^t_=$, with $(\ddot{S}_i)_{i \leq n}$ being a witness trace. We can now obtain, by backward construction (starting in $\ddot{S}_n$, going towards $\ddot{S}_0$), a trace in M that has $\langle R_0 a_0 R_1 a_1 \ldots a_{n-1} R_n\rangle$ as ready trace, as required.

$\mathbb{T}\$_{\mathsf{FT}}(M) \subseteq \mathbb{T}\$_{\mathsf{RS}}(M^t_{\subseteq})$: Let M^d be a deterministic transition system that is a failure trace implementation of M. Then, $\{(s^d, \ddot{S}) \mid (s^d.\Theta^{M^d}_{\mathsf{FT}}) \subseteq (\ddot{S}.\Theta^{M}_{\mathsf{FT}}) \land \forall s \in \ddot{S} : \mathbf{O}_{\longrightarrow}(s) \subseteq \mathbf{O}_{\longrightarrow^d}(s^d)\}$, with Θ^{M}_{FT} defined as in Definition 2.8, yields a ready simulation that is a witness that M^d is a refinement of $M^t_{\subseteq}$. This is seen as follows:

Suppose s_1^d and $\ddot{S}_1$ are related. Then, it follows that $\mathbf{O}_{M^t_{\subseteq}}(\ddot{S}_1) = \mathbf{O}_{\longrightarrow^d}(s_1^d)$. Now suppose $s_2^d \in (s_1^d.\xrightarrow{a}{}^d)$. Let $\ddot{S}_2 = \{s_2 \in (\ddot{S}_1.\xrightarrow{a}) \mid \mathbf{O}_{\longrightarrow}(s_2) \subseteq \mathbf{O}_{\longrightarrow^d}(s_2^d)\}$. From $(s_1^d.\Theta^{M^d}_{\mathsf{FT}}) \subseteq (\ddot{S}_1.\Theta^{M}_{\mathsf{FT}})$ we get (i) $\ddot{S}_2 \neq \emptyset$, which implies $\ddot{S}_1 \xrightarrow{a}{}^t_{\subseteq} \ddot{S}_2$ and (ii) $(s_2^d.\Theta^{M^d}_{\mathsf{FT}}) \subseteq (\ddot{S}_2.\Theta^{M}_{\mathsf{FT}})$, which implies that s_2^d and $\ddot{S}_2$ are related, as required. It is easily seen that the unique initial element of M^d and $\{s_2 \in S^{\mathrm{i}} \mid \mathbf{O}_{\longrightarrow}(s_2) \subseteq \mathbf{O}_{\longrightarrow^d}(s_2^d)\}$ are related.

$\mathbb{T}\$_{\mathsf{FT}}(M^t_{\subseteq}) \subseteq \mathbb{T}\$_{\mathsf{FT}}(M)$: Let M^d be a deterministic transition system that is a failure trace implementation of $M^t_{\subseteq}$. Let $\langle F_0 a_0 F_1 a_1 \ldots a_{n-1} F_n\rangle$ be a failure trace of M^d. Then, this failure trace also exists in $M^t_{\subseteq}$, with $(\ddot{S}_i)_{i \leq n}$ being a witness trace. We can now obtain, by backward construction (starting in $\ddot{S}_n$, going towards $\ddot{S}_0$), a trace in M that has $\langle F_0 a_0 F_1 a_1 \ldots a_{n-1} F_n\rangle$ as failure trace, as required. □

Ready and failure pair semantics make coarser observations than their trace variants, because there is no intermediate ready or failure set information which is only provided for the last state. Thus, states are merged if they are reachable via an arbitrary common label sequence (regardless of intermediate ready or failure sets). However, we have to distinguish different ready or failure pairs in the encoding of merged states, and therefore these are now pairs of state sets and label sets.

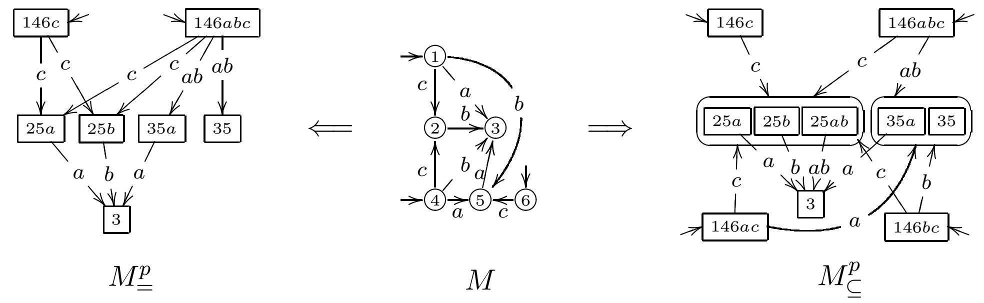


Figure 3.6: Examples of the transformations from $\mathbb{T}\$_{\mathsf{RP}}$ and $\mathbb{T}\$_{\mathsf{FP}}$. The first of these transforms the system in the middle (M) to the system on the left ($M^p_=$), the second one transforms the system in the middle (M) to the system on the right ($M^p_\subseteq$). The numbers and labels of the merged states in the transformed systems ($M^p_=$ and $M^p_\subseteq$) correspond to their pair of elementary states and label set. For the illustration of $M^p_\subseteq$, states that have the same targets are identified. A transition pointing to an oval indicates a set of transitions pointing to each element inside the oval.

$\Psi^p_=(\widehat{S})$, for ready pair inclusion, describes the merged states that are pairs $(\widehat{S}, \widehat{L})$ such that there is an elementary state in $\widehat{S}$ that corresponds to ready set $\widehat{L}$. $\Psi^p_\subseteq(\widehat{S})$ describes the corresponding merged states for failure pair inclusion; here, the enabled labels need to be a subset of $\widehat{L}$.

Figure 3.6 gives examples of both transformations.

Transformation 3.6. *For any transition system $M = (S, S^{\mathrm{i}}, \longrightarrow)$, we have*

(i) $\mathbb{T}\$_{\mathsf{RP}}(M) = \mathbb{T}\$_{\mathsf{RS}}(M^p_=) = \mathbb{T}\$_{\mathsf{RT}}(M^p_=) = \mathbb{T}\$_{\mathsf{RP}}(M^p_=)$, *and*

(ii) $\mathbb{T}\$_{\mathsf{FP}}(M) = \mathbb{T}\$_{\mathsf{RS}}(M^p_\subseteq) = \mathbb{T}\$_{\mathsf{RT}}(M^p_\subseteq) = \mathbb{T}\$_{\mathsf{FT}}(M^p_\subseteq) = \mathbb{T}\$_{\mathsf{RP}}(M^p_\subseteq) = \mathbb{T}\$_{\mathsf{FP}}(M^p_\subseteq)$,

with $M^p_\lhd = (\mathcal{P}(S) \times \mathcal{P}(\mathcal{L}), \Psi^p_\lhd(S^{\mathrm{i}}), \{((\ddot{S}, L), a, Z') \mid a \in L \wedge Z' \in \Psi^p_\lhd(\ddot{S}.\xrightarrow{a})\})$, *where* $\lhd \in \{=, \subseteq\}$ *and* $\Psi^p_\lhd(\widehat{S}) = \{(\widehat{S}, \widehat{L}) \mid \exists s \in \widehat{S} : \mathbf{O}_{\longrightarrow}(s) \lhd \widehat{L}\}$.

Proof. It is easy to see that $M^p_=$ and $M^p_\subseteq$ are transition systems. From Lemma 3.5 we get $\mathbb{T}\$_{\mathsf{RS}}(M^p_=) \subseteq \mathbb{T}\$_{\mathsf{RT}}(M^p_=) \subseteq \mathbb{T}\$_{\mathsf{RP}}(M^p_=)$ and $\mathbb{T}\$_{\mathsf{RS}}(M^p_\subseteq) \subseteq \mathbb{T}\$_{\mathsf{RT}}(M^p_\subseteq) \subseteq \mathbb{T}\$_{\mathsf{FT}}(M^p_\subseteq) \subseteq \mathbb{T}\$_{\mathsf{FP}}(M^p_\subseteq)$ and $\mathbb{T}\$_{\mathsf{RT}}(M^p_\subseteq) \subseteq \mathbb{T}\$_{\mathsf{RP}}(M^p_\subseteq) \subseteq \mathbb{T}\$_{\mathsf{FP}}(M^p_\subseteq)$. Therefore, in order to complete a circle of inclusions, the following four cases remain to be shown:

$\mathbb{T}\$_{\mathsf{RP}}(M) \subseteq \mathbb{T}\$_{\mathsf{RS}}(M^p_=)$: Let M^d be a deterministic transition system that is a ready pair implementation of M. Then, $\{(s^d, (\ddot{S}, \mathbf{O}_{\longrightarrow^d}(s^d))) \mid (s^d.\Theta^{M^d}_{\mathsf{RP}}) \subseteq (\ddot{S}.\Theta^{M}_{\mathsf{RP}}) \wedge \exists s \in \ddot{S} : \mathbf{O}_{\longrightarrow}(s) = \mathbf{O}_{\longrightarrow^d}(s^d)\}$, with Θ^M_{RP} defined as in Definition 2.5, yields a ready simulation that witnesses that M^d is a refinement of $M^p_=$. This is seen as follows:

Suppose s^d_1 and $(\ddot{S}_1, L_1)$ are related. Then, it follows that $\mathbf{O}_{M^p_=}(\ddot{S}_1, L_1) = \mathbf{O}_{\longrightarrow^d}(s^d_1)$. Now suppose $s^d_2 \in (s^d_2.\xrightarrow{a}{}^d)$. From $(s^d_1.\Theta^{M^d}_{\mathsf{RP}}) \subseteq (\ddot{S}_1.\Theta^M_{\mathsf{RP}})$ we get (i) $\exists s_2 \in (\ddot{S}_1.\xrightarrow{a}) : \mathbf{O}_{\longrightarrow}(s_2) = \mathbf{O}_{\longrightarrow^d}(s^d_2)$, which implies $(\ddot{S}_1, L_1) \xrightarrow{a}{}^p_= ((\ddot{S}_1.\xrightarrow{a}), \mathbf{O}_{\longrightarrow^d}(s^d_2))$ and (ii) $(s^d_2.\Theta^{M^d}_{\mathsf{RP}}) \subseteq ((\ddot{S}_1.\xrightarrow{a}).\Theta^M_{\mathsf{RP}})$, which implies that s^d_2 and $(\ddot{S}_2, \mathbf{O}_{\longrightarrow^d}(s^d_2))$ are related, as required. By similar arguments we obtain that for the unique initial element s^d_i of M^d we have that $(S^{\mathrm{i}}, \mathbf{O}_{\longrightarrow^d}(s^d_i))$ is an initial element of $M^p_=$, which is related to s^d_i.

$\mathbb{T}\$_{\mathsf{RP}}(M^p_{=}) \subseteq \mathbb{T}\$_{\mathsf{RP}}(M)$: Let M^d be a deterministic transition system that is a ready pair implementation of $M^p_{=}$. Let $\langle a_0a_1 \dots a_{n-1}R_n\rangle$ be a ready pair of M^d. Then, this ready pair also exists in $M^p_{=}$, with $(\ddot{S}_i, L_i)_{i \leq n}$ being a witness trace. We can now obtain, by backward construction (starting in $\ddot{S}_n$, going towards $\ddot{S}_0$), a trace in M that has $\langle a_0a_1 \dots a_{n-1}R_n\rangle$ as ready pair, as required.

$\mathbb{T}\$_{\mathsf{FP}}(M) \subseteq \mathbb{T}\$_{\mathsf{RS}}(M^p_{\subseteq})$: Let M^d be a deterministic transition system that is a failure pair implementation of M. Then, $\{(s^d, (\ddot{S}, \mathbf{O}_{\longrightarrow^d}(s^d))) \mid (s^d.\Theta^{M^d}_{\mathsf{FP}}) \subseteq (\ddot{S}.\Theta^{M}_{\mathsf{FP}})\}$, with Θ^{M}_{FP} defined as in Definition 2.7, yields a ready simulation that witnesses that M^d is a refinement of $M^p_{\subseteq}$. This is seen as follows:

Suppose s^d_1 and $(\ddot{S}_1, L_1)$ are related. Then, from $(s^d_1.\Theta^{M^d}_{\mathsf{FP}}) \subseteq (\ddot{S}_1.\Theta^{M}_{\mathsf{FP}})$ we get $\mathbf{O}_{M^p_{\subseteq}}(\ddot{S}_1, L_1) = \mathbf{O}_{\longrightarrow^d}(s^d_1)$. Now suppose $s^d_2 \in (s^d_2.\xrightarrow{a}{}^d)$. From $(s^d_1.\Theta^{M^d}_{\mathsf{RP}}) \subseteq (\ddot{S}_1.\Theta^{M}_{\mathsf{RP}})$ we get (i) $\exists s_2 \in (\ddot{S}_1.\xrightarrow{a}) : \mathbf{O}_{\longrightarrow}(s_2) \subseteq \mathbf{O}_{\longrightarrow^d}(s^d_2)$, which implies $(\ddot{S}_1, L_1) \xrightarrow{a}{}^p_{\subseteq} ((\ddot{S}_1.\xrightarrow{a}), \mathbf{O}_{\longrightarrow^d}(s^d_2))$ and (ii) $(s^d_2.\Theta^{M^d}_{\mathsf{FP}}) \subseteq ((\ddot{S}_1.\xrightarrow{a}).\Theta^{M}_{\mathsf{FP}})$, which implies that s^d_2 and $(\ddot{S}_2, \mathbf{O}_{\longrightarrow^d}(s^d_2))$ are related, as required. By similar arguments we obtain that for the unique initial element s^d_i of M^d we have that $(S^{\mathrm{i}}, \mathbf{O}_{\longrightarrow^d}(s^d_i))$ is an initial element of $M^p_{\subseteq}$, which is related to s^d_i.

$\mathbb{T}\$_{\mathsf{RP}}(M^p_{\subseteq}) \subseteq \mathbb{T}\$_{\mathsf{RP}}(M)$: Let M^d be a deterministic transition system that is a ready pair implementation of $M^p_{\subseteq}$. Let $\langle a_0a_1 \dots a_{n-1}F_n\rangle$ be a failure pair of M^d. Then, this failure pair also exists in $M^p_{\subseteq}$, with $(\ddot{S}_i, L_i)_{i \leq n}$ being a witness trace. We can now obtain, by backward construction (starting in $\ddot{S}_n$, going towards $\ddot{S}_0$), a trace in M that has $\langle a_0a_1 \dots a_{n-1}F_n\rangle$ as ready pair, as required. $\square$

The increase of expressiveness of these settings is illustrated in Figure 3.2 and argued as follows. The failure approach cannot express an exclusive alternative between two labels. For example, no transition system with respect to failure pair inclusion or failure trace inclusion can have $\to\circ\xrightarrow{a}\circ$ and $\to\circ\xrightarrow{b}\circ$ as implementations, without also having $\begin{tikzcd}[row sep=tiny] & & \circ \\ {} \arrow[r] & \circ \arrow[ur, "a"] \arrow[dr, "b"'] & \\ & & \circ \end{tikzcd}$ as implementation. However, the ready approach can express such an exclusive alternative, as shown by $M_{\mathrm{r,ft}}$ in Figure 3.7.

In pair approaches, behavior can only be described up to alternatives having the same label path histories, whereas a trace approach can also distinguish alternatives that have the same label path history but different next-step possibilities (up to failure or ready interpretation). For example, no transition system with respect to failure pair inclusion or ready pair inclusion can have $\begin{tikzcd}[row sep=tiny] & & \circ & \\ {} \arrow[r] & \circ \arrow[ur, "a"] \arrow[dr, "b"'] & & \\ & & \circ \arrow[r, "b"] & \circ \end{tikzcd}$ and $\to\circ\xrightarrow{b}\circ$ as implementations, without also having $\to\circ\xrightarrow{b}\circ\xrightarrow{b}\circ$ as implementation. However, $M_{\mathrm{ft,r}}$ of Figure 3.7 defines such a language via failure trace inclusion and ready trace inclusion.

Ready simulation increases the expressiveness even more by distinguishing also alternatives with the same label path history and next-step possibilities, but different future behaviors in the past. For example, no transition system with respect to a trace approach can have $\begin{tikzcd}[row sep=tiny] & & \circ \arrow[r, "b"] & \circ \\ {} \arrow[r] & \circ \arrow[ur, "a"] \arrow[dr, "b"'] & & \\ & & \circ & \end{tikzcd}$ and $\begin{tikzcd}[row sep=tiny] & & \circ & \\ {} \arrow[r] & \circ \arrow[ur, "a"] \arrow[dr, "b"'] & & \\ & & \circ \arrow[r, "b"] & \circ \end{tikzcd}$ as implementations, without also having $\begin{tikzcd}[row sep=tiny] & & \circ \\ {} \arrow[r] & \circ \arrow[ur, "a"] \arrow[dr, "b"'] & \\ & & \circ \end{tikzcd}$. However, $M_{\mathrm{rs,rt}}$ of Figure 3.7 defines such a language via ready simulation. The following lemma summarizes the above results, from which the 'strictly more' expressiveness results for the transition system-based settings are derived via transitivity arguments.

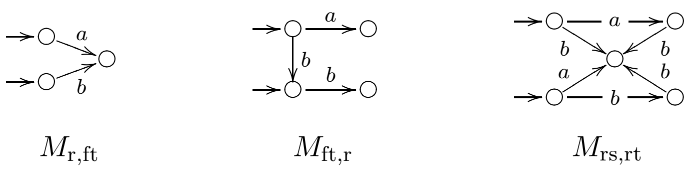


Figure 3.7: Transition systems illustrating increases of expressiveness.

Lemma 3.7. *For $M_{\mathrm{r,ft}}$, $M_{\mathrm{ft,r}}$, $M_{\mathrm{rs,rt}}$ in Figure 3.7 and arbitrary M, we have:*

$$\mathbb{T}\$_{\mathsf{RP}}(M_{\mathrm{r,ft}}) \neq \mathbb{T}\$_{\mathsf{FT}}(M),\ \mathbb{T}\$_{\mathsf{FT}}(M_{\mathrm{ft,r}}) \neq \mathbb{T}\$_{\mathsf{RP}}(M) \text{ and } \mathbb{T}\$_{\mathsf{RS}}(M_{\mathrm{rs,rt}}) \neq \mathbb{T}\$_{\mathsf{RT}}(M).$$

Proof. The proofs for all these results have already been sketched before the Lemma. Here, we work out the details for the first case, $\mathbb{T}\$_{\mathsf{RP}}(M_{\mathrm{r,ft}}) \neq \mathbb{T}\$_{\mathsf{FT}}(M)$, for arbitrary M. Assume there is some M with $\mathbb{T}\$_{\mathsf{RP}}(M_{\mathrm{r,ft}}) = \mathbb{T}\$_{\mathsf{FT}}(M)$. Then, M has failure trace implementations $\rightarrow\circ\xrightarrow{a}\circ$ and $\rightarrow\circ\xrightarrow{b}\circ$, because $M_{\mathrm{r,ft}}$ has these ready pair implementations. Thus, M has (at least) the following failure traces:

$$\{\langle\emptyset\rangle, \langle\{a\}\rangle, \langle\{b\}\rangle, \langle\emptyset aX\rangle, \langle\emptyset bX\rangle, \langle\{a\}bX\rangle, \langle\{b\}aX\rangle \mid X \subseteq \{a,b\}\}.$$

The set of failure traces of $M' \overset{\mathrm{def}}{=} \rightarrow\circ\begin{smallmatrix}\xrightarrow{a}\circ\\ \xrightarrow[b]{}\circ\end{smallmatrix}$ is a subset of this, namely

$$\{\langle\emptyset\rangle, \langle\emptyset aX\rangle, \langle\emptyset bX\rangle \mid X \subseteq \{a,b\}\},$$

and, consequently, M' is a failure trace implementation of M, too. This, however, is a contradiction because $\mathbb{T}\$_{\mathsf{RP}}(M_{\mathrm{r,ft}}) = \mathbb{T}\$_{\mathsf{FT}}(M)$ and M' is *not* a ready pair implementation of $M_{\mathrm{r,ft}}$, since M' has ready pair $\langle\{a,b\}\rangle$ which $M_{\mathrm{r,ft}}$ has not. □

3.3.2 Ready simulation and STS_+

$\mathbb{T}\$_{\mathsf{RS}}$ is transformed to $\$\mathbb{T}\$_+$ by setting the existence predicate to the set of labels for which an outgoing transition exists.

Transformation 3.8. *For any transition system $M = (S, S^{\mathrm{i}}, \longrightarrow)$,*

$$\mathbb{T}\$_{\mathsf{RS}}(M) = \$\mathbb{T}\$_+((S, S^{\mathrm{i}}, \longrightarrow, \mathbf{O}_{\longrightarrow})).$$

Proof. Is straightforwardly checked. □

For transforming $\$\mathbb{T}\$_+$ to $\mathbb{T}\$_{\mathsf{RS}}$, every state s is combined with a ready set $L \subseteq \mathcal{L}$, indicating that exactly these labels may *not* be removed in refinements. The incoming transitions are determined by the incoming ones of s. Figure 3.8 presents a simple example.

Transformation 3.9. *For any STS_+ $M = (S, S, \longrightarrow, e)$, we have*

$$\$\mathbb{T}\$_+(M) = \mathbb{T}\$_{\mathsf{RS}}((S', S' \cap (S^{\mathrm{i}} \times \mathcal{P}(\mathcal{L})), \longrightarrow'))$$

with $S' = \{(s, L) \mid e(s) \subseteq L \subseteq \mathbf{O}_{\longrightarrow}(s)\}$, $\longrightarrow' = \{((s,L), a, (s', L')) \mid s \xrightarrow{a} s' \wedge a \in L\}$.

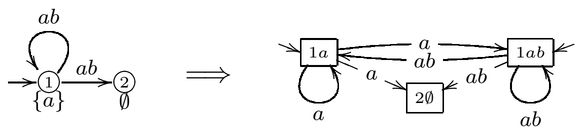


Figure 3.8: Example of the transformation from $\mathbb{STS}_+$ to $\mathbb{TS}_{\mathsf{RS}}$. The numbers (resp. labels) of the state names in the right picture correspond to the state (resp. label) subset encoding of the transformation.

Proof. It can immediately be seen that $(S', S' \cap (S^{\mathrm{i}} \times \mathcal{P}(\mathcal{L})), \longrightarrow')$ is a transition system.

Let M^d be a deterministic transition system that STS-refines M, with R being a suitable STS-refinement relation. Then, it can be checked that $\{(s^d, (s, \mathbf{O}_{\longrightarrow^d}(s^d))) \mid (s^d, s) \in R\}$ witnesses that M^d ready simulation-refines $(S', S' \cap (S^{\mathrm{i}} \times \mathcal{P}(\mathcal{L})), \longrightarrow')$.

For the other direction, let M^d be a deterministic transition system that ready simulation-refines $(S', S' \cap (S^{\mathrm{i}} \times \mathcal{P}(\mathcal{L})), \longrightarrow')$, with R' being a suitable ready simulation relation. Then, it can be checked that $\{(s^d, s) \mid \exists L : (s^d, (s, L)) \in R'\}$ witnesses that M^d STS-refines M. □

$\mathbb{STS}$ is indeed strictly more expressive than $\mathbb{TS}_{\mathsf{RS}}$, because the latter does not allow for the specification of the empty language.

Lemma 3.10. *For the* STS $\rightarrow\underset{\{a\}}{\bigcirc}$ *and an arbitrary transition system M, we have*

$$\mathbb{STS}(\rightarrow\underset{\{a\}}{\bigcirc}) \neq \mathbb{TS}_{\mathsf{RS}}(M).$$

Proof. It is easily checked that every transition system has an implementation with respect to ready simulation: take the same underlying state space and remove, as long as possible, transitions for which another transition having the same source and label remain existent. □

If we were to allow the initial set of a transition system to be empty and, therefore, not to have any deterministic transition system as refinement, we obtain, by the following algorithm, that $\mathbb{STS}$ and $\mathbb{STS}_+$ are equally expressive. Hence, the empty set is the only language that increases the expressive power of $\mathbb{STS}$ and any other equally expressive refinement setting.

Proposition 3.11. *An* STS *can be linearly transformed into an equivalent must-saturated* STS_+ *–adapted such that an empty state set is allowed – by successively removing those states s and their in- and outgoing transitions, for which $e(s) \nsubseteq \mathbf{O}_{\longrightarrow}(s)$.*

Proof. The language remains unaffected by such a transformation since the removed states can never be related to a state of an implementation. A linear algorithm works as follows: for every state and label, a count variable is used to store the number of outgoing transitions having this label. These variables are initialized with -1 indicating that the state has not been examined yet. Then, traverse the state space. Determine the corresponding count variables for those labels that have to be present (otherwise, the value remains unmodified). If one of the count variables becomes 0, remove this state and its incoming transitions. When removing transitions, decrement the counters of their sources by 1. If this leads to a counter value of 0, also schedule this state to be removed. Obviously, this algorithm is sound and linear. □

3.3.3 Remaining settings

$\mathbb{STS}$ is easily transformed to $\mathbb{MTS}^{\mathsf{det}}$: For every state, model its associated existence predicate e by introducing must transitions, labeled by elements from e and leading to a special state s_{all} that is refined by each implementation state.

Transformation 3.12. *For any* STS $M = (S, S^{\mathrm{i}}, \longrightarrow, e)$ *and fresh* $s_{\text{all}} \notin S$*, we have*

$$\mathbb{STS}(M) = \mathbb{MTS}^{\mathsf{det}}((S \cup \{s_{\text{all}}\}, S^{\mathrm{i}}, \longrightarrow \cup (\{s_{\text{all}}\} \times \mathcal{L} \times \{s_{\text{all}}\}), \bigcup_{s \in S} \{s\} \times e(s) \times \{s_{\text{all}}\})).$$

Proof. It can immediately be seen that $M' \overset{\text{def}}{=} (S \cup \{s_{\text{all}}\}, S^{\mathrm{i}}, \longrightarrow \cup (\{s_{\text{all}}\} \times \mathcal{L} \times \{s_{\text{all}}\}), \bigcup_{s \in S} \{s\} \times e(s) \times \{s_{\text{all}}\})$ is an MTS.

Let M^d be a deterministic transition system that STS-refines M, with R being a suitable STS-refinement relation. Then, it is easily checked that $R \cup \{(s, s_{\text{all}}) \mid s \in S\}$ witnesses that M^d MTS-refines M'.

For the other direction, let M^d be a deterministic transition system that MTS-refines M', with R' being a suitable MTS-refinement relation. Then, it is easily checked that $R' \setminus \{(s, s_{\text{all}}) \mid s \in S\}$ witnesses that M^d STS-refines M. □

$\mathbb{MTS}^{\mathsf{det}}$ and $\mathbb{MTS}_{+}^{\mathsf{det}}$ are transformed to $\mathbb{DMTS}^{\mathsf{det}}$ and $\mathbb{DMTS}_{+}^{\mathsf{det}}$, respectively, by turning every must transition pointing to a state s into a must hypertransition pointing to the singleton set $\{s\}$.

Transformation 3.13. *(i) For any* MTS $M = (S, S^{\mathrm{i}}, \dashrightarrow, \longrightarrow)$*, we have*

$$\mathbb{MTS}^{\mathsf{det}}(M) = \mathbb{DMTS}^{\mathsf{det}}((S, S^{\mathrm{i}}, \dashrightarrow, \{(s, a, \{s'\}) \mid s \xrightarrow{a} s'\})).$$

(ii) For any MTS$_{+}$ $M = (S, S^{\mathrm{i}}, \dashrightarrow, \longrightarrow)$*, we have*

$$\mathbb{MTS}_{+}^{\mathsf{det}}(M) = \mathbb{DMTS}_{+}^{\mathsf{det}}((S, S^{\mathrm{i}}, \dashrightarrow, \{(s, a, \{s'\}) \mid s \xrightarrow{a} s'\})).$$

Proof. For the proof of statement (i), it can immediately be seen that $M' \overset{\text{def}}{=} (S, S^{\mathrm{i}}, \dashrightarrow, \{(s, a, \{s'\}) \mid s \xrightarrow{a} s'\})$ is an MTS.

Let M^d be a deterministic transition system that MTS-refines M, with R being a suitable MTS-refinement relation. Then, it is easily checked that the same relation R witnesses that M^d DMTS-refines M'.

For the other direction, let M^d be a deterministic transition system that DMTS-refines M', with R' being a suitable DMTS-refinement relation. Then, it is easily checked that the same relation R' witnesses that M^d MTS-refines M.

Statement (ii) is proven analogously. □

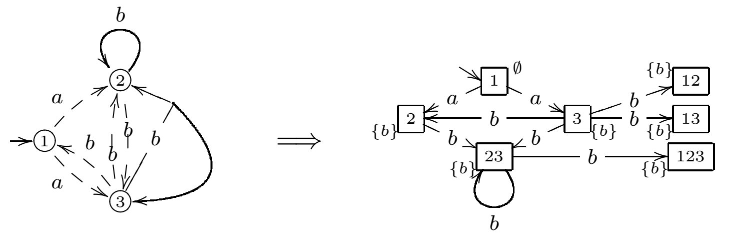


Figure 3.9: Example of the transformation from $\mathbb{DMTS}^{\mathsf{det}}$ to $\mathbb{STS}$. Numbers in merged states in the right system correspond to their elementary states; e.g., the self loop of state $\{2,3\}$ is obtained by choosing g and h such that $g(2) = 3$, $g(3) = 2$, $h(\{2\}) = 2$, and $h(\{2,3\}) = 2$.

We proceed with the transformation from $\mathbb{DMTS}^{\mathsf{det}}$ to $\mathbb{STS}$. The states in the transformed system are power set states, with the intuition that a related implementation state has to be related to all elementary states of a merged state. Transitions from a merged state lead to those merged states that consist of a combination of targets of hypertransitions from contained elementary states, together with one may-target for each elementary state. In the definition of these successor sets $C^a_{\ddot{S}}$, we use choice functions $h : \mathcal{P}(S) \to S$ for the selection of an element from a must hypertransition target, and $g : S \to S$ for the selection of a may transition target. The existence predicate holds for a at a merged state iff there is a must hypertransition with label a and leaving an elementary state of the merged state. Figure 3.9 shows an example of this transformation.

Transformation 3.14. *For any* DMTS $M = (S, S^{\mathrm{i}}, \dashrightarrow, \longrightarrow)$*,*

$$\mathbb{DMTS}^{\mathsf{det}}(M) = \mathbb{STS}((\mathcal{P}(S), \{\{s^0\} \mid s^0 \in S^{\mathrm{i}}\}, \bigcup_{\ddot{S} \subseteq S, a \in \mathcal{L}} \{\ddot{S}\} \times \{a\} \times C^a_{\ddot{S}}, \mathbf{O}_{\longrightarrow}))$$

with $C^a_{\ddot{S}} = \{g(\ddot{S}) \cup h(\ddot{S}.\xrightarrow{a}) \mid \forall s \in \ddot{S} : s \overset{a}{\dashrightarrow} g(s) \wedge \forall \ddot{S}' \in (\ddot{S}.\xrightarrow{a}) : h(\ddot{S}') \in \ddot{S}'\}$*.*

Proof. It can immediately be seen that $(\mathcal{P}(S), \{\{s^0\} \mid s^0 \in S^{\mathrm{i}}\}, \bigcup_{\ddot{S},a} \{\ddot{S}\} \times \{a\} \times C^a_{\ddot{S}}, \mathbf{O}_{\longrightarrow})$ is an STS.

Let M^d be a deterministic transition system that DMTS-refines M, with R being a suitable DMTS-refinement relation. Then, it can be checked that $\{(s^d, \ddot{S}) \in S^d \times \mathcal{P}(S) \mid \forall s \in \ddot{S} : (s^d, s) \in R\}$ witnesses that M^d STS-refines $(\mathcal{P}(S), \{\{s^0\} \mid s^0 \in S^{\mathrm{i}}\}, \bigcup_{\ddot{S},a} \{\ddot{S}\} \times \{a\} \times C^a_{\ddot{S}}, \mathbf{O}_{\longrightarrow})$.

For the other direction, let M^d be a deterministic transition system that STS-refines $(\mathcal{P}(S), \{\{s^0\} \mid s^0 \in S^{\mathrm{i}}\}, \bigcup_{\ddot{S},a} \{\ddot{S}\} \times \{a\} \times C^a_{\ddot{S}}, \mathbf{O}_{\longrightarrow})$, with R' being a suitable STS-refinement relation. Then, it can be checked that $\{(s^d, s) \mid \exists \ddot{S} \in (s^d.R') : s \in \ddot{S}\}$ witnesses that M^d DMTS-refines M. $\square$

Finally, we present the transformation from $\mathbb{DMTS}^{\mathsf{det}}$ to $\mathbb{MTS}^{\mathsf{det}}_{+}$ with complexity $\mathcal{O}((2^{|S|})^{|\mathcal{L}|})$ and, by restriction to $\mathbb{MTS}^{\mathsf{det}}$, obtain a transformation from $\mathbb{MTS}^{\mathsf{det}}$ into $\mathbb{MTS}^{\mathsf{det}}_{+}$ with complexity $\mathcal{O}(|S|^{|\mathcal{L}|})$.

A first observation is that $\mathbb{MTS}^{\mathsf{det}}_{+}$ can represent conjunctive behavior, i.e. it is possible to require the conjunction of several follow-up behaviors. This can be modeled by several must transitions, that all have to be implemented by a deterministic transition system. However, a state in $\mathbb{MTS}_{+}^{\mathsf{det}}$ cannot enforce the existence of a label a and, at the same time, model disjunctive behavior after the execution of a (via nondeterminism) since, as soon as an outgoing must transition (with implicit may transition) exists, all further outgoing may transitions with the same label are redundant: in deterministic refinements, the unique transition of the implementation already has to match with the may transition corresponding to the must transition. The solution is to distribute the needed requirements to multiple states, where one state $(s, -)$ enforces action existence, and several further states $(s, \ddot{S})$, with $\ddot{S} \subseteq S$, encode the nondeterministic behavior. Must transitions to all these states make sure that each of them is related to a single implementation state, which therefore has to meet all of the requirements. These must transitions originate from another kind of state $(s, f) \in S \times \mathrm{F}_s$, which encodes a complete resolution of the next-step nondeterminism in the $\mathbb{DMTS}^{\mathsf{det}}$-system. Here, F_s is a set of functions that collect, per label a, a set from $C^a_{\{s\}}$, i.e. an element from every must hypertransition target together with one may transition target. The resulting collection contains those $\mathbb{DMTS}^{\mathsf{det}}$-states to which the successor of a related implementation state must be related. To be precise, also no element can be collected if no must transition is present ($a \notin \mathbf{O}_{\longrightarrow}(s) \wedge f(a) = \emptyset$). For contradictory states, F_s is empty. A state (s, f) points, via a-labeled must transitions, to every element of $f(a) \times (\{-\} \cup \mathcal{P}(S))$. A state $(s, -)$ encodes the labels necessary in s via must transitions to state s_{all}, which is refined by any implementation state. A state $(s, \ddot{S})$ is used to model the nondeterministic behavior of (i) the must hypertransition target $\ddot{S}$, or (ii) the may transitions if $\ddot{S} = (\{s\}.\xrightarrow{a})$. This is achieved by outgoing may transitions to every element s' of $\ddot{S}$, combined with any value of $\mathrm{F}_{s'}$. For technical reasons, $(s, \ddot{S})$ points to s_{all} if $\ddot{S}$ does not correspond to a must hypertransition target or to the may transition targets.

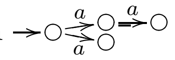
Figure 3.10 shows an example of this transformation.

Transformation 3.15. *For any* DMTS $M = (S, S^{\mathrm{i}}, \dashrightarrow, \longrightarrow)$, *we have*

$$\mathbb{DMTS}^{\mathsf{det}}(M) = \mathbb{MTS}_{+}^{\mathsf{det}}((S', S^{\mathrm{i}\prime}, \dashrightarrow', \longrightarrow'))$$

with $C^a_{\{s\}}$ *defined as in Transformation 3.14 and*

$$\begin{aligned}
\mathrm{F}_s &= \{f : \mathcal{L} \to \mathcal{P}(S) \mid \forall a \in \mathcal{L} : f(a) \in C^a_{\{s\}} \vee (a \notin \mathbf{O}_{\longrightarrow}(s) \wedge f(a) = \emptyset)\} \\
S' &= \{s_{\mathrm{all}}\} \cup \{(s, x) \mid s \in S \wedge x \in \mathrm{F}_s \cup \{-\} \cup \mathcal{P}(S)\} \\
S^{\mathrm{i}\prime} &= \{(s, x) \mid s \in S^{\mathrm{i}} \wedge x \in \mathrm{F}_s\} \qquad\qquad W^a_s = (s.\xrightarrow{a}) \cup \{(s.\overset{a}{\dashrightarrow})\} \\
\dashrightarrow' &= \longrightarrow' \cup (\{s_{\mathrm{all}}\} \times \mathcal{L} \times \{s_{\mathrm{all}}\}) \cup \{((s, x), a, s_{\mathrm{all}}) \mid x \in \{-\} \cup \mathcal{P}(S) \setminus W^a_s\} \cup \\
&\quad\ \{((s, \ddot{S}), a, (s', f')) \mid s' \in \ddot{S} \wedge \ddot{S} \in W^a_s \wedge f' \in \mathrm{F}_{s'}\} \\
\longrightarrow' &= \{((s, f), a, (s', x')) \mid f \in \mathrm{F}_s \wedge s' \in f(a) \wedge x' \in \{-\} \cup \mathcal{P}(S)\} \cup \\
&\quad\ \{((s, -), a, s_{\mathrm{all}}) \mid a \in \mathbf{O}_{\longrightarrow}(s)\}
\end{aligned}$$

In the transformation, we allow the initial state set to be empty. This, however, does not affect expressiveness, since, e.g., the modal transition system  also describes the empty language.

Proof of Transformation 3.15. It can immediately be seen that $(S', S^{\mathrm{i}\prime}, \dashrightarrow', \longrightarrow')$ is a modal transition system.

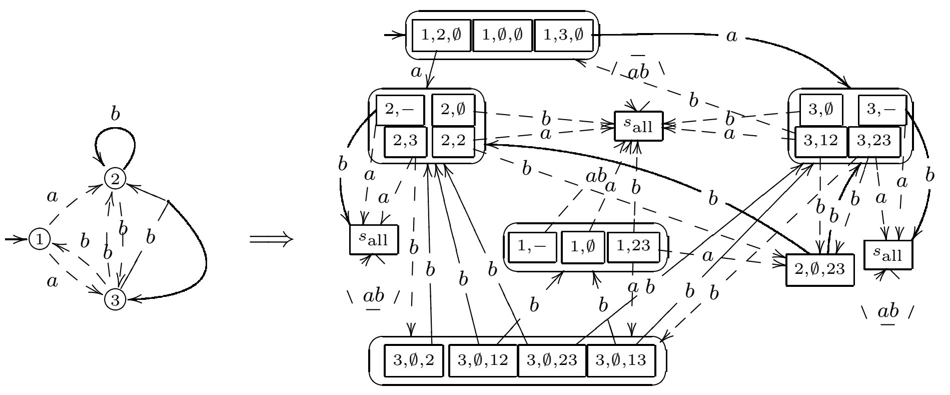


Figure 3.10: Example of the transformation from $\mathbb{DMTS}^{\mathsf{det}}$ to $\mathbb{MTS}_{+}^{\mathsf{det}}$. On the right, the may transitions that are implied by must transitions are omitted, and the symbols of the state names correspond to the encoding of the transformation: (i) pairs with second element "–" correspond to the states that encode the existent labels; (ii) the remaining pairs, which have a subset as second element, encode the may transition targets and the must hypertransitions (states $(s, \ddot{S})$ are omitted if $\ddot{S} \notin W_s^a \cup W_s^b$, since they do not influence refinement); (iii) a triple corresponds to states that have a complete resolution, where the second (resp., third) component encodes the image of a (resp., b). To improve readability, several copies of state s_{all} are used.

Let M^d be a deterministic transition system that DMTS-refines M, with R being a suitable DMTS-refinement relation. Then, it can be checked that $\{(s^d, (s, x)) \in S^d \times S' \mid (s^d, s) \in R \wedge (x \in \mathrm{F}_s \Rightarrow \forall a : ((s^d. \overset{a}{\dashrightarrow}{}^{d}).R) \supseteq x(a) \wedge (a \in \mathbf{O}_{\dashrightarrow^d}(s^d) \Rightarrow f(a) \neq \emptyset))\} \cup \{(s^d, s_{\text{all}}) \mid s^d \in S^d\}$ witnesses that M^d MTS-refines $(S', S^{\mathrm{i}\prime}, \dashrightarrow', \longrightarrow')$.

For the other direction, let M^d be a deterministic transition system that MTS-refines $(S', S^{\mathrm{i}\prime}, \dashrightarrow', \longrightarrow')$, with R' being the maximal MTS-refinement relation that witnesses this refinement. The maximal MTS-refinement is obtained by making the union of all refinement relations that are a witness. Then, it can be checked that $\{(s^d, s) \mid (\exists f \in \mathrm{F}_s : (s^d, (s, f)) \in R') \vee (\forall x \in \{-\} \cup \mathcal{P}(S) : (s^d, (s, x)) \in R')\}$ witnesses that M^d DMTS-refines M. ☐

This concludes our presentation of transformations, since all the remaining transformations can be obtained by composition, resulting in all of the cases in quite competitive (efficient) equivalent models.

Theorem 3.16. *Our transformations yield the expressiveness hierarchy of refinement settings that is depicted in Figure 3.2.*

Proof. This is an immediate consequence of (i) the fact that the identity function yields transformations from $\mathbb{STS}_{+}$ to $\mathbb{STS}$; from $\mathbb{MTS}_{+}^{\mathsf{det}}$ to $\mathbb{MTS}^{\mathsf{det}}$; from $\mathbb{DMTS}_{+}^{\mathsf{det}}$ to $\mathbb{DMTS}^{\mathsf{det}}$; from $\mathbb{TS}_{\mathsf{PW}}$ to $\mathbb{TS}_{\mathsf{RS}}$; and from $\mathbb{TS}_{\mathsf{RS}}$ to $\mathbb{TS}_{\mathsf{PW}}$, (ii) the soundness of all presented transformations and (iii) Lemmas 3.7 and 3.10. ☐

3.4 Related results

3.4.1 Deciding trace inclusions

The following proposition shows how our transformations and the efficient decision procedures for simulation-like preorders [CS01] can be used to decide the various inclusion problems. However, in general such derived algorithms would have limited practical relevance since deciding trace-like preorders is known to be PSPACE-complete [SHRS96], but in some practical cases the complexity of the decision procedure is certainly much lower.

Proposition 3.17. *M ready trace-refines (failure trace-, ready pair-, failure pair-refines) $\widetilde{M}$ iff $M^{\underline{t}}_{=}$ ($M^{\underline{t}}_{\subseteq}$, $M^{\underline{p}}_{=}$, $M^{\underline{p}}_{\subseteq}$) ready simulation-refines $\widetilde{M}^{\underline{t}}_{=}$ ($\widetilde{M}^{\underline{t}}_{\subseteq}$, $\widetilde{M}^{\underline{p}}_{=}$, $\widetilde{M}^{\underline{p}}_{\subseteq}$, respectively).*

Proof. The statement can be obtained as a consequence of the two following lemmas. The first lemma states that the four trace inclusion notions coincide with the corresponding "thorough refinement" notions, where thorough refinement is defined by set inclusion of implementation sets. This property does not hold in general for ready simulation, but it holds for the images of our transformations. Therefore our Transformations 3.4 and 3.6 not only preserve the sets of implementations, but also the sets of all nondeterministic refinements.

Lemma 3.18. *Let $\mathrm{x} \in \{\mathsf{RT}, \mathsf{FT}, \mathsf{RP}, \mathsf{FP}\}$. Then, we have that x-inclusion between transition systems coincides with "thorough refinement with respect to x-inclusion", i.e. for any two transition systems M_1, M_2 we have $(S^{\mathrm{i}}_1.\Theta^{M_1}_{\mathrm{x}}) \subseteq (S^{\mathrm{i}}_2.\Theta^{M_2}_{\mathrm{x}})$ iff $\mathbb{T}\$_{\mathrm{x}}(M_1) \subseteq \mathbb{T}\$_{\mathrm{x}}(M_2)$, where Θ^{M}_{x} is defined as in the definition of the refinement setting $\mathbb{T}\$_{\mathrm{x}}$.*

Proof. $\Rightarrow$: Let $M_d \in \mathbb{T}\$_{\mathrm{x}}(M_1)$. Then, $(S^{\mathrm{i}}_d.\Theta^{M_d}_{\mathrm{x}}) \subseteq (S^{\mathrm{i}}_1.\Theta^{M_1}_{\mathrm{x}})$. From this fact, taking into account that $(S^{\mathrm{i}}_1.\Theta^{M_1}_{\mathrm{x}}) \subseteq (S^{\mathrm{i}}_2.\Theta^{M_2}_{\mathrm{x}})$, we get $(S^{\mathrm{i}}_d.\Theta^{M_d}_{\mathrm{x}}) \subseteq (S^{\mathrm{i}}_1.\Theta^{M_1}_{\mathrm{x}}) \subseteq (S^{\mathrm{i}}_2.\Theta^{M_2}_{\mathrm{x}})$. Hence, $M_d \in \mathbb{T}\$_{\mathrm{x}}(M_2)$, as required.

$\Leftarrow$: Let $\vartheta \in (S^{\mathrm{i}}_1.\Theta^{M_1}_{\mathrm{x}})$ and $\langle s_0 s_1 s_2 \ldots \rangle$ be a sequence of states witnessing ϑ. Then, we apply the following construction: take any deterministic transition system M'_d obtained from M_1 by removing a collection of transitions until M'_d becomes deterministic, but no outgoing label from any state is lost. This deterministic transition system is extended by the fresh states $\mathbb{N}$. Then, the unique initial state is set to 0 and state i points via the i^{th} label of ϑ to $i+1$, and by any label a different from the i^{th} label of ϑ to some a-target of s_i (if one exists; otherwise, no target is determined). It is easily checked that the so obtained transition system, M_d, is deterministic, $M_d \in \mathbb{T}\$_{\mathrm{x}}(M_1)$, and $\vartheta \in (S^{\mathrm{i}}_d.\Theta^{M_d}_{\mathrm{x}})$. Therefore, we get $M_d \in \mathbb{T}\$_{\mathrm{x}}(M_2)$, which is equivalent to $(S^{\mathrm{i}}_d.\Theta^{M_d}_{\mathrm{x}}) \subseteq (S^{\mathrm{i}}_2.\Theta^{M_2}_{\mathrm{x}})$. Thus, $\vartheta \in (S^{\mathrm{i}}_2.\Theta^{M_2}_{\mathrm{x}})$, as required. $\Diamond$

Lemma 3.19. *After applying any of the transformations presented in Transformations 3.4 or 3.6, ready simulation coincides with possible worlds-refinement*[2]*, i.e. ${M_1}^{x}_{\lhd} \leqslant_{\mathsf{RS}} {M_2}^{x}_{\lhd}$ iff ${M_1}^{x}_{\lhd} \leqslant_{\mathsf{PW}} {M_2}^{x}_{\lhd}$, where $x \in \{t, p\}$ and $\lhd \in \{=, \subseteq\}$.*

Proof. It follows directly from the transitivity of ready-simulation refinement that $M_1 \leqslant_{\mathsf{RS}} M_2$ implies $\mathbb{T}\$_{\mathsf{RS}}(M_1) \subseteq \mathbb{T}\$_{\mathsf{RS}}(M_2)$, i.e. $M_1 \leqslant_{\mathsf{PW}} M_2$. The opposite implication for the four transformations is shown in the following case analysis, where $\lhd \in \{=, \subseteq\}$. Here we take as

[2]Possible worlds-refinement corresponds to "thorough refinement with respect to ready simulation" in the terminology of Lemma 3.18.

$M\Diamond\ddot{S}$ the transition system equal to M, except that the set of initial states is replaced by $\ddot{S}$.

$M^t_\lhd$: It can be checked that $\{(\ddot{S}_1, \ddot{S}_2) \mid \ddot{S}_1 \neq \emptyset \wedge \ddot{S}_2 \neq \emptyset \wedge \mathbb{T}\$_{\mathsf{RS}}({M_1}^t_\lhd\Diamond\ddot{S}_1) \subseteq \mathbb{T}\$_{\mathsf{RS}}({M_2}^t_\lhd\Diamond\ddot{S}_2)\}$ is a refinement relation witnessing ${M_1}^t_\lhd \leqslant_{\mathsf{RS}} {M_2}^t_\lhd$.

$M^p_\lhd$: It can be checked that $\{((\ddot{S}_1, L_1), (\ddot{S}_2, L_2)) \mid \mathbb{T}\$_{\mathsf{RS}}({M_1}^p_\lhd\Diamond(\ddot{S}_1, L_1)) \subseteq \mathbb{T}\$_{\mathsf{RS}}({M_2}^p_\lhd\Diamond(\ddot{S}_2, L_2))\}$ is a refinement relation witnessing ${M_1}^p_\lhd \leqslant_{\mathsf{RS}} {M_2}^p_\lhd$. ◇

This concludes the proof of Proposition 3.17. □

3.4.2 Consistency checking

As a corollary, our transformations also yield a technique for checking consistency, i.e. for checking whether the language of a model is non-empty. This is trivial for trace-like settings, ready-simulation settings and must-saturated STSs, because these settings cannot describe the empty language.

Corollary 3.20. *For a* DMTS, DMTS$_+$, MTS, MTS$_+$, *or* STS, *consistency can be checked by transforming it via our transformations into an* STS, *applying the algorithm given in Proposition 3.11, and finally checking if the initial state set is non-empty.*

Proof. This is an immediate consequence of (i) Proposition 3.11, (ii) the fact that our transformations preserve the languages and (iii) the fact that all STS$_+$s have a deterministic transition system as implementation (provided the initial state set is non-empty). □

3.5 Related work

For trace and tree languages, many transformations have been developed between automata having different fairness constraints (Büchi, Muller, Rabin, Streett, parity), see e.g. [GTW02] and the references therein. Transformations between non-automata settings are given in [CH93] and [LS91], where the must-testing and ready simulation ($\frac{2}{3}$-bisimulation) preorders, respectively, are transformed to prebisimulation. In [FS05], forward/backward simulation and trace inclusion are transformed to disjunctive modal transition systems (*underspecified* transition systems). These transformations implicitly demonstrate that the transformed settings are less or equally expressive with respect to the describable sets of (not necessarily deterministic) transition systems. Transformations preserving the complete preorder (and not only the languages) are given in [FS07a] and [GJ03], where [FS07a] proves that disjunctive modal transition systems can be transformed into 1-selecting modal transition systems but not vice versa, whereas [GJ03] presents transformations between modal transition system variants with transition labels and predicates on states.

An alternative approach to the examination of expressiveness is taken, e.g., in [EF02, vG01], where preorders are compared regarding their coarseness. The obtained hierarchy – known as the linear-time branching-time spectrum – coincides with ours for many transition system-based settings, but not in general, as illustrated by possible worlds and ready simulation semantics: by definition of possible-worlds semantics, they trivially have the same implementation-set expressiveness, although the possible worlds preorder is finer than the ready simulation preorder. For the coinciding settings, our results cannot be immediately derived from the corresponding results in [vG01]. Consider, e.g., the increase of expressiveness between ready-trace inclusion and ready simulation. This cannot be derived from Counterexample 8 of [vG01], which is illustrated here in Figures 2.1(d) and (e), since both systems have exactly the same sets of implementations, both with respect to ready-trace inclusion and ready simulation.

Yet another approach to studying the expressiveness of refinement settings is via modal logics in the style of Hennessy and Milner [HM85]. While much work focuses on characterizing preorders on general transition systems, [BL92] shows a correspondence between the preorder underlying modal transition systems and the prime and consistent formulas of Hennessy-Milner logic.

The problem of consistency checking is considered e.g. in [HH06]. The authors present an algorithm, along with a complexity study, for checking the consistency of sets of modal transition systems, i.e. for checking non-emptiness of the intersection of the modal transition systems' implementation sets in terms of general transition systems. [LNW07] gives algorithms and complexity results of consistency checks for several refinement notions.

Further refinement settings have been proposed in the literature, albeit for general transition systems rather than deterministic transition systems, e.g. in [JW95, DN04, DN05, SG06, FH06, FHSS09]. It is future work to check whether all of them (when ignoring their possible fairness constraints) can be transformed to disjunctive mixed/modal transition systems and vice versa while preserving their languages in terms of general transition systems.

3.6 Conclusions

This chapter studied the expressiveness of popular transition-system-based specification formalisms with respect to their describable languages in terms of deterministic transition systems. Our results are summarized in the expressiveness hierarchy depicted in Figure 3.2. Our work is of importance for system designers and verification-tool builders alike. The established expressiveness hierarchy aids system designers in selecting the right specification formalism for a problem in hand, while our transformations allow tool builders to reuse refinement checking algorithms across different formalisms.

Regarding future work, we wish to examine the succinctness of our refinement settings, show that our transformations lie in optimal complexity classes, and compare refinement settings based on preorders that abstract from internal computation.

Chapter 4

Nondeterminism in Process Algebra

This chapter gives operational and axiomatic semantics to a process algebra having (i) a parallel operator interpreted in a concurrent (rather than distributed) setting, implying resolvable nondeterminism, and (ii) a persistent choice operator, interpreted as persistent nondeterminism. In order to handle the different kinds of nondeterminism, the operational semantics uses μ-automata as underlying semantic model. Soundness and completeness of our axiom system with respect to the operational semantics is shown.

4.1 Introduction

Process algebras, see [Bae05] for an overview, are standard formalisms for compositionally describing systems based e.g. on synchronous communication on an abstract level by the dependencies of their observable communication. They serve as a domain for semantic foundations of programming or modeling languages and are also used as modeling languages [BB87].

Example 4.1. *Suppose there are two processes running concurrently on a single processor computer which both have the possibility to send print jobs to a printer. Prior to sending the data to the printer, any process must gain exclusive access to the printer by synchronizing on an action* request*. After that, the print job can be sent. In our example, the first process sends a photo (*sendPhoto*), whereas the second sends a document (*sendDoc*). Furthermore, the first process can be disrupted by a user via action* cancel*. This simplified concurrent system (in a parallel environment) can be modeled in process algebra based on synchronous communication by*

$$(\mathsf{request}.\mathsf{sendPhoto} + \mathsf{cancel})\|(\mathsf{request}.\mathsf{sendDoc}) \tag{4.1}$$

where the printer and the user belong to the environment. Here, operator $\|$ denotes parallel composition, $a.B$ denotes action prefix, and $+$ denotes the choice operator. The semantics of process algebra term (4.1) in terms of transition system is given by M, illustrated in Figure 4.1.

In the transition system of the above example two kinds of choice can be distinguished: (i) External choice represented by outgoing edges with different labels. This choice occurs in implementations and remains undecided until in an execution the environment decides which of the possible actions is performed. (ii) Internal (nondeterministic) choice represented by outgoing edges with the same labels. In this example, this nondeterminism is resolved by the scheduler of the operating system since the two processes run on a single processor computer: the scheduler will decide which process may perform its request action, i.e. synchronize to get

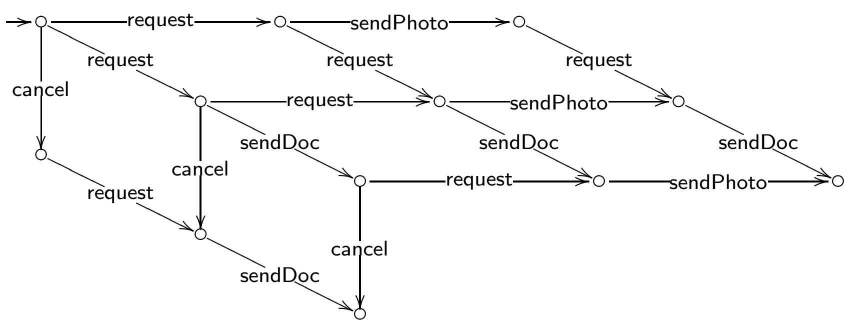


Figure 4.1: Transition system M giving the semantics of process algebra term (4.1).

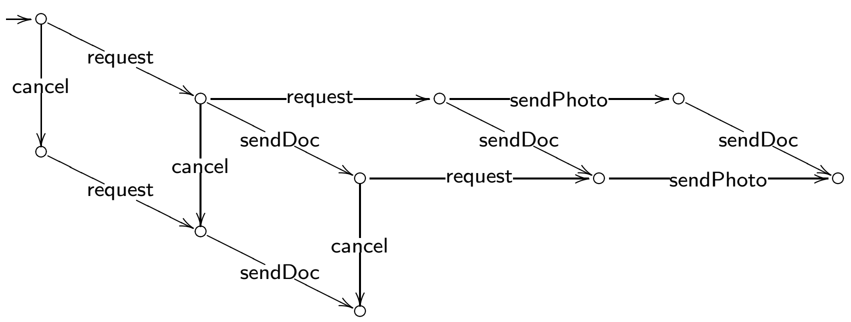


Figure 4.2: Transition system M_{ref} which ready simulation-refines M from Figure 4.1.

exclusive access to the printer. This refinement, performed by adding the scheduler, is formally made precise by ready simulation.

Since this kind of nondeterminism is resolved by the refinement of adding a scheduler, it is *resolvable nondeterminism*. Note, however, that in standard semantics based on bisimulation, the abstraction from schedulers is usually regarded as *persistent* (since transition systems with bisimulation cannot express resolvable nondeterminism). We consider this to be counterintuitive in most cases because schedulers usually do not show "randomized" behavior that is determined independently for each execution.

Branching time logics, like the modal μ-calculus [Koz83], are often used for describing properties of process algebras. Unfortunately, they are not preserved under refinement based on ready simulation. Consider, e.g., the statement that there is an immediate request action enabled such that afterwards no document can be sent to the printer. This property is described by the μ-calculus formula[1]

$$\langle \mathsf{request} \rangle \, ([\mathsf{sendDoc}] \, \mathtt{ff}). \tag{4.2}$$

Property (4.2) holds in M of Figure 4.1, but not in its refinement M_{ref} of Figure 4.2. This illustrates that branching time properties applied to a concurrent setting need to be interpreted with respect to sets of schedulers: $\langle a \rangle \, \phi$ requires the existence of a scheduler such that ϕ holds after the execution of a, and $[a] \, \phi$ states that, independent of which scheduler is chosen, ϕ will hold after the execution of a. Consequently, property (4.2) has to be understood as follows: there is a scheduler for the next step such that request is enabled and its execution (which is deterministic if the scheduler is given) leads to a state where sendDoc is disabled for any

[1]See Appendix A for the definition of the μ-calculus syntax and semantics.

scheduler. Since the refinement in Figure 4.2 specializes schedulers, μ-calculus formula (4.2) is not preserved.

In Example 4.1, we illustrated that resolvable nondeterminism naturally arises through parallel composition. In the following examples we argue that also persistent nondeterminism, i.e. internal choice that remains in running implementations, occurs in applications:

Example 4.2. *In the situation of Example 4.1, assume a faulty channel between the first process and the printer. Then, it is possible that a signal* request *(e.g., encoded as 1) can be turned into a signal* cancel *(e.g., encoded as 0). This is reflected by the process algebra term*

$$(\mathsf{request}.\mathsf{sendPhoto} + \mathsf{request} + \mathsf{cancel})\|(\mathsf{request}.\mathsf{sendDoc})\,. \tag{4.3}$$

We get persistent nondeterminism with respect to label request*, because in case of a faulty transmission it has to be handled as an action* cancel*. The nondeterminism is persistent because refinement cannot decide on the existence of a fault, this is decided for every execution. In other words, the existence of a fault is as uncontrollable as the result of throwing a dice.*

Example 4.3. *In the situation of Example 4.1, consider* request *to be an abstract action [Tho87] for more refined labels like* requestLowRes *and* requestHighRes*, where the first establishes access to the low resolution printing features of the printer, and the second gains access to the high resolution features. Then, based on the printing capabilities offered by the printer, either a high resolution photo (*sendHighResPhoto*) or a low resolution alternative (*sendLowResPhoto*) can be sent. This is reflected by the following process algebra term having persistent nondeterminism with respect to label* request*:*

$$(\mathsf{request}.\mathsf{sendHighResPhoto} + \mathsf{request}.\mathsf{sendLowResPhoto} + \mathsf{cancel})\|(\mathsf{request}.\mathsf{sendDoc})\,. \tag{4.4}$$

Note that in these scenarios we have both persistent nondeterminism, introduced by the choice operator of the first process, and resolvable nondeterminism, introduced by the unknown scheduler of the parallel composition. In case there is no resolvable nondeterminism, i.e., if we consider concrete systems with persistent nondeterminism only, transition systems together with bisimulation as underlying equivalence notion is an appropriate semantic model. Then, the μ-calculus is a suitable logic, since it characterizes transition systems up to bisimulation. Nevertheless, transition systems are not an appropriate model whenever persistent and resolvable nondeterminism occur in a single setting, like in Examples 4.2 and 4.3. This is because a choice for an underlying equivalence or preorder has to be made: bisimulation interprets nondeterminism as persistent (nondeterminism is preserved in both directions) and ready simulation (or similar preorders) as resolvable (nondeterminism can be resolved, as seen in Figures 4.1 and 4.2). Furthermore, when using resolvable nondeterminism we expect a three-valued satisfaction interpretation over the μ-calculus, with the possibility that a formula is neither satisfied nor falsified. For example, we expect that $\langle\mathsf{request}\rangle\,((\langle\mathsf{sendDoc}\rangle\,\mathtt{tt}) \vee (\langle\mathsf{sendLowResPhoto}\rangle\,\mathtt{tt}))$ holds in (4.4), since all its implementations do, but property (4.2) is "unknown" for (4.4) since there are implementations that satisfy the property and there are implementations that do not satisfy the property.

Contribution. In this chapter we give an operational and an axiomatic semantics to a process algebra having a parallel operator interpreted in a concurrent setting and a choice operator interpreted as persistent with respect to equal next-step actions. In particular, we adjust the semantics of [BB94] which has a persistent as well as a resolvable choice operator, to our interpretation of parallel composition, since in their interpretation parallel composition yields persistent nondeterminism. In order to handle the two kinds of nondeterminism adequately, a semantic model with two kinds of transition relations, as in [BB94] and [HJ90], is used, namely μ-automata [JW95] with their standard refinement notion and their three-valued satisfaction relation over the μ-calculus. The two transition relations are defined using structural operational semantics rules. One relation corresponds to the execution of actions, the other corresponds to the removal of underspecification for the next action execution, which is called concretization. In order to develop an axiomatic semantics, the process algebra is extended by further operators, especially by a choice operator corresponding to resolvable nondeterminism. From our axiom system, we derive an expansion theorem which expresses the parallel composition operator in terms of choice operators. Soundness and completeness of this axiom system with respect to the operational semantics is shown.

Outline. Section 4.2 presents the syntax of the considered process algebra, and Section 4.3 gives its operational semantics in terms of μ-automata. Section 4.4 first extends the syntax of the process algebra by further operators (Subsection 4.4.1), then uses the extended syntax in the established (proven to be sound and complete) axiomatic semantics (Subsection 4.4.2), and finally gives the modified expansion theorem in our context and proves its correctness (Subsection 4.4.3). Section 4.5 discusses related work, and Section 4.6 concludes.

4.2 Syntax

In order not to distract from the technical problems and their solutions, we only present a simple process algebra that does not have recursion, sequential composition, and parallel composition with a synchronization mechanism. Instead, our process algebra consists of action prefix, persistent nondeterminism, parallel composition, and action renaming. Note that a (CSP-based) hiding operator is a special case of a renaming operator where the action is renamed to the internal action. Here, we follow the philosophy that internal actions are observable. In other words, we do not consider weak equivalence or weak refinement notions. Furthermore, we assume that nondeterminism obtained via "mixed choices" is resolved inside the environment: for example, if the system provides actions a and b and the environment their corresponding counterparts, then the environment decides whether a or b is executed. The renaming/hiding operator is also of special interest in our setting since it introduces nondeterminism, too. For example, the process that provides action a leading to B_1 and that provides action b leading to B_2 does not contain nondeterminism, but after hiding a and b, i.e., renaming a and b to the internal action, nondeterminism occurs since now an internal step either leads to B_1 or to B_2. Again, both interpretations of persistent or resolvable nondeterminism are possible for the renaming operator. But in our setting nondeterminism obtained through hiding (and therefore implicitly for renaming) must be interpreted as resolvable nondeterminism. This is argued as follows: The system has a stimulus for any hidden action. These hidden actions can be considered to originate from an additional component that continuously provides these hidden actions. Then, the scheduler of the parallel composition decides which parallel component executes next. Consequently, hiding (and therefore renaming) yields resolvable nondeterminism with our interpretation of schedulers of parallel components, which assumes that schedulers do not behave randomly.

PA, the set of all basis process algebra terms, is generated by

$$B \quad ::= \quad 0 \mid a.B \mid B + B \mid B \| B \mid B\langle a/b \rangle \,,$$

where $a, b \in \mathcal{L}$. Process 0 describes a deadlocked process, i.e. no further actions can be executed. We sometimes omit symbol 0 by writing a instead of $a.0$. Process $a.B$ allows the execution of action a resulting in process B. Persistent choice[2] is described by $B_1 + B_2$, and $B_1 \| B_2$ describes parallel composition. The parallel composition has implicit resolvable nondeterminism, introduced by abstraction from a scheduler favoring one of the two sides. $B\langle a/b \rangle$ describes the process where the execution of a in B becomes the execution of b. All other action execution, including action b, remains unaffected. Note that this renaming process also introduces resolvable nondeterminism as described at the beginning of this section.

4.3 Operational semantics

Our semantic model is μ-automata, as defined in Section 2.3.2. This model is equipped with two transition relations where the **branch**-transition relation, leading from **branch**-states to **or**-states and denoted by $\longrightarrow$, is used for execution steps, with nondeterminism interpreted as persistent. The **or**-transition relation, leading from **or**-states to **branch**-states and denoted by $\rightrightarrows$, is used for the definition of resolvable nondeterminism. μ-automata come with a three-valued satisfaction relation for the μ-calculus, which is preserved under refinement [DN05].

In order to define the operational semantics we use two different kinds of expressions: one where underspecification for the next step is allowed (PA, corresponding to the **or**-states) and one where it is not ($\mathsf{PA}^{\mathrm{con}}$, corresponding to the **branch**-states), i.e. where resolvable nondeterminism for the next execution is resolved[3]. Then, additionally to the step transition relation ($\longrightarrow$, corresponding to **branch**-transition relation) from $\mathsf{PA}^{\mathrm{con}}$ to PA, a concretization relation ($\rightrightarrows$, corresponding to the **or**-transition relation) from PA to $\mathsf{PA}^{\mathrm{con}}$, which resolves the resolvable nondeterminism for the next execution, is used. Formally, $\mathsf{PA}^{\mathrm{con}}$ denotes the set of all process algebra terms generated by

$$P \quad ::= \quad [0] \mid [a.\widehat{B}] \mid P + P \mid P|\rangle_{\widehat{B},A,\widetilde{B}} P \mid P\langle a/b, v \rangle \,,$$

where $a, b \in \mathcal{L}$, $A \subseteq \mathcal{L}$, $\widehat{B}, \widetilde{B} \in \mathsf{PA}$, and $v \in \{\mathbf{d}, \mathbf{s}\}$. The intuition of the operators is similar to the one of the operators given in Section 4.2, except that here the scheduler of the parallel composition and of the renaming operator is determined for the next step. First, we explain the intuition of parallel composition $P_1|\rangle_{B_1,A,B_2} P_2$, at first neglecting B_1 and B_2, which will be explained later, and using the notation $P_1|\rangle_A P_2$ instead. Here, A specifies a scheduler, which need not be explicitly specified in standard semantics based on ready simulation, since there schedulers are adequately handled via ready simulation. However, the scheduler needs to be modeled if both resolvable nondeterminism and persistent choice should appear in one setting. The scheduler information is interpreted as follows: The right side is favored in $P_1|\rangle_A P_2$ for actions in A, whereas the left side is favored for actions in $\mathcal{L} \setminus A$. This favoring concerns only the next step, i.e., after the execution of an action, any scheduling is allowed again. This is even the case if an action parallel to $P_1|\rangle_A P_2$ is executed, e.g., if P_3 executes action a leading to

[2]In Section 4.4.1 we will also introduce a resolvable choice operator $\oplus$.
[3]The superscript in $\mathsf{PA}^{\mathrm{con}}$ is short for "concrete".

Table 4.1: Resolution of next-step-underspecification by relation $\rightrightarrows\ \subseteq \mathsf{PA} \times \mathsf{PA}^{\mathrm{con}}$.

$$\frac{}{0 \rightrightarrows [0]} \qquad \frac{}{a.B \rightrightarrows [a.B]} \qquad \frac{B_1 \rightrightarrows P_1 \quad B_2 \rightrightarrows P_2}{B_1 + B_2 \rightrightarrows P_1 + P_2}$$

$$\frac{B_1 \rightrightarrows P_1 \quad B_2 \rightrightarrows P_2 \quad A \subseteq \mathcal{L}}{B_1 \| B_2 \rightrightarrows P_1 |\rangle_{B_1,A,B_2} P_2} \qquad \frac{B \rightrightarrows P \quad v \in \{\mathbf{d}, \mathbf{s}\}}{B\langle a/b\rangle \rightrightarrows P\langle a/b, v\rangle}$$

B_3 in $(P_1|\rangle_A P_2)|\rangle_A P_3$, the resulting process is, roughly speaking, $(P_1\|P_2)\|B_3$, where all next step scheduling is removed. Note that this approach is more appropriate than the approach in which the partial scheduler is kept (in this case $(P_1|\rangle_A P_2)\|B_3$ would be the result), since the scheduler is global and therefore can depend on any past execution.[4] Furthermore, associativity of $\|$ would be lost in the alternative approach, which is illustrated later in Example 4.6. In order to model the undoing of the scheduler information efficiently, the parallel composition stores the original processes of its components (here B_1 and B_2) and replaces the non-executing component by its stored original one where no scheduler information is present. This will be clarified by the transition rules.

The scheduler information v of the next execution is added to the renaming operator: the execution of the action corresponding to the <u>s</u>ource label a of the renaming, which becomes b, is favored in $B\langle a/b, \mathbf{s}\rangle$, whereas the execution of the action corresponding to the <u>**d**</u>estination label b is favored in $B\langle a/b, \mathbf{d}\rangle$. Here, favoring means that for $B\langle a/b, \mathbf{d}\rangle$, process B may only execute a (which will be renamed to b) if B cannot execute b, and analogously for $B\langle a/b, \mathbf{s}\rangle$ where a is favored. Again, this scheduling of the renaming operator only applies for the next action execution, i.e. after the execution of any action, possibly different from b, the current favoring is removed.

The concretization relation $\rightrightarrows$ is given in the upper section of Table 4.1, where the underspecification of the next step is resolved, and the step transition relation $\longrightarrow$ is presented in the lower section of Table 4.2, where actions are executed resulting in processes with underspecification kept for the next step executions.

We give some comments on $\rightrightarrows$. The resolution of next-step-underspecification has to take place in every subpart that can potentially make the next execution, consequently resolution does not take place for B in $a.B$. In the parallel composition, however, the next-step-underspecification is resolved by choosing an arbitrary scheduler. The parallel composition operator stores the original processes such that it can efficiently undo the concretization. In the renaming operator, the next-step-underspecification is resolved by either favoring the action corresponding to the source label ($\mathbf{s}$) or favoring the action corresponding to the destination label ($\mathbf{d}$) of the renaming.

We proceed with some comments on $\longrightarrow$. The rules for $[a.B]$ and $P_1 + P_2$ are standard. The left side of the parallel composition $P_1|\rangle_{B_3,A,B_4} P_2$ can execute a if (i) the left side is favored for a by the scheduler ($a \notin A$) or (ii) the right side does not provide a ($P_2 \stackrel{a}{\not\longrightarrow}$). Symmetric

[4]The approach where the partial scheduler is kept makes only sense if each parallel component has its own scheduler and there is an additional global scheduler that decides which of the parallel components is favored.

Table 4.2: Action execution by relation $\longrightarrow \;\subseteq \mathsf{PA}^{\mathrm{con}} \times \mathcal{L} \times \mathsf{PA}$. Here, $\Rightarrow$ denotes logical implication.

$$\frac{}{[a.B] \xrightarrow{a} B} \qquad \frac{i \in \{1,2\} \qquad P_i \xrightarrow{a} B}{P_1 + P_2 \xrightarrow{a} B}$$

$$\frac{P_1 \xrightarrow{a} B_1 \qquad a \in A \Rightarrow P_2 \not\xrightarrow{a}}{\begin{array}{c} P_1 |\rangle_{B_3,A,B_4} P_2 \xrightarrow{a} B_1 \| B_4 \\ P_2 |\rangle_{B_4,\mathcal{L}\setminus A,B_3} P_1 \xrightarrow{a} B_4 \| B_1 \end{array}} \qquad \frac{P \xrightarrow{c} B \qquad c \notin \{a,b\}}{P\langle a/b, v\rangle \xrightarrow{c} B\langle a/b\rangle}$$

$$\frac{P \xrightarrow{b} B \qquad v = \mathbf{s} \Rightarrow P \not\xrightarrow{a}}{P\langle a/b, v\rangle \xrightarrow{b} B\langle a/b\rangle} \qquad \frac{P \xrightarrow{a} B \qquad v = \mathbf{d} \Rightarrow P \not\xrightarrow{b}}{P\langle a/b, v\rangle \xrightarrow{b} B\langle a/b\rangle}$$

constraints hold for the execution of a on the right side of $P_1 |\rangle_{B_3,A,B_4} P_2$. As already mentioned before, the next-step-underspecification resolution has to be undone for the parallel component that did not make the execution. Therefore, process B_3 or B_4 replaces the non-executed side. In $P\langle a/b, v\rangle$ an action c different from a and b can be executed leading to $B\langle a/b\rangle$, whenever P can execute c leading to B. This is stated in the first rule. Furthermore, $P\langle a/b, v\rangle$ can execute b, leading to a process $B\langle a/b\rangle$, where B can be obtained after executing b in P, whenever v favors the destination label ($v = \mathbf{d}$) or no action a is provided by P. Similarly, $P\langle a/b, v\rangle$ can execute b, leading to a process $B\langle a/b\rangle$, where B can be obtained after executing a in P, whenever v favors the source label ($v = \mathbf{s}$) or no action b is provided by P. In all these three cases, the scheduler is removed after the execution. Note that by definition, the target processes of $\longrightarrow$ do not contain any scheduling information.

Definition 4.4 (Operational semantics). *The operational semantics of a process algebra term $B \in \mathsf{PA}$ is the μ-automaton $(\mathsf{PA}, \mathsf{PA}^{\mathrm{con}}, \{B\}, \rightrightarrows, \longrightarrow)$, where $\rightrightarrows$ is given as in Table 4.1 and $\longrightarrow$ is given as in Table 4.2. We say that a process algebra term B from* PA refines *another one B', written $B \leq B'$, if the operational semantics of B μ-refines the operational semantics of B'. Furthermore, B is* refinement equivalent *to B', written $B \equiv B'$, if B refines B' and B' refines B.*

Example 4.5. *The operational semantics for $(a + a.b)\|a$ is illustrated in Figure 4.3.*

Example 4.6. *We illustrate that associativity of parallel composition does not hold, if the resolution of underspecification in the parallel composition rule of Table 4.2 is not undone, i.e., if the original process does not replace the current process in case of non-execution. Under this assumption, $\tilde{B}_1 = a.b\|(a\|a.c)$ does not refine $\tilde{B}_2 = (a.b\|a)\|a.c$, which is seen as follows. By definition $\tilde{B}_1 \rightrightarrows [a.b] \,|\rangle_{\mathcal{L}}([a] \,|\rangle_{\mathcal{L}}[a.c])$. Since action c has to be possible afterwards, this process can only be adequately matched by $\tilde{B}_2$ via a process that is refinement equivalent to $([a.b] \,|\rangle_{\{a\}}[a])|\rangle_{\{a\}} [a.c]$ or to $([a.b] \,|\rangle_{\emptyset}[a])|\rangle_{\{a\}} [a.c]$. Thus after the execution of a we would need that $a.b\|(a\|c)$ refines $([a.b] \,|\rangle_{\{a\}}[a])\| [c]$ or refines $([a.b] \,|\rangle_{\emptyset}[a])\| [c]$. Furthermore, $a.b\|(a\|c)$ can be concretized such that either b is possible after the execution of a or no b is possible after the execution of a. But only one of these concretizations is possible in $([a.b] \,|\rangle_{\{a\}}[a])\| [c]$ and in $([a.b] \,|\rangle_{\emptyset}[a])\| [c]$. Hence, $\tilde{B}_1$ does not refine $\tilde{B}_2$.*

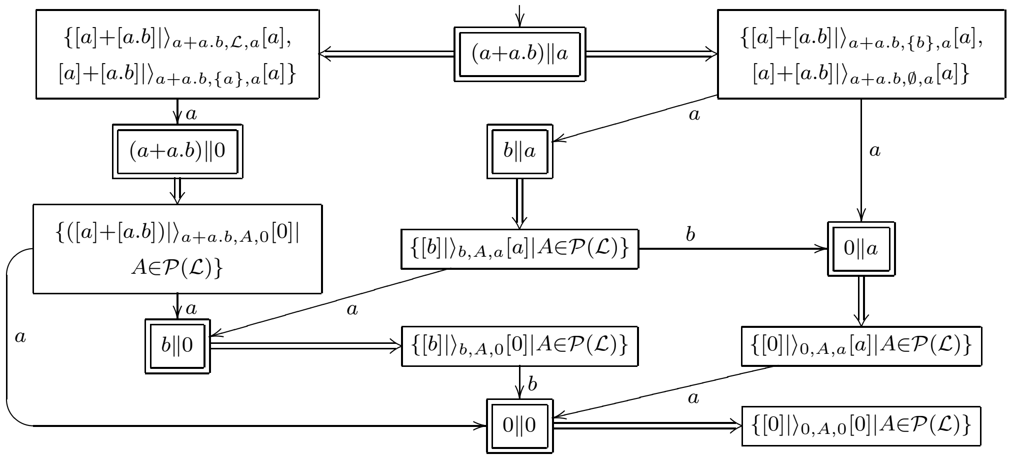

$\{[a]+[a.b]|\rangle_{a+a.b,\mathcal{L},a}[a],$
$[a]+[a.b]|\rangle_{a+a.b,\{a\},a}[a]\}$
$(a+a.b)\|a$
$\{[a]+[a.b]|\rangle_{a+a.b,\{b\},a}[a],$
$[a]+[a.b]|\rangle_{a+a.b,\emptyset,a}[a]\}$
$(a+a.b)\|0$
$b\|a$
$\{([a]+[a.b])|\rangle_{a+a.b,A,0}[0]|$
$A\in\mathcal{P}(\mathcal{L})\}$
$\{[b]|\rangle_{b,A,a}[a]|A\in\mathcal{P}(\mathcal{L})\}$
$0\|a$
$b\|0$
$\{[b]|\rangle_{b,A,0}[0]|A\in\mathcal{P}(\mathcal{L})\}$
$\{[0]|\rangle_{0,A,a}[a]|A\in\mathcal{P}(\mathcal{L})\}$
$0\|0$
$\{[0]|\rangle_{0,A,0}[0]|A\in\mathcal{P}(\mathcal{L})\}$
a
b

Figure 4.3: The operational semantics of $(a + a.b)\|a$, where $\mathcal{L} = \{a, b\}$. **or**-states of the μ-automaton have double-lined frames, whereas **branch**-states have single-lined frames. **or**-transitions are drawn as double-line arrows, whereas **branch**-transitions are drawn as labeled single-line arrows. A state described by a set stands for a set of states, described by the elements of the set and having the same incoming and outgoing transitions as the state labeled with the set.

Example 4.7. *Process $a.(B_1\|(a.B_2))$ refines $(a.B_1)\|(a.B_2)$, but not vice versa. This illustrates that refinement over* PA *is no equivalence relation.*

Theorem 4.8. *Refinement is preserved under all process algebra operators, i.e., if $B_1 \le B_1' \wedge B_2 \le B_2'$ then*

(i) $a.B_1 \le a.B_1'$,

(ii) $B_1 + B_2 \le B_1' + B_2'$,

(iii) $B_1\|B_2 \le B_1'\|B_2'$, *and*

(iv) $B_1\langle a/b\rangle \le B_1'\langle a/b\rangle$.

Proof. We only show the preservedness of refinement under parallel composition, since the other operators can be easier shown by using the same technique. Let R be a μ-refinement relation containing (B_1, B_1') and (B_2, B_2'). Define $R' = \{(B_3\|B_4, B_3'\|B_4') \mid (B_3, B_3') \in R \wedge (B_4, B_4') \in R\} \cup \{(P_1|\rangle_{B_3,A,B_4}P_2, P_1'|\rangle_{B_3',A,B_4'}P_2') \mid A \subseteq \mathcal{L} \wedge (P_1, P_1') \in R \wedge (P_2, P_2') \in R \wedge (B_3, B_3') \in R \wedge (B_4, B_4') \in R\}$. Then it can be checked that R' is a μ-refinement relation, which establishes $B_1\|B_2 \le B_1'\|B_2'$, as required. □

Note that the straightforward extension of $\le$ to PA^{con} is also preserved under all process algebra operators for PA^{con}.

4.4 Axiomatic semantics

In this section we present a sound and complete axiom system for refinement over PA.

Table 4.3: Additional transition rules for the extended process algebra.

$$\frac{i \in \{1,2\} \qquad B_i \rightrightarrows P}{B_1 \oplus B_2 \rightrightarrows P} \qquad \frac{B_2 \rightrightarrows P_2}{c.B_1 >_a B_2 \rightrightarrows c.B_1 >_a P_2}$$

$$\frac{B \rightrightarrows P}{B \setminus a \rightrightarrows P \setminus a} \qquad \frac{B_1 \rightrightarrows P_1 \qquad B_2 \rightrightarrows P_2}{B_1 |\rangle_{B_3,A,B_4} B_2 \rightrightarrows P_1 |\rangle_{B_3,A,B_4} P_2} \qquad \frac{B \rightrightarrows P}{B\langle a/b, v\rangle \rightrightarrows P\langle a/b, v\rangle}$$

$$\frac{P_2 \overset{a}{\not\longrightarrow}}{c.B_1 >_a P_2 \xrightarrow{c} B_1} \qquad \frac{P \xrightarrow{a} B \qquad b \neq a}{P \setminus b \xrightarrow{a} B}$$

4.4.1 Syntax extension

In order to define the axioms, further terms are introduced: the most interesting one is the resolvable nondeterminism operator $\oplus$, which is also of interest for modeling by itself. The intuition of resolvable nondeterminism $B_1 \oplus B_2$ is that either B_1 or B_2 is implemented, but not both. Thus, $B_1 + B_2$ is in general not an allowed implementation.

Further added terms are: (i) A next step restriction process $B \setminus a$, where B may not execute action a as its next step. Note that action a may be executed if an action different from a is executed before. (ii) A conditional prefix term $c.B_1 >_a B_2$, which is equivalent to $c.B_1$ whenever B_2 cannot execute action a in its next step, and it is equivalent to 0 if B_2 can execute action a in its next step. (iii) A parallel composition $B_1 |\rangle_{B_3,A,B_4} B_2$ where the scheduler information for the next execution and the replacing processes for non-execution are already given. (iv) A renaming operator $B\langle a/b, v\rangle$, where the scheduler information for the next execution is already present. Also some counterparts of these new expressions are added to the process algebra terms where the next-step-underspecification is resolved. Formally, we define $\widetilde{\mathsf{PA}}$ to be the set of all process algebra terms generated by

$$\begin{aligned} B \ ::= \ & 0 \mid a.B \mid B + B \mid B \| B \mid B\langle a/b\rangle \mid B \oplus B \mid B \setminus a \mid c.B >_a B \mid B|\rangle_{B,A,B} B \mid \\ & B\langle a/b, v\rangle \end{aligned}$$

and we define $\widetilde{\mathsf{PA}}^{\mathrm{con}}$ to be the set of all process algebra terms generated by

$$P \ ::= \ [0] \mid [a.\widehat{B}] \mid P + P \mid P|\rangle_{\widehat{B},A,\widetilde{B}} P \mid P\langle a/b, v\rangle \mid P \setminus a \mid c.\widehat{B} >_a P$$

where $a, b, c \in \mathcal{L}$, $A \subseteq \mathcal{L}$, $\widehat{B}, \widetilde{B} \in \widetilde{\mathsf{PA}}$ and $v \in \{\mathbf{d}, \mathbf{s}\}$. The previous transition rules (Table 4.1) are extended by those in Table 4.3. Note that refinement is preserved also under the newly introduced operators, as is easily checked.

We comment on $\rightrightarrows$. The resolvable nondeterminism $\oplus$ is resolved by choosing either the right or the left side and resolving this term. In $B_1 |\rangle_{B_3,A,B_4} B_2$ only B_1 and B_2 have to be resolved, since no next step analysis takes place in term B_3 and B_4. By the same argument, only B_2 has to be resolved in $c.B_1 >_a B_2$.

Table 4.4: Definition of predicate nuf.

$$\frac{}{\mathsf{nuf}(0)} \qquad \frac{}{\mathsf{nuf}(a.B_1)} \qquad \frac{\mathsf{nuf}(B_1) \quad \mathsf{nuf}(B_2)}{\mathsf{nuf}(B_1 + B_2)} \qquad \frac{\mathsf{nuf}(B)}{\mathsf{nuf}(B \setminus a)}$$

$$\frac{\mathsf{nuf}(B_2)}{\mathsf{nuf}(c.B_1 >_a B_2)} \qquad \frac{\mathsf{nuf}(B_1) \quad \mathsf{nuf}(B_2)}{\mathsf{nuf}(B_1 |\rangle_{B_3,A,B_4} B_2)} \qquad \frac{\mathsf{nuf}(B)}{\mathsf{nuf}(B\langle a/b, v\rangle)}$$

Now some comments on $\longrightarrow$. In $c.B_1 >_a P_2$ a c-step to B_1 is possible iff the right hand side cannot execute a. In $P \setminus b$ all executions of P that differ from b can take place as next step, in which case $\setminus b$ is removed, since this restriction only holds for the next step execution.

Remark 4.9. *For terms of* PA*, the same set of provided actions is obtained after every* $\rightrightarrows$*-step. This does not hold for terms of* $\widetilde{\mathsf{PA}}$*, as, e.g., illustrated by* $a \oplus 0$*.*

The operational semantics of terms in $\widetilde{\mathsf{PA}}$ and refinement (equivalence) over $\widetilde{\mathsf{PA}}$ are defined as in Definition 4.4 except that the extended concretization and step transition relations are used.

4.4.2 Axioms

Before we present the axiom system, we discuss some of the (standard) axioms that do not hold in general, i.e. where refinement equivalence cannot be guaranteed, which is denoted by $\not\simeq$:

First, $B + B \not\simeq B$, since the left hand side allows more kinds of resolution of underspecification. For example, $(a \oplus b) + (a \oplus b)$ can be resolved to $[a] + [b]$, which provides a as well as b. On the other hand, $a \oplus b$ cannot be resolved such that both actions are provided. By similar arguments,

$$(B_1 \| B_2) + (B_1 \| B_3) \not\simeq B_1 \| (B_2 + B_3), \tag{4.5}$$

since the underspecification in B_1 can possibly be resolved in two different ways, such that it cannot be matched by a single resolution of B_1. More precisely, after $\rightrightarrows$ it is possible that the left hand side of (4.5) can provide more actions than the right hand side. In case B_1 is next-step-underspecification-free, e.g. if B_1 is of form $\sum_{i \in I} a_i.B_i$ for some I, a_i, and B_i, we do not have this problem. Predicate nuf on $\widetilde{\mathsf{PA}}$, which is formally defined in Table 4.4, collects those **n**ext-step-**u**nderspecification-**f**ree processes. Resolution of next-step-underspecification yields a unique term if nuf holds:

Lemma 4.10. *Suppose* $B \in \widetilde{\mathsf{PA}}$ *and* $\mathcal{L} \neq \emptyset$*. If* $\mathsf{nuf}(B)$*, then* B *has a unique target with respect to* $\rightrightarrows$*, i.e.,* $\mathsf{nuf}(B) \Rightarrow |\{B\} \circ \rightrightarrows| = 1$*.*

Proof. Can be straightforwardly shown by induction on the structure of B. □

Nevertheless, the processes of (4.5) are not equivalent even if $\mathsf{nuf}(B_1)$ holds since the resolvable nondeterminism of the parallel composition can be resolved in different ways. For example, after applying $\rightrightarrows$ on $(a.b)\|(a.c+0)$, there is exactly one transition t labeled with a, and either b or c is possible afterwards. On the other hand a $\rightrightarrows$ step from $((a.b)\|(a.c)) + ((a.b)\|0)$ exists where two transitions t_1, t_2 labeled with a are enabled such that b is possible after t_1 and c is possible after t_2. We remedy this problem by resolving the resolvable nondeterminism of the parallel operator before parallel composition is expanded. This motivates why we introduced the term $B_1|\rangle_{B_3,A,B_4}B_2$ already in $\widetilde{\mathsf{PA}}$.

In Table 4.5, axioms for the refinement relation $\preceq$ over $\widetilde{\mathsf{PA}}$ are presented, where we assume, as usual, that $\mathcal{L}$ is finite. Furthermore, $\simeq$ is defined to be $\preceq \cap \succeq$, $\bigoplus_{i\in\{j\}} B_i$ is defined to be B_j, and for finite I, $\bigoplus_{i\in I} B_i$ is defined to be $B_{i'} \oplus (\bigoplus_{i\in I\setminus\{i'\}} B_i)$ where $i' \in I$ (by the associativity and commutativity of $\oplus$ the definition is independent of the chosen i').

Theorem 4.11. *The axioms from Table 4.5 are sound and complete, i.e., $\forall B_1, B_2 \in \widetilde{\mathsf{PA}} : B_1 \preceq B_2 \iff B_1 \leq B_2$.*

Proof. Soundness follows from the soundness of each axiom, which can be checked straightforwardly. Completeness is shown as follows: by applying induction on an adequate well-founded order on terms[5] it can be checked that every term can be reduced to a form

$$\bigoplus_{i=0}^{n} \sum_{j=0}^{m_{(i)}-1} a_{i,j}.B_{i,j},$$

where also $B_{i,j}$ is of that form (note that we do not consider recursion).

By using Axioms A1 – A7 together with Axiom A32, two refining terms having the above form can be transformed to each other, which is shown as follows:

Suppose $B = \bigoplus_{i=0}^{n} \sum_{j=0}^{m_{(i)}-1} a_{i,j}.B_{i,j} \leq \bigoplus_{i'=0}^{n'} \sum_{j'=0}^{m_{(i')}-1} a'_{i',j'}.B'_{i',j'} = B'$, where $B_{i,j}$ and $B'_{i',j'}$ are also of that form. We show $B \preceq B'$ by induction on the maximum action prefix depth of B, which is always finite since there is no recursion. From $B \leq B'$ we get, for all $i \leq n$, that there exists some $i'_i \leq n'$ such that $\sum_{j=0}^{m_{(i)}-1} a_{i,j}.B_{i,j} \leq \sum_{j'=0}^{m_{(i'_i)}-1} a'_{i'_i,j'}.B'_{i'_i,j'}$. Then, by Axioms A32 and A3, we obtain

$$\bigoplus_{i=0}^{n} \sum_{j'=0}^{m_{(i'_i)}-1} a'_{i'_i,j'}.B'_{i'_i,j'} \preceq \bigoplus_{i'=0}^{n'} \sum_{j'=0}^{m_{(i')}-1} a'_{i',j'}.B'_{i',j'} = B'\,.$$

From $\sum_{j=0}^{m_{(i)}-1} a_{i,j}.B_{i,j} \leq \sum_{j'=0}^{m_{(i'_i)}-1} a'_{i'_i,j'}.B'_{i'_i,j'}$ we get (α) for every $j < m_{(i)}$ there is $j'_{i,j} < m_{(i'_i)}$ such that $a_{i,j} = a'_{i'_i,j'_{i,j}}$ and $B_{i,j} \leq B'_{i'_i,j'_{i,j}}$ and (β) for every $j' < m_{(i'_i)}$ there is $j_{i,j'} < m_{(i)}$ such that $a_{i,j_{i,j'}} = a'_{i'_i,j'}$ and $B_{i,j_{i,j'}} \leq B'_{i'_i,j'}$. Then, by Axioms A4 – A6, we obtain

$$\bigoplus_{i=0}^{n} \left(\sum_{j'=0}^{m_{(i'_i)}-1} a_{i,j_{i,j'}}.B'_{i'_i,j'} \right) + \left(\sum_{j=0}^{m_{(i)}-1} a'_{i'_i,j'_{i,j}}.B'_{i'_i,j'_{i,j}} \right) \preceq \bigoplus_{i=0}^{n} \sum_{j'=0}^{m_{(i'_i)}-1} a'_{i'_i,j'}.B'_{i'_i,j'}\,.$$

[5]Terms in the described normal form are assigned a value of 0. For other terms, determine their syntax tree and multiply the values of substructures with a suitable constant, e.g. $|2^{\mathcal{L}}| + 1$. Then, add the weights of all orthogonal branches.

Table 4.5: Axiomatic semantics of the refinement relation $\preceq$ over $\widetilde{\mathsf{PA}}$.

$(A1)$	$B_1 \oplus B_2 \simeq B_2 \oplus B_1$	
$(A2)$	$B_1 \oplus (B_2 \oplus B_3) \simeq (B_1 \oplus B_2) \oplus B_3$	
$(A3)$	$B \oplus B \simeq B$	
$(A4)$	$B_1 + B_2 \simeq B_2 + B_1$	
$(A5)$	$B_1 + (B_2 + B_3) \simeq (B_1 + B_2) + B_3$	
$(A6)$	$a.B + a.B \simeq a.B$	
$(A7)$	$B + 0 \simeq B$	
$(A8)$	$B_1 + (B_2 \oplus B_3) \simeq (B_1 + B_2) \oplus (B_1 + B_3)$	
$(A9)$	$B_1 \Vert B_2 \simeq \bigoplus_{A \subseteq \mathcal{L}} (B_1 \vert\rangle_{B_1,A,B_2} B_2)$	
$(A10)$	$B_1 \vert\rangle_{B_4,A,B_5} B_2 \simeq B_2 \vert\rangle_{B_5,\mathcal{L}\setminus A,B_4} B_1$	
$(A11)$	$B_1 \vert\rangle_{B_4,A,B_5} (B_2 \oplus B_3) \simeq (B_1 \vert\rangle_{B_4,A,B_5} B_2) \oplus (B_1 \vert\rangle_{B_4,A,B_5} B_3)$	
$(A12)$	$B_1 \vert\rangle_{B_4,A,B_5} (B_2 + a.B_3) \simeq ((B_1 \setminus a) \vert\rangle_{B_4,A,B_5} B_2) + a.(B_4 \Vert B_3)$	if $a \in A$
$(A13)$	$B_1 \vert\rangle_{B_4,A,B_5} (B_2 + a.B_3) \simeq (B_1 \vert\rangle_{B_4,A,B_5} B_2) + (a.(B_4 \Vert B_3) >_a B_1)$	if $a \notin A \wedge \mathsf{nuf}(B_1)$
$(A14)$	$0 \vert\rangle_{B_4,A,B_5} 0 \simeq 0$	
$(A15)$	$B\langle a/b\rangle \simeq B\langle a/b, \mathbf{d}\rangle \oplus B\langle a/b, \mathbf{s}\rangle$	
$(A16)$	$(B_1 \oplus B_2)\langle a/b, v\rangle \simeq B_1\langle a/b, v\rangle \oplus B_2\langle a/b, v\rangle$	
$(A17)$	$(B_1 + c.B_2)\langle a/b, v\rangle \simeq B_1\langle a/b, v\rangle + c.(B_2\langle a/b\rangle)$	if $c \notin \{a, b\}$
$(A18)$	$(B_1 + a.B_2)\langle a/b, \mathbf{d}\rangle \simeq B_1\langle a/b, \mathbf{d}\rangle + (b.(B_2\langle a/b\rangle) >_b B_1)$	if $\mathsf{nuf}(B_1)$
$(A19)$	$(B_1 + b.B_2)\langle a/b, \mathbf{d}\rangle \simeq (B_1 \setminus a)\langle a/b, \mathbf{d}\rangle + b.(B_2\langle a/b\rangle)$	
$(A20)$	$(B_1 + a.B_2)\langle a/b, \mathbf{s}\rangle \simeq (B_1 \setminus b)\langle a/b, \mathbf{s}\rangle + b.(B_2\langle a/b\rangle)$	
$(A21)$	$(B_1 + b.B_2)\langle a/b, \mathbf{s}\rangle \simeq B_1\langle a/b, \mathbf{s}\rangle + (b.(B_2\langle a/b\rangle) >_a B_1)$	if $\mathsf{nuf}(B_1)$
$(A22)$	$0\langle a/b, v\rangle \simeq 0$	
$(A23)$	$(B_1 \oplus B_2) \setminus a \simeq (B_1 \setminus a) \oplus (B_2 \setminus a)$	
$(A24)$	$(B_1 + B_2) \setminus a \simeq (B_1 \setminus a) + (B_2 \setminus a)$	
$(A25)$	$0 \setminus a \simeq 0$	
$(A26)$	$(a.B) \setminus a \simeq 0$	
$(A27)$	$(b.B) \setminus a \simeq b.B$	if $b \neq a$
$(A28)$	$c.B_1 >_a (B_2 \oplus B_3) \simeq (c.B_1 >_a B_2) \oplus (c.B_1 >_a B_3)$	
$(A29)$	$c.B_1 >_a (B_2 + a.B_3) \simeq 0$	
$(A30)$	$c.B_1 >_a (B_2 + b.B_3) \simeq c.B_1 >_a B_2$	if $b \neq a$
$(A31)$	$c.B >_a 0 \simeq c.B$	
$(A32)$	$B_1 \preceq B_1 \oplus B_2$	

Induction yields $B_{i,j} \preceq B'_{i'_i,j'_{i,j}}$ and $B_{i,j_{i,j'}} \preceq B'_{i'_i,j'}$. Therefore, we obtain

$$\bigoplus_{i=0}^{n} \left(\sum_{j'=0}^{m_{(i'_i)}-1} a_{i,j_{i,j'}}.B_{i,j_{i,j'}} \right) + \left(\sum_{j=0}^{m_{(i)}-1} a_{i,j}.B_{i,j} \right) \preceq$$
$$\bigoplus_{i=0}^{n} \left(\sum_{j'=0}^{m_{(i'_i)}-1} a_{i,j_{i,j'}}.B'_{i'_i,j'} \right) + \left(\sum_{j=0}^{m_{(i)}-1} a'_{i'_i,j'_{i,j}}.B'_{i'_i,j'_{i,j}} \right).$$

By Axioms A4 – A6 we get

$$B = \bigoplus_{i=0}^{n} \sum_{j=0}^{m_{(i)}-1} a_{i,j}.B_{i,j} \preceq \bigoplus_{i=0}^{n} \left(\sum_{j'=0}^{m_{(i'_i)}-1} a_{i,j_{i,j'}}.B_{i,j_{i,j'}} \right) + \left(\sum_{j=0}^{m_{(i)}-1} a_{i,j}.B_{i,j} \right),$$

as required. □

Example 4.12. *By using Axiom A32, we get*

$$a.(b \oplus c) \preceq (a.(b \oplus c)) \oplus (a.b).$$

Furthermore, by using Axiom A32 and then Axiom A3, we obtain

$$(a.(b \oplus c)) \oplus (a.b) \preceq (a.(b \oplus c)) \oplus (a.(b \oplus c)) \simeq a.(b \oplus c).$$

Thus, we have shown that

$$a.(b \oplus c) \simeq (a.(b \oplus c)) \oplus (a.b).$$

Note that this cannot be shown by using only Axioms A1 – A31 since the necessary removal of $\oplus$ then becomes impossible. This illustrates that Axiom A32 is essential for the completeness of our axiom system with respect to $\simeq$.

4.4.3 Expansion theorem

How parallel composition can be expressed in terms of nondeterminism is known as the expansion theorem [Mil80, BK85]. In our setting, the expansion theorem is more complicated than usual since resolvable as well as persistent nondeterminism has to be handled. The theorem is directly derived from our axiom system and illustrated in the following, where $\sum_{j=0}^{-1} B_j$ yields 0:

Theorem 4.13 (Expansion theorem).

$$B \| B' \simeq \bigoplus_{A \subseteq \mathcal{L}} \bigoplus_{i=0}^{n} \bigoplus_{i'=0}^{n'} \Big(\sum_{j \in J_{(A,i,i')}} a_{i,j}.(B_{i,j} \| B') + \sum_{j' \in J'_{(A,i,i')}} a'_{i',j'}.(B \| B'_{i',j'}) \Big)$$

where

$$
\begin{aligned}
B &= \bigoplus_{i=0}^{n} \sum_{j=0}^{m_{(i)}-1} a_{i,j}.B_{i,j},\\
B' &= \bigoplus_{i=0}^{n'} \sum_{j=0}^{m'_{(i)}-1} a'_{i,j}.B'_{i,j},\\
J_{(A,i,i')} &= \{j < m_{(i)} \mid a_{i,j} \in A \Rightarrow \forall j' < m'_{(i')} : a_{i,j} \neq a'_{i',j'}\},\ \textit{and}\\
J'_{(A,i,i')} &= \{j' < m'_{(i')} \mid a'_{i',j'} \notin A \Rightarrow \forall j < m_{(i)} : a_{i,j} \neq a'_{i',j'}\}.
\end{aligned}
$$

Proof. By Axioms A2 and A5, which express associativity with respect to $\oplus$ and $+$, respectively, it is allowed to reorder terms. Such steps are not explicitly mentioned in the following proof. We start by applying Axiom A9 to $B\|B'$ which yields

$$\bigoplus_{A\subseteq\mathcal{L}} \left(B|\rangle_{B,A,B'}B'\right) .$$

Thereafter, by applying Axiom A10 we obtain

$$\bigoplus_{A\subseteq\mathcal{L}} \left(B'|\rangle_{B',\mathcal{L}\setminus A,B}B\right) .$$

Applying Axiom A11 repeatedly and using the definitions of B and B' yields

$$\bigoplus_{A\subseteq\mathcal{L}} \bigoplus_{i=0}^{n} \left(B'|\rangle_{B',\mathcal{L}\setminus A,B} \left(\sum_{j=0}^{m_{(i)}-1} a_{i,j}.B_{i,j} \right)\right) .$$

By applying Axiom A10 and thereafter Axiom A11 we obtain

$$\bigoplus_{A\subseteq\mathcal{L}} \bigoplus_{i=0}^{n} \bigoplus_{i'=0}^{n'} \left(\left(\sum_{j=0}^{m_{(i)}-1} a_{i,j}.B_{i,j} \right) |\rangle_{B,A,B'} \left(\sum_{j'=0}^{m'_{(i')}-1} a'_{i',j'}.B'_{i',j'} \right)\right) . \tag{4.6}$$

By applying Axiom A13 (and by suitably reordering using Axiom A5) we get (where also Axiom A7 is applied if necessary)

$$
\begin{aligned}
&\bigoplus_{A\subseteq\mathcal{L}} \bigoplus_{i=0}^{n} \bigoplus_{i'=0}^{n'} \left(\left(\sum_{j\in\{0,..,m'_{(i)}-1\}:a'_{i',j'}\notin A} \left(a'_{i',j'}.\left(B\|B'_{i',j'}\right) >_{a'_{i',j'}} \sum_{j=0}^{m_{(i)}-1} a_{i,j}.B_{i,j} \right)\right) + \right.\\
&\left.\left(\left(\sum_{j=0}^{m_{(i)}-1} a_{i,j}.B_{i,j} \right) |\rangle_{B,A,B'} \left(\sum_{j'\in\{0,..,m'_{(i')}-1\}:a'_{i',j'}\in A} a'_{i',j'}.B'_{i',j'} \right)\right)\right) .
\end{aligned}
$$

By applying Axiom A29, A30 and A31 repeatedly (and also Axiom A7 if necessary) we obtain

$$
\begin{aligned}
&\bigoplus_{A\subseteq\mathcal{L}} \bigoplus_{i=0}^{n} \bigoplus_{i'=0}^{n'} \left(\left(\sum_{j\in J'_{(A,i,i')}:a'_{i',j'}\notin A} a'_{i',j'}.\left(B\|B'_{i',j'}\right) \right) + \right.\\
&\left.\left(\left(\sum_{j=0}^{m_{(i)}-1} a_{i,j}.B_{i,j} \right) |\rangle_{B,A,B'} \left(\sum_{j'\in\{0,..,m'_{(i')}-1\}:a'_{i',j'}\in A} a'_{i',j'}.B'_{i',j'} \right)\right)\right) .
\end{aligned}
$$

Applying Axiom A12 repeatedly yields (by using also Axiom A7)

$$\bigoplus_{A\subseteq\mathcal{L}}\bigoplus_{i=0}^{n}\bigoplus_{i'=0}^{n'}\left(\left(\sum_{j'\in J'_{(A,i,i')}:a'_{i',j'}\notin A} a'_{i',j'}.\left(B\|B'_{i',j'}\right)\right)+\right.$$
$$\left(\left(\left(\sum_{j=0}^{m_{(i)}-1} a_{i,j}.B_{i,j}\right)\setminus\{a'_{i',j'}\mid j'\in\{0,..,m'_{(i')}-1\}\wedge a'_{i',j'}\in A\}\right)|\rangle_{B,A,B'}0\right)+$$
$$\left.\left(\sum_{j'\in J'_{(A,i,i')}:a'_{i',j'}\in A} a'_{i',j'}.B\|B'_{i',j'}\right)\right),$$

where $\tilde{B}\backslash A'$ denotes any linearization of $A'=\{a_1,...a_n\}$, i.e., $\tilde{B}\backslash A'\equiv(\cdots((\tilde{B}\backslash a_1)\backslash a_1)\cdots a_n)$. By reordering (Axioms A4 and A5) we obtain (also using Axiom A7, when necessary)

$$\bigoplus_{A\subseteq\mathcal{L}}\bigoplus_{i=0}^{n}\bigoplus_{i'=0}^{n'}\left(\left(\sum_{j'\in J'_{(A,i,i')}} a'_{i',j'}.\left(B\|B'_{i',j'}\right)\right)+\right.$$
$$\left.\left(\left(\left(\sum_{j=0}^{m_{(i)}-1} a_{i,j}.B_{i,j}\right)\setminus\{a'_{i',j'}\mid j'\in\{0,..,m'_{(i')}-1\}\wedge a'_{i',j'}\in A\}\right)|\rangle_{B,A,B'}0\right)\right).$$

Applying Axioms A24 – A27 yields

$$\bigoplus_{A\subseteq\mathcal{L}}\bigoplus_{i=0}^{n}\bigoplus_{i'=0}^{n'}\left(\left(\sum_{j'\in J'_{(A,i,i')}} a'_{i',j'}.\left(B\|B'_{i',j'}\right)\right)+\left(\left(\sum_{j\in J_{(A,i,i')}} a_{i,j}.B_{i,j}\right)|\rangle_{B,A,B'}0\right)\right).$$

By using a derivation similar to the one starting at (4.6), we obtain

$$\bigoplus_{A\subseteq\mathcal{L}}\bigoplus_{i=0}^{n}\bigoplus_{i'=0}^{n'}\left(\left(\sum_{j'\in J'_{(A,i,i')}} a'_{i',j'}.\left(B\|B'_{i',j'}\right)\right)+\left(\sum_{j\in J_{(A,i,i')}} a_{i,j}.\left(B_{i,j}\|B'\right)\right)\right).$$

By reordering (Axioms A4 and A5), we get

$$\bigoplus_{A\subseteq\mathcal{L}}\bigoplus_{i=0}^{n}\bigoplus_{i'=0}^{n'}\left(\sum_{j\in J_{(A,i,i')}} a_{i,j}.\left(B_{i,j}\|B'\right)+\sum_{j'\in J'_{(A,i,i')}} a'_{i',j'}.\left(B\|B'_{i',j'}\right)\right),$$

as required. □

In the expansion theorem, (i) the scheduler ($A\subseteq\mathcal{L}$) is determined by a resolution of resolvable nondeterminism; (ii) any combination of resolutions of both sides is considered; and (iii) the complete persistent nondeterminism of a component for a is taken if this component is favored by the scheduler or if the current resolution of the other component cannot provide a.

4.5 Related work

An overview on algebraic approaches to nondeterminism is given in [WM97]. Nondeterminism is often interpreted as angelic (chosen positively with respect to a desired property) or demonic (chosen by an adversary). In [MCR04], both nondeterminism interpretations are modeled in a single setting. The angelic/demonic view is orthogonal to our view, leading to the following four interpretations: (i) The task of resolving resolvable, angelic nondeterminism is a *satisfiability* check, i.e., an abstraction has an implementation satisfying the desired property if there is an angelic resolution. (ii) The task of resolving resolvable, demonic nondeterminism is a *satisfaction* check on the abstract level: the property holds if a demonic resolution satisfies it. (iii) Persistent, angelic nondeterminism models the existence of a step such that the desired property holds, whereas (iv) persistent, demonic nondeterminism ensures that all possible next steps satisfy the property. In alternating-time temporal logic [AHK02], interpretations (iii) and (iv) are generalized to multiple actors.

In [BB94], both our choice operators (persistent and resolvable) are used in a process algebra and an operational semantics in terms of μ-automata as well as an axiomatic semantics is given. The difference to our work consists in the semantics of the parallel operator, since the semantics of the parallel operator in [BB94] is based on persistent instead of resolvable scheduling choice, i.e., schedulers behave "randomly" also at the concrete level, which is in most applications, such as schedulers in operating systems, not the case. If the scheduler can prefer different parallel components per action, we cannot apply the usual approach to split parallel composition by using the *left merge* operator, as is also made in [BB94]. Therefore, our axiom system becomes more complicated by using additional operators and by using predicate `nuf`. A further choice operator is presented in [BB94]. The interpretation of this choice operator, which we donte here by $\oplus'$, is similar to our resolvable choice operator $\oplus$, except that resolution can be moved outwards, e.g. $a.(b \oplus' c)$ is equivalent to $a.b \oplus' a.c$, which is not the case for $\oplus$. The choice operator $\oplus'$ has the same expressiveness as $\oplus$ with respect to the describable sets of concrete systems, i.e. replacing $\oplus'$ by $\oplus$, or vice versa, does not change the set of concrete systems that refine the corresponding expressions. A difference arises in the refinement relation between different abstract levels, i.e., the refinement relation between non-concrete systems is different.

In [VDN98], a process algebra having our resolvable choice operator together with a choice operator that is only resolvable with respect to the same action, which harmonizes with ready simulation, is presented. Thus, full persistent nondeterminism is not handled there. The semantics is given as possible worlds, i.e. sets of concrete systems. No parallel operator is considered. This process algebra is extended by recursion in [MC01], but still no parallel composition is considered. Different kinds of choice operators have been introduced in hybrid systems, see, e.g., [BS98], but those choice operators concentrate on underspecification (i.e., resolvable nondeterminism) resulting from time aspects.

In some settings, the scheduler can be restricted by further constraints, like priority or fairness assumptions. They can be interpreted as a refinement of an abstract level where no constraint on the scheduler is enforced. This allowance to describe a more detailed scheduler is orthogonal to the problem of handling the interaction of persistent nondeterminism with the resolvable nondeterminism of an underspecified scheduler.

In probabilistic process algebras, see [LN04] for an overview, choice operators are extended with a distribution determining the way the different operands are favored. Randomized choice operators are special kinds of persistent nondeterminism. Nevertheless, it is important to examine persistent choice operators, which do not contain a distribution, for their own, since sometimes the distribution is not known and sometimes it does not exist at all (cf. Examples 4.2 and 4.3). Note that approaches trying to embed pure persistent nondeterminism into probabilistic settings lead to unnecessarily complex models and, thus, unnecessarily increase the cost of verification.

4.6 Conclusions

We presented the structural operational semantics of a process algebra that handles choice operators corresponding to (i) persistent nondeterminism as well as (ii) resolvable nondeterminism obtained by abstraction from the scheduling of the parallel composition or the renaming/hiding operator. In particular, μ-automata, which have two kinds of transition relations (one for action execution and one for resolution of resolvable nondeterminism), are used as underlying model for the structural operational semantics. In order to avoid any restriction on schedulers, the resolution of resolvable nondeterminism has to be made undone if it was not affected by the previous execution step. A sound and complete axiom system was developed, where a choice operator representing resolvable nondeterminism is used. Note that the increased complexity of our semantics, when compared to the standard semantics, is unavoidable if μ-calculus formulas should be preserved under refinement.

Our operational semantics can straightforwardly be adapted to process algebra employing parallel composition with a CSP-based synchronization [Hoa80], sequential composition, or recursion. The only problem arises for unguarded recursion, i.e. if there exists a variable that is bound but not located behind an action prefix: there the definition of the concretization relation $\rightrightarrows$ yields problems since infinite derivation trees may be generated. This is not very critical because process algebras without unguarded recursion are sufficient for applications.

Chapter 5

Nondeterminism in State Diagrams

The semantics of state diagrams contains resolvable nondeterminism because models are usually underspecified. In this chapter we give a formal operational semantics to state diagrams enriched with an operator expressing persistent nondeterminism, thus requiring a semantic domain supporting both resolvable and persistent nondeterminism. A new such refinement setting is introduced – a variant of μ-automata with a novel refinement relation – and a sound three-valued satisfaction relation for properties expressed in the μ-calculus is given. We also show how existing state machine semantics can be adapted to support persistent nondeterminism.

5.1 Introduction

The preceding chapter has addressed how the two kinds of nondeterminism, resolvable and persistent nondeterminism, appear and should be handled in process algebra semantics. This and the next chapter deal with the second large area of applications considered in this thesis, namely state diagrams. The term *state diagram* shall stand for a variety of different visual languages that have been defined, including "Harel statecharts" [Har87] and state machines as defined in the UML [Obj07]. While this chapter focuses on the two kinds of nondeterminism as they occur in state diagrams, especially UML state machines, the next one will go into more technical detail and present refinement patterns for practical top-down development.

State diagrams serve as a modeling language to describe the behavior of a system in a succinct way. The basic elements, common to all state diagram variants, are states, usually represented by rectangles, which sometimes have rounded corners, and transitions between states. A label on a transition can have (i) an *event* which has to be provided by the environment in order for the transition to be fired, (ii) a *guard* which is a boolean expression required to evaluate to true in order for the transition to be fired, and (iii) an *action* which is performed when the transition fires. The action can usually modify a variable assignment, to which a guard can refer. The common terminology in state diagrams introduces some name conflicts: *actions* in the context of state diagrams are not to be mixed up with *actions* in the context of semantic domains like labeled transition systems: *actions* in the context of semantic domains are called *events* in the context of state diagrams. To avoid confusion, we use the term *SD-action* instead of *action* in the context of state diagrams, and stick to the term *event*, although events in the context of state diagrams are the same thing as actions in the context of semantic domains, both simply denote the elements from our globally fixed set $\mathcal{L}$.

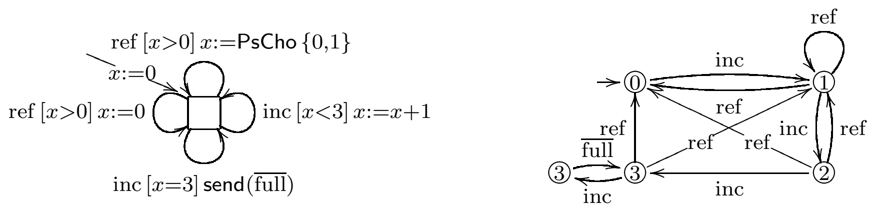


Figure 5.1: An example state diagram and its naïve semantics in terms of transition systems. On the left, we have a state machine with initial value $x = 0$. Transitions are labeled with triples of synchronization event, guard, and execution description; e.g. on event *inc* variable x is incremented by 1 if $x < 3$, and an output event $\overline{\text{full}}$ is sent if $x = 3$. On the right, the naïve semantics in terms of a transition system is shown. The presence of PsCho makes this semantics incorrect for refinements based on bisimulation or ready simulation: with respect to bisimulation, the resolvable nondeterminism cannot be resolved, whereas with respect to ready simulation, the persistent choice can be resolved to always choosing 0 or always choosing 1.

There are many further extensions in various state diagram dialects, e.g. state hierarchy, which allows for state nesting, and orthogonality, which introduces a kind of concurrent behavior.

Underspecification, expressed by resolvable nondeterminism, is common to state diagram models, as e.g. elaborated in [Har87]. For example, underspecification might express that the implementation of an audit logging facility is optional. This nondeterminism is resolvable, ideally in a stepwise fashion, by a controlled choice of implementation alternatives, e.g. the decision not to implement audit logs.

State diagrams should in many cases also be able to express persistent nondeterminism, which can be used to model systems with randomized behavior without knowledge of any probability distributions [RRS06], or fault-tolerant systems. For example, a method header may have an input x of static type `String`, but a complex precondition may enforce semantic constraints on legal inputs, e.g. “at most two coding errors occur in string x”. One then wants to reason about the correctness of an implementation of that method for all inputs with at most two coding errors. The implementor or verifier is not in control over the choice of input (zero, one or two coding errors) meeting these constraints, and seeks assurance against all such choices. In particular, this need to consider all scenarios persists even when the method body is fully implemented – including its communication details with external services.

We present an example that exhibits both resolvable and persistent nondeterminism:

Example 5.1. *The state diagram on the left in Figure 5.1 models part of a system that counts the number of currently occupied sectors of memory, stored in variable x. Setting x to 0 models reformatting, i.e. all sectors are freed up again. This is the initial state, and up to three sectors may be occupied. External nondeterminism is present, e.g., in the choice of events* ref *(reformatting) and* inc *(increment sector count) when $0 < x \leq 3$. Persistent nondeterminism is expressed with the “persistent choice” operator* PsCho $\{0, 1, 2, 3\}$. *On event* ref *and positive value of x, that variable can be set to any value n between 0 and 3 where n is understood to be the leftmost* non-faulty *sector. We cannot control the choice of that value since $x :=$* PsCho $\{0, 1, 2, 3\}$ *models all relevant fault scenarios. (However, to keep this running example small, we will use only a persistent choice between no fault and a fault in the first sector,* *expressible as $x := \mathsf{PsCho}\,\{0,1\}$.) This state machine has resolvable nondeterminism between this persistent choice and that of $x := 0$. The latter transition is risky as it ignores any fault in any modeled sector of memory.*

In the above example, program and faulty memory are combined into a single model. This is advantageous in model checking, when the verification property would not have to specify any fault scenarios, since property specifications become more transparent and reusable and one can compensate for the increased complexity of the model through further model abstraction.

Persistent nondeterminism is not explicitly supported in UML state machines explicitly. But even without such explicit support, it has an implicit presence as soon as the underlying programming language has a "persistent choice" operator as seen in Example 5.1 – which could also be implemented by a random choice. To the best of our knowledge, no formal semantics of UML state machines in the extant literature can express such a construct adequately.

Contribution and outline. Our aim is to support verification on an abstract level for systems that also have persistently nondeterministic behavior, where we mainly focus on state diagram formalisms. Specifically, we use the newly developed refinement setting of ν-automata, which has already been introduced in Section 2.3.2, as semantic domain, and equip ν-automata in Section 5.2 with a 3-valued satisfaction relation between our models and the modal μ-calculus [Koz83, Wil01]. We prove its soundness with respect to our refinement notion, i.e. that satisfied properties are preserved under refinement. In Section 5.3 we discuss state diagram semantics in terms of ν-automata, where Subsection 5.3.1 presents a formal semantics for a simple state diagram variant with a persistent choice operator for arithmetic expressions, and Subsection 5.3.3 discusses how existing formal semantics of more complex state diagram variants can be accommodated to handle persistent choice operators adequately. Next, we sketch in Section 5.4 how our technique can be generalized to state machine variants with, possibly underspecified, randomized choice operators, i.e. operators equipped with probability distributions. Section 5.5 discusses related work and Section 5.6 concludes.

5.2 Satisfaction for ν-automata

We use ν-automata, as introduced in Section 2.3.2, as semantic domain for state diagrams. As discussed in Section 2.3.2, ν-automata are a modification of μ-automata that basically exchanges the order of resolvable and persistent nondeterminism, as they occur in hypertransitions. This has the advantage that, at each state of a ν-automaton, the modeler can decide whether to express persistent or (non-exclusively) resolvable non-determinism, allowing for a more succinct representation of state diagrams with persistent nondeterminism.

In this discussion we consider every label of a transition system to be an *input event*, i.e., to denote an incoming communication. However, *output events* are needed if asynchronous output communication of the system (e.g., label $\overline{\text{full}}$ in Figure 5.1) shall be modeled. If different output events are used, the set of concrete systems of ν-automata has a different constraint for output events: from any state, there may be at most one outgoing transition with an output event. Our refinement notion then has to be amended such that transitions having output events can be removed, as long as one remains present. For details, refer to Section 2.3.3 and Chapter 6.

We give a 3-valued satisfaction relation between ν-automata and the μ-calculus [Koz83]. We choose this logic since persistent choice leads us into the world of branching time, and the μ-calculus is a canonical branching-time logic in which many popular branching-time logics, such as CTL, can be translated. The set of all μ-calculus formulas $\mathcal{F}$ is generated by the following BNF-grammar:

$$\phi \ ::= \ \mathtt{tt} \mid \mathtt{ff} \mid Z \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle e \rangle\, \phi \mid [e]\, \phi \mid \mu Z.\phi \mid \nu Z.\phi$$

where $e \in \mathcal{L}$ and Z is from a μ-calculus variable set $\mathcal{V}ar$. The dual formula $\mathsf{dual}(\phi)$ of a μ-calculus formula ϕ is obtained by replacing $\mathtt{tt}$ by $\mathtt{ff}$, $\wedge$ by $\vee$, $\langle e \rangle$ by $[e]$, μ by ν, and vice versa.

Definition 5.2. *Suppose $M = (S, S^{\mathrm{i}}, \longrightarrow)$ is a ν-automaton. Then, the semantic function $[\![\_]\!]_{\_} : \mathcal{F} \times (\mathcal{V}ar \to \mathcal{P}(S)) \to \mathcal{P}(S)$ with respect to $\mathcal{A}$ is:*

$$[\![\mathtt{tt}]\!]_\rho = S \qquad [\![\mathtt{ff}]\!]_\rho = \emptyset \qquad [\![Z]\!]_\rho = \rho(Z)$$

$$[\![\phi_1 \wedge \phi_2]\!]_\rho = [\![\phi_1]\!]_\rho \cap [\![\phi_2]\!]_\rho \qquad [\![\langle e \rangle\, \phi]\!]_\rho = \{s \in S \mid \forall \ddot{S} \in (s.\xrightarrow{e}) : \ddot{S} \cap [\![\phi]\!]_\rho \neq \emptyset\}$$

$$[\![\phi_1 \vee \phi_2]\!]_\rho = [\![\phi_1]\!]_\rho \cup [\![\phi_2]\!]_\rho \qquad [\![[e]\, \phi]\!]_\rho = \{s \in S \mid \forall \ddot{S} \in (s.\xrightarrow{e}) : \ddot{S} \subseteq [\![\phi]\!]_\rho\}$$

$$[\![\mu Z.\phi]\!]_\rho = \bigcap\{\ddot{S} \mid [\![\phi]\!]_{\rho[Z \to \ddot{S}]} \subseteq \ddot{S}\} \qquad [\![\nu Z.\phi]\!]_\rho = \bigcup\{\ddot{S} \mid \ddot{S} \subseteq [\![\phi]\!]_{\rho[Z \to \ddot{S}]}\}$$

Let ϕ be a closed formula, i.e. every variable Z only occurs in ϕ within the scope of μZ or νZ. Then M satisfies *ϕ, written $M \models \phi$, if $S^{\mathrm{i}} \subseteq [\![\phi]\!]_{\rho_0}$, where ρ_0 maps every variable to the empty set. Furthermore, M* falsifies *ϕ if $M \models \mathsf{dual}(\phi)$.*

The above satisfaction definition is standard (cf. Appendix A), except for the diamond and box operators: to guarantee that satisfaction of $[\![\langle e \rangle\, \phi]\!]_\rho$ under ρ is being preserved at every refinement, all transitions labeled e in the ν-automaton (refinement can remove all but one) must have a suitable candidate in its target set $\ddot{S}$, i.e., $\ddot{S} \cap [\![\phi]\!]_\rho \neq \emptyset$ is required. To guarantee that satisfaction of $[\![[e]\, \phi]\!]_\rho$ under ρ is being preserved at every refinement, all targets $\ddot{S}$ of any transition labeled e in the ν-automata must be suitable candidates: $\ddot{S} \subseteq [\![\phi]\!]_\rho$.

An example illustrates that this satisfaction definition is indeed 3-valued: The ν-automaton $\begin{tikzcd}[sep=small] \arrow[r] & \circ \arrow[r, dash, "e"] \arrow[r, "e"', shift right] & \emptyset \\[-10pt] & & \circ \end{tikzcd}$ neither satisfies nor falsifies the formula $\langle e \rangle\, \mathtt{tt}$ (and this is expected, because the initial state of valid implementation $\to\circ\xrightarrow{e}\circ$ has a successor state via e, and the initial state of the further valid implementation $\to\circ\overset{e}{\text{---}}\emptyset$ has not). However, for concrete ν-automata, the satisfaction definition is 2-valued and corresponds (via embedding h, as defined in the refinement setting of ν-automata) to the standard μ-calculus satisfaction for transition systems (where the dual formula corresponds to negation). Note that M might not satisfy ϕ, although all concrete refinements of M satisfy ϕ. This is, e.g., the case for the above ν-automaton and the formula $\phi = \langle e \rangle\, \mathtt{tt} \vee [e]\, \mathtt{ff}$. However, our satisfaction is sound, as stated by the following theorem:

Theorem 5.3. *Satisfaction is sound for refinement: if a ν-automaton M_1 refines a ν-automaton M_2, then, for any closed formula $\phi \in \mathcal{F}$, we have $M_2 \models \phi \Rightarrow M_1 \models \phi$.*

Proof. First, we reduce satisfaction to the equivalent definition based on parity games [Wil01]. Without loss of generality we assume that every variable is bound at most once in ϕ. Let $\mathcal{V}ar$ be the (finite) set of variables that occur in ϕ, let β be the function mapping elements from $\mathcal{V}ar$ to their corresponding body with respect to ϕ, where subformulas corresponding to variable bindings are replaced by the corresponding variable, e.g., if $\phi = \langle e \rangle\, \mu Y.(([e]\, Y) \wedge (\langle e \rangle\, \mu Z.Z))$ then $\beta(Y) = ([e]\, Y) \wedge (\langle e \rangle\, Z)$. Furthermore, let γ be the function mapping elements from $\mathcal{V}ar$ to their fixed point alternation depth with respect to ϕ. This is well defined since variables are bound at most once in ϕ.

The configurations of the satisfaction game are pairs (s, ϕ') of ν-automaton states and subformulas of ϕ. Subformulas corresponding to variable bindings are replaced by the corresponding binding. The rules of the game are as follows:

- If $\phi' = \texttt{tt}$, then Player 1 wins.
- If $\phi' = \texttt{ff}$, then Player 2 wins.
- If $\phi' = \phi_1 \wedge \phi_2$, then Player 2 may choose $\phi'' \in \{\phi_1, \phi_2\}$ and the next configuration is (s, ϕ'').
- If $\phi' = \phi_1 \vee \phi_2$, then Player 1 may choose $\phi'' \in \{\phi_1, \phi_2\}$ and the next configuration is (s, ϕ'').
- If $\phi' = Z$, then the next configuration is $(s, \beta(Z))$.
- If $\phi' = \langle e \rangle\, \phi''$, then Player 2 chooses $\Theta \in (s.\xrightarrow{e})$, Player 1 chooses $s' \in \Theta$. The next configuration is (s', ϕ'').
- If $\phi' = [e]\, \phi''$, then Player 2 chooses $\Theta \in (s.\xrightarrow{e})$ and $s' \in \Theta$. The next configuration is (s', ϕ'').

A finite play is winning for a player, if the other player has to choose, but cannot. An infinite play η is a win for Player 1 if

$$\max\{\gamma(Z) \mid Z \in \mathcal{V}ar \text{ occurs infinitely often in } \eta\}$$

is even; otherwise, Player 2 wins. A formula is defined to be satisfied if Player 1 has a winning strategy. Note that the corresponding game is a parity game and therefore (i) either Player 1 or Player 2 has a winning strategy and (ii) if there exists a winning strategy for Player 1 (respectively Player 2), then there exists also a history independent winning strategy for Player 1 (respectively Player 2).

It can be shown, analogously to [Wil01], that this game-based definition of satisfaction coincides with the co-inductive one given in Definition 5.2.

Now this game-based definition of satisfaction is used in order to prove the theorem. Let θ be a winning strategy for Player 1 with respect to the satisfaction game $\mathcal{A}_2 \models \phi$, and let R be a ν-refinement relation between $\mathcal{A}_1$ and $\mathcal{A}_2$. Let U be the set of configurations (s_1, ϕ', s_2) such that θ is a winning strategy for (s_2, ϕ') and $(s_1, s_2) \in R$. Here, the usual satisfaction configurations (s_1, ϕ') are already extended by a further component (s_2) that encodes partial history information. The strategies of Player 1 are defined depending on this further component, which may be modified by Player 1 after any step (of Player 1 or Player 2).

A winning strategy for Player 1 on any $(s_1, \phi', s_2) \in U$ is defined as follows: If $\phi' = \phi_1 \wedge \phi_2$ or if $\phi' = \langle e \rangle \phi''$, Player 1 plays according to θ at (s_2, ϕ') and according to the ν-refinement constraint of R, whereas the latter is also used in any other step.

It is easily checked that a play beginning in a configuration of U and played by Player 1 according to the memoryless strategy described above (and the corresponding adaption of s_2) stays within U and is won by Player 1. $\square$

5.3 State diagram semantics

5.3.1 Simple state diagram with a persistent choice operator

The basic modules of UML state machines[1] are variables, states and annotated transitions between them.

Example 5.4. *The state diagram on the left hand side of Figure 5.1 has a single state and four transitions (which are all selfloops in this case). The initial configuration is given by the arrow without a source state, which points at the initial state and defines the initial variable assignment, in which x is assigned value 0. Consider the uppermost selfloop in the figure: it is labeled by a trigger (*ref*), a guard ($x > 0$), and an action ($x := \mathsf{PsCho}\,\{0,1\}$). This transition is enabled, if the environment provides external trigger event* ref *and if the current variable assignment satifies the guard $x > 0$. If the transition is taken, it executes its action, i.e. assigns a new value to x, nondeterministically 0 or 1, and activates the target state of the transition, which in this case is the same state again.*

Our aim is to give a formal semantics to this visual language. For this reason we represent any given state diagram variant by a mathematical object: a simple state diagram SD with respect to input event set $\underline{\mathsf{Ev}}$ and output event set $\overline{\mathsf{Ev}}$ is a tuple $(V, \sigma^{\mathrm{i}}, S_\square, s^{\mathrm{i}}_\square, T)$ comprising of a set of integer-variables V, an initial variable assignment σ^{i}, a set of states $S_\square$, an initial state $s^{\mathrm{i}}_\square \in S_\square$, and a set of transitions T. A transition t consists of a source $\pi_{\mathrm{src}}(t) \in S_\square$, a target $\pi_{\mathrm{tgt}}(t) \in S_\square$, an event $\pi_{\mathrm{ev}}(t) \in \underline{\mathsf{Ev}}$ denoting the external trigger, a guard $\pi_{\mathrm{gd}}(t)$ denoting a necessary condition for enabledness, and an SD-action $\pi_{\mathrm{act}}(t)$, which has to be executed when the transition is taken. For sake of simplicity we restrict the set of guards $g \in G$ to boolean expressions over V. An SD-action consists of a 'do nothing'-action skip, the sending of an event $\mathsf{send}(\overline{e})$, or a variable assignment $v := exp$ – where exp may contain instances of the **per**sistent **cho**ice operator (PsCho) that models persistent nondeterminism for variable assignments. Formally, an SD-action α is given by the following BNF-grammar:

$$\alpha ::= \mathsf{skip} \mid \mathsf{send}(\overline{e}) \mid v := exp \qquad\qquad exp ::= v \mid n \mid exp \otimes exp \mid \mathsf{PsCho}\,N$$

where v is a variable from V, n is an integer-constant, $\overline{e}$ is an element from the set of output events $\overline{\mathsf{Ev}}$ of the state machine, $\otimes$ is any standard binary operator on integers such as addition or subtraction, and N is a (finite) set of integer-constants – the possible return values of the persistent choice operator. The set of all SD-actions is denoted by $\mathcal{A}ct$.

[1]Note that it is possible to introduce persistent nondeterminism to "Harel statechart" semantics in a similar way. But since the semantics of triggering events is different (sets of events in "Harel statecharts" in contrast to single events in UML state machines), this simple but cumbersome adaptation is omitted.

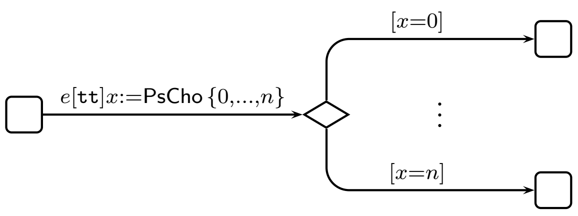


Figure 5.2: Persistent nondeterminism with respect to states using a choice pseudostate.

Remark 5.5. *So far we only modeled persistent nondeterminism with respect to variable assignments. In order to model persistent nondeterminism with respect to state machine states, one can use a* choice pseudostate *as it exists in UML state machines. It allows different target states, depending on the result of a previous action execution. Therefore, one can execute* $x := \mathsf{PsCho}\,\{0,\dots,n\}$ *on a transition leading to a choice pseudostate and, originating from the choice pseudostate, one can have several transitions to different states, each with a guard depending on* x*. This is illustrated in Figure 5.2.*

We first discuss the semantics of state diagrams informally. At each step, a single event e is dispatched from the event pool, where the order of dispatching is left open in [Obj07]. This event enables a transition if (i) its source is currently active, (ii) e is its trigger, and (iii) its guard evaluates to true with respect to the current variable assignment. Among all enabled transitions, one is chosen, its corresponding SD-action is executed, and the active state becomes its target instead of its source. If no transition is currently enabled for e, different interpretations are possible:

U ('no enabled transition' corresponds to underspecification): Arbitrary behavior is possible after consuming e. A transition in state machines can be added as long as its guard conjoined with the guard of any other transition from the same source and with the same event is unsatisfiable. This harmonizes with the refinement patterns of [Rum96].

D ('no enabled transition' corresponds to discarding): e is consumed without executing any SD-action. This corresponds to the standard interpretation of UML state machines.[2]

We first provide a formal semantics to the **U**-approach. The formal semantics to the **D**-approach is given in Example 5.6.

$\mathcal{V}$ denotes the set of all possible variable assignments of V. The guards are evaluated with respect to a variable assignment in the usual way, by a function eval : $G \times \mathcal{V} \to \{\top, \bot\}$ where $\top$ corresponds to the boolean value *true* and $\bot$ to *false*. Function $\mathsf{calc} : \mathcal{A}ct \times \mathcal{V} \to \mathcal{P}(\mathcal{V})$ yields the set of updated variable assignments, which is in general a singleton set except in the case of the usage of the persistent choice operator. Formally:

$$\mathsf{calc}(\alpha,\sigma) = \begin{cases} \{\sigma[v \mapsto n] \mid n \in \widetilde{\mathsf{calc}}(exp,\sigma)\} & \text{if } \alpha \text{ is } v := exp \\ \{\sigma\} & \text{otherwise} \end{cases}$$

[2]In UML state machines, an event can also be declared to be deferred at a state, in which case the triggering or discarding of this event is moved to a subsequent state where it is no longer deferred and becomes active.

Table 5.1: Transition derivation rules.

$$\text{(a)}\ \frac{t \in T \quad \mathrm{eval}(\pi_{\mathrm{gd}}(t), \sigma) = \top \quad \pi_{\mathrm{act}}(t) = \mathsf{send}(\overline{e})}{(\pi_{\mathrm{src}}(t), \sigma) \xrightarrow{\pi_{\mathrm{ev}}(t)} \{(\overline{e}, \pi_{\mathrm{tgt}}(t), \sigma)\}} \qquad \text{(b)}\ \frac{\overline{e} \in \overline{\mathsf{Ev}}}{(s_\square, \sigma) \xrightarrow{\overline{e}} \emptyset} \qquad \text{(c)}\ \frac{e \in \underline{\mathsf{Ev}} \cup \overline{\mathsf{Ev}}}{\begin{array}{c} c_r \xrightarrow{e} \{c_r\} \\ c_r \xrightarrow{e} \emptyset \end{array}}$$

$$\text{(d)}\ \frac{t \in T \quad \mathrm{eval}(\pi_{\mathrm{gd}}(t), \sigma) = \top \quad \forall \overline{e} \in \overline{\mathsf{Ev}} : \pi_{\mathrm{act}}(t) \neq \mathsf{send}(\overline{e})}{(\pi_{\mathrm{src}}(t), \sigma) \xrightarrow{\pi_{\mathrm{ev}}(t)} \{(\pi_{\mathrm{tgt}}(t), \sigma') \mid \sigma' \in \mathsf{calc}(\pi_{\mathrm{act}}(t), \sigma)\}} \qquad \text{(e)}\ \frac{}{(\overline{e}, s_\square, \sigma) \xrightarrow{\overline{e}} \{(s_\square, \sigma)\}}$$

$$\text{(f)}\ \frac{\forall t \in T : (\pi_{\mathrm{src}}(t) = s_\square \wedge \pi_{\mathrm{ev}}(t) = e) \Rightarrow \mathrm{eval}(\pi_{\mathrm{gd}}(t), \sigma) = \bot}{(s_\square, \sigma) \xrightarrow{e} \{c_r\}} \qquad \text{(g)}\ \frac{e \in \underline{\mathsf{Ev}} \cup \overline{\mathsf{Ev}} \setminus \{\overline{e}\}}{(\overline{e}, s_\square, \sigma) \xrightarrow{e} \emptyset}$$

with

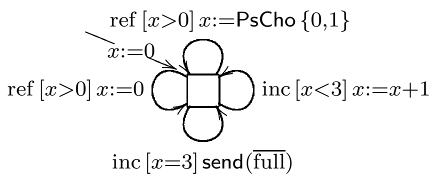
$$\widetilde{\mathsf{calc}}(exp, \sigma) = \begin{cases} \{\sigma(v)\} & \text{if } exp \text{ is } v \in V \\ \{n\} & \text{if } exp \text{ is constant } n \\ \{n_1 \otimes n_2 \mid \forall j \in \{1,2\} : n_j \in \widetilde{\mathsf{calc}}(exp_j, \sigma)\} & \text{if } exp \text{ is } exp_1 \otimes exp_2 \\ N & \text{if } exp \text{ is } \mathsf{PsCho}\, N \end{cases}$$

The formal semantics is given as a ν-automaton: its state space consists of (i) configurations – SD-states combined with variable assignments –, (ii) intermediate states for modeling the sending of output events, and (iii) an additional state c_r, which corresponds to a ν-automaton state abstracting anything. Formally, its state space is $(S_\square \times \mathcal{V}) \cup (\overline{\mathsf{Ev}} \times S_\square \times \mathcal{V}) \cup \{c_r\}$. Its only initial state is the initial SD-state combined with the initial variable assignment $(s^{\mathrm{i}}_\square, \sigma^{\mathrm{i}})$. Its hypertransitions are given by the derivation rules in Table 5.1, where its underlying set of labels is $\underline{\mathsf{Ev}} \cup \overline{\mathsf{Ev}}$. Here, the firing of a transition that sends an output event leads (by adapting the active SD-state) to an intermediate state (a), which only sends exactly the corresponding output event (e) and nothing else (g). Output events can only be sent in such intermediate states (b). If the SD-action is different from a send-action, a hypertransition subsuming all the possible outcomes with respect to calc is derived (d). In case no transition is enabled for an event (f), a transition to the state c_r, which abstracts everything ensured by (c), is derived. Note that (c) encodes two rules: one where $c_r \xrightarrow{e} \{c_r\}$ is the conclusion, another where $c_r \xrightarrow{e} \emptyset$ is the conclusion – both having the same premise.

Example 5.6. *Figures 5.3 and 5.4 illustrate the semantics of two state machines, where the first is the same state machine as in Figure 5.1 and the second is a refinement of the first. Here, the state machine constraint "all other events discarded" means that the* **D***-approach is taken, in which case rule (f) is replaced by*

$$\text{(f}'\text{)}\ \frac{\forall t \in T : (\pi_{\mathrm{src}}(t) = s_\square \wedge \pi_{\mathrm{ev}}(t) = e) \Rightarrow \mathrm{eval}(\pi_{\mathrm{gd}}(t), \sigma) = \bot}{(s_\square, \sigma) \xrightarrow{e} \{(s_\square, \sigma)\}}$$

where self loops (corresponding to discarding) – rather than the reaching of the state that abstracts everything – are introduced. Note that adding the constraint "all other events discarded" is a refinement step, as well as the removal of transitions, as long as for every variable assignment the set of enabled events remains the same.


ref [x>0] x:=PsCho {0,1}
x:=0
ref [x>0] x:=0
inc [x<3] x:=x+1
inc [x=3] send($\overline{\text{full}}$)

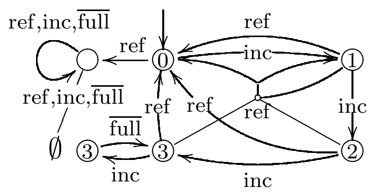
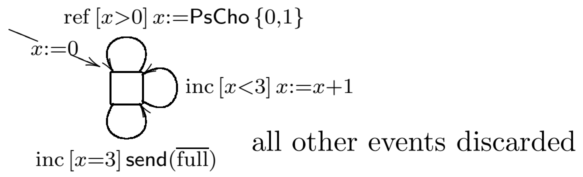

ref,inc,$\overline{\text{full}}$
ref
ref
inc
ref,inc,$\overline{\text{full}}$
ref
ref
ref
inc
$\overline{\text{full}}$
$\emptyset$
inc
inc

Figure 5.3: The state machine of Figure 5.1 and its semantics in terms of ν-automata. In this semantics, a set of hypertransitions sharing the same targets and label are combined using a small circle. Furthermore, all states that do not have a drawn outgoing transition for some $e \in \mathcal{L}$ have an implicit transition with label e to $\emptyset$. Here, the set of labels is $\mathcal{L} = \{\text{ref}, \text{inc}, \overline{\text{full}}\}$.


ref [x>0] x:=PsCho {0,1}
x:=0
inc [x<3] x:=x+1
inc [x=3] send($\overline{\text{full}}$)
all other events discarded

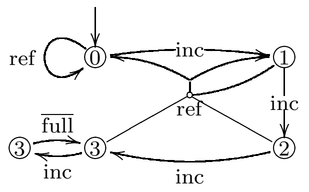


Figure 5.4: A refinement of the state machine of Figure 5.3, where the transition on the left has been removed and non-present events are interpreted as "discarded", together with its semantics in terms of ν-automata.

5.3.2 Succinctness of ν-automata

The order of persistent and resolvable nondeterminism defined in μ-automata, introduced in Subsection 2.3.2, does not fit well the requirements for a semantics of state diagrams with persistent choice operator: ν-automata with their ready simulation approach of refinement allow direct use of variable assignments as ν-automaton states. This is possible, because underspecification is modeled similarly in state diagrams and ν-automata: removing transitions in the state diagram directly corresponds to removing transitions in the semantic model. This correspondence does not exist between state diagrams and μ-automata. Consequently, it is necessary to use more complex μ-automaton states: in addition to variable assignments, extra information is needed for expressing every possible choice of taken transitions for each SD-action. This exponentially blows up the state space:

Example 5.7. *A state diagram and its semantics in terms of ν-automata are given in Figure 5.5. The semantics in terms of μ-automata is significantly more complex, and for this reason not depicted. Its set of states is $\{0,1\} \times \mathcal{P}(\{1,\ldots,n\})$, its set of initial states is $\{0\} \times \mathcal{P}(\{1,\ldots,n\})$, and its transition relation is*

$$\bigcup_{j\in\{1,2\},\ddot{C}\in\mathcal{P}(\{1,\ldots,n\})} (\{((j,\ddot{C}),\{0\}\times\mathcal{P}(\{1,\ldots,n\}))\} \cup \{((j,\ddot{C}),\{1\}\times\mathcal{P}(\{1,\ldots,n\})) \mid j \in \ddot{C}\}$$

In different settings, μ-automata may be more suitable than ν-automata, as is illustrated e.g. by Chapter 4 of this thesis, which gives an operational semantics to a process algebra with parallel composition and a persistent choice operator.

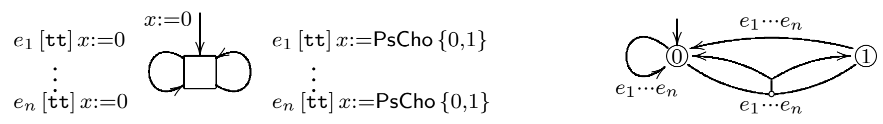


Figure 5.5: Example of the succinctness of ν-automata. Depicted is a state machine together with its semantics in terms of ν-automata. A semantics in terms of μ-automata would be significantly more complex and is therefore not depicted.

5.3.3 Adaptation of existing semantics

We sketch how existing formal semantics of state diagrams can be adapted such that they can handle also persistent choice operators. Since our extension of state diagrams with persistent choice is orthogonal to their run-to-completion steps, concurrency, and hierarchies, those notions are still adequately handled in the presence of persistent choice operators.

We assume that the formal semantics is given in terms of transition systems that are obtained by derivation rules (small-step semantics), as e.g. in [FS07c, vdB02]. Also, we assume that the derivation rules use an evaluation function calc for the SD-actions, i.e. calc maps an SD-action together with a variable assignment to a variable assignment (and possibly additional information). Such an evaluation function exists in [FS07c] under the name calc, in [vdB02] it is not explicitly mentioned, because that paper abstracts from variables.

As the first transformation, function calc is adapted to SD-actions containing also persistent choice operators. This is achieved by changing the range of calc to sets (rather than elements) of variable assignments.

Next, the derivation rules are adapted such that hypertransitions are obtained. This is done by replacing the target of the derived transition by the set consisting of the original target only. When calc is used for the derivation, the target set contains all possible variations with respect to the result of calc. For example, rule

$$\frac{\alpha \neq \mathsf{skip} \quad \mathsf{calc}(\alpha, \sigma) = (\ell, \alpha', \sigma')}{(\sigma, \mathcal{A}, \mathsf{do}, H, \alpha, \ddot{s}_{\square}, \beta, T, \ddot{T}) \xrightarrow{\ell} (\sigma', \mathcal{A}, \mathsf{do}, H, \alpha', \ddot{s}_{\square}, \beta, T, \ddot{T})}$$

taken from [FS07c] – describing the execution of an SD-action – is transformed into

$$\frac{\alpha \neq \mathsf{skip}}{(\sigma, \mathcal{A}, \mathsf{do}, H, \alpha, \ddot{s}_{\square}, \beta, T, \ddot{T}) \xrightarrow{\ell} \{(\sigma', \mathcal{A}, \mathsf{do}, H, \alpha', \ddot{s}_{\square}, \beta, T, \ddot{T}) \mid (\ell, \alpha', \sigma') \in \mathsf{calc}(\alpha, \sigma)\}}$$

Note that in the semantics of [FS07c] the evaluation function also yields (i) a label (e.g. the sending of an output event) which will be the same for all elements of $\mathsf{calc}(\alpha, \sigma)$, (ii) an SD-action which remains to be executed, and (iii) a variable assignment.

Finally, absence of outgoing transitions for a given label is handled such that it harmonizes with ν-refinement. If at a state there is no outgoing transition with an output event (see Section 5.2), then a corresponding transition pointing to the empty set is added. The same is done for events whenever no event can be accepted in the current state, e.g. if some computation must be finished first. The absence of outgoing transitions – for an event e in a semantic state s where events can be accepted – is handled by a case analysis on the different interpretations introduced in Section 5.3.1:

U: Add a new state c_r to the ν-automaton, and add the additional two derivation rules (c) from Figure 5.1. Thus, c_r represents a state that is refined by any ν-automaton state. A transition labeled with e from s to $\{c_r\}$ is also added.[3]

D: A self-loop from s labeled with event e is added.

5.4 Random choice operators

A random choice operator can be used to express a probability distribution. Probability distributions are a form of persistent choice. For example, the probabilistic choice of a successor state of state s in a Markov chain expresses a nondeterministic choice that remains to be nondeterministic each time state s is visited. Of course, this persistence is controllable to some degree through the use of familiar tools for stochastic processes.

5.4.1 ν-automata and random choice

We sketch how ν-automata can be adapted to support random choice operators with an underlying distribution: first, they are extended by moving from power sets to distributions as transition targets, i.e., $\longrightarrow \ \subseteq S \times \mathcal{L} \times \mathsf{Dist}(S)$ where $\mathsf{Dist}(S)$ denotes the possible discrete distributions over the set S (total functions f from S to $[0,1]$ such that $\sum_{s \in S} f(s) = 1$). Furthermore, the refinement notion is adapted to handle probabilities by lifting the refinement relation to distributions, i.e. *probabilistic automata* with *strong simulation* [Seg06] as refinement notion. Then calc has to be adapted such that distributions rather than sets of states are obtained. The adaptation of the existing semantics is made similar to that in Section 5.3.3. Note that in such a probabilistic setting, satisfaction is defined over probabilistic logics, like PCTL [HJ94], rather than over the μ-calculus.

5.4.2 Underspecified random choice

This approach can also deal with randomized choice that is underspecified. Exact knowledge of distributions is often not a realistic assumption and does not facilitate top-down-development, because abstract systems already need to specify exact probabilities. Therefore, approaches emerged [JL91] where only sets or intervals of allowed probabilities are given. We can accommodate this as follows:

Probabilistic automata (PA) are extended by moving from distributions to sets of possible distributions as transition targets, i.e., $\longrightarrow \ \subseteq S \times \mathcal{L} \times \mathcal{P}([0,1])^S$, with $\mathcal{P}([0,1])^S$ denoting the set of total functions from S to $\mathcal{P}([0,1])$, which yields a generalization of *probabilistic specification systems* (PSS) [JL91], where the transition relation is a subset of $S \times \mathcal{P}([0,1]) \times S$. Furthermore the refinement notion has to be adequately adapted, similar to the treatment in [JL91]. This new class of models, which we denote by PSSA, has more expressive power than PSS in the sense that more sets of concrete refinements can be described via refinement. This is informally argued in Figure 5.6. Note that PSSA has also more expressive power than PA, first with respect to strong simulation – since probabilistic intervals cannot be finitely described

[3]An additional transition to the empty set is not needed here since *input enabledness* is usually guaranteed. Otherwise, e.g. in the context of event deferral, such a transition has to be added.

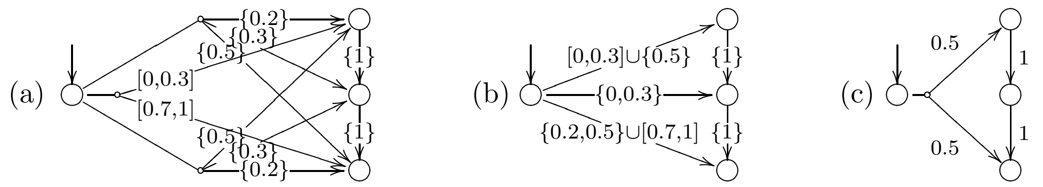


Figure 5.6: Examples of probabilistic models. In particular, (a) a ν-automaton extended with sets of probabilities, (b) a probabilistic specification system, which has all concrete refinements of (a) and is minimal with respect to refinement among those models having all concrete refinements of (a); and (c) a deterministic probabilistic automaton being a concrete refinement of (b) but not of (a). All three models have the same single event-label, which is therefore omitted.

by nondeterminism – and, second with respect to *strong probabilistic simulation* [Seg06], since this has already less expressive power than PSS.

In order to give the formal semantics, calc has to be adapted such that sets of distributions rather than distributions are obtained. The adaption of the existing semantics is then made similar to that in Section 5.3.3. The exact development of the theoretical foundation of this class of models, including precise refinement and satisfaction definitions, is a topic for future work.

5.5 Related work

In Section 5.3.2 we argued why μ-automata [JW95] are not suitable as semantic models for state diagrams having a persistent choice operator. Further abstract models that can express persistent as well as resolvable nondeterminism are, e.g., (disjunctive) modal transition systems [LT88, LX90], mixed transition systems [DGG97], generalized Kripke modal transition systems [SG04], hypermixed Kripke structures [FH06], and modal automata [DN05]. These models have additional may-transitions meaning that refinements may have those transitions but not necessarily so. Hyper Kripke modal transition systems [SG06] interpret must-hypertransitions analogously to the μ-refinement approach, whereas may-hypertransitions are interpreted analogously to the ν-refinement approach. None of these models are suitable as semantic models for state diagrams, by the same arguments as put forward for μ-automata. Note that the additional may-transitions do not help one to define the semantics, since there is no concept in state diagrams that supports such kind of modeling directly (in state diagram refinements, one transition per label has to remain, i.e. not all transitions can be removed [Rum96]). A comparison to probabilistic models is already made in Section 5.4. In [KNP06], the authors use probabilistic games to differentiate between probabilistic (persistent) and resolvable nondeterminism. Their model is close to PSSA based on *strong probabilistic simulation.*

An overview of existing formal semantics of UML state machines is given in [CD05a], where 26 different approaches are compared – and none of them features persistent choice. [FS07c], which due to later publication is not referenced in [CD05a], gives a complete formal semantics of UML 2.0 state machines without persistent choice. As already mentioned, refinement patterns of statecharts are presented in [Rum96], where the underlying formal semantics are streams and therefore no persistent nondeterminism is handled. In [DC03] state machines are mapped

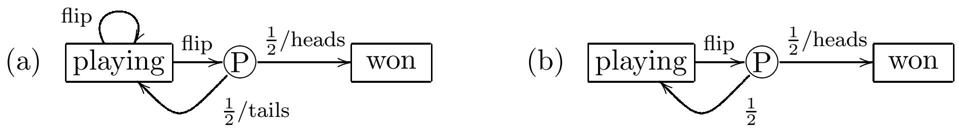


Figure 5.7: Illustration of an unexpected refinement in the setting of [JHK02]. In particular, (a) a state machine having probabilistic pseudo-states and (b) one of its possible (unwanted) refinements with respect to the formal semantics of [JHK02], where heads but no tails can occur.

to terms of CSP and so a stronger semantics is given by considering trace- and failure-based refinement.

The work most closely related to ours is [JHK02] where a formal semantics in terms of probabilistic automata based on strong simulation is given to state machines having an additional randomized pseudo-state. Their semantic mapping corresponds more to a *run-to-completion* step rather than to the firing of transitions. This leads to the problems that (i) their refinement notion – implicitly given via probabilistic automata – is only sound but imprecise, i.e. there are unexpected semantic refinements, illustrated in Figure 5.7, (ii) a well-formedness condition – that every variable may only be changed by one of the firing transitions – is needed, and (iii) their state space is exponential in the number of enabled probabilistic transitions. None of these problems occur in our definition of probabilistic pseudo-states via choice pseudo-states (see the discussion in Remark 5.5).

5.6 Conclusions

This chapter presented state diagram semantics as an application domain for the refinement setting of ν-automata, which has been newly introduced in Chapter 2. It employs hypertransitions, its refinement notion generalizes ready simulation, and it is suitable as semantic domain for systems that feature or benefit from persistent nondeterminism as well as resolvable nondeterminism. In particular, a formal semantics for a simple state machine language was given in terms of ν-automata, and it was sketched how existing semantics of state machines can be adapted if persistent choice operators are added to the underlying programming language. A compositional 3-valued satisfaction definition over the μ-calculus was given for ν-automata and proved to be sound for property verification.

It is future work to establish tool support based on the ν-automata semantics for satisfaction and refinement. In particular, it should be examined how ν-automata can be used in the context of abstraction in order to speed up verification.

Chapter 6

Verifiable Top-Down Development of State Diagrams

This chapter defines a variant of state diagrams which has (i) a restricted set of features to avoid semantic ambiguities and (ii) an extended syntax in order to support top-down development. It is given a precise operational semantics using a refinement setting with input and output actions. A set of so-called refinement patterns is established which can be used to perform refinement steps in the development process, where these steps are correct by construction since the patterns are proved sound. The results pave the way for the development and implementation of a CASE (computer-aided software engineering) tool supporting verified refinement in state diagrams.

6.1 Introduction

State diagrams are a widely-used visual formalism for the description of complex reactive systems. They were originally developed by David Harel [Har87] and since then emerged in a variety of different variants, including an own diagram type in the Unified Modeling Language (UML) [Obj07], which made them even more attractive for industrial use. State diagrams manage to compactly represent complex behavior, primarily thanks to the concepts of *hierarchy*, nesting substates into other states, and *orthogonality*, representing parallel execution. Already for the basic concept of hierarchy, different state diagram variants have different semantic interpretations: whereas the original Harel statecharts give outer transitions priority over inner transitions, the UML variant gives priority to deeper nested transitions. The different dialects are characterized by a diversity of further syntactic and semantic differences [CD05b] and, even for any given variant like the UML one, the semantics is ambiguous and not agreed upon [FSKdR05]. Safety-critical systems should therefore be modeled using a "safe" subset of state diagrams, avoiding notations with an unclear semantics, as is proposed e.g. in [HRW07] and [SAM96]. This subset should have a formal semantics such that properties of the modeled system can be proved.

Even though the range of features may be restricted to a subset, real-world state diagrams can get considerably large, with many orthogonal regions and deep state nesting. Consequently, there is a need for a structured development approach following the top-down paradigm: first a general layout of the model is designed and afterwards its different components are refined, as independent from one another as possible. None of the current state diagram variants formally supports such a kind of top-down development, giving a formal semantics to each stage of the development process. In a tool, one may envisage that refinement steps are offered to the modeler from a menu of different model transformations that are correct by construction, which we call *refinement patterns*. These transformations are always guaranteed to correspond to a semantic refinement.

Contribution and outline. Section 6.2 defines a variant of state diagrams that is designed for correct and verifiable top-down development. Section 6.3 then gives its formal, refinement-sensitive semantics. Section 6.4 presents an expressive and proven-to-be-sound collection of refinement patterns. Its use is illustrated by an example in Section 6.5 that shows how these refinement patterns can be applied to design a non-trivial model of an elevator controller. Possible extensions of our state diagram variant are discussed in Section 6.6, and Section 6.7 concludes.

6.2 Syntax

In contrast to some of the previous chapters, we employ a precise definition of a *state diagram* here. A state diagram with respect to an input event set $\underline{\mathsf{Ev}}$ and an output event set $\overline{\mathsf{Ev}}$ consists of states and transitions between them. Every transition is labeled with one input event, say $\underline{e} \in \underline{\mathsf{Ev}}$, and a set of output events, say $\overline{E} \subseteq \overline{\mathsf{Ev}}$. The idea is that a transition fires, if $\underline{e}$ is provided by the environment. Then, the events in $\overline{E}$ are sent to the environment and the current state changes. In this respect, output events are *locally controlled*, i.e. the modeled system has control over their occurrence, whereas input events happen from the outside and the system only reacts on it. The state machine represented by a diagram is *input-enabled* [Lyn96] in the sense that any possible input event is always handled, if necessary by implicit discarding.

Important features of state diagrams are *state hierarchy* and *orthogonality*. This is realized in our variant by requiring every state to have at least one subregion, which again may contain states and transitions. If a state has several subregions, they are interpreted as orthogonal, i.e. behavior in different subregions is executed in parallel. In our variant, a substate may only be entered from the outside via an *entry point*, analogously it may only be left via an *exit point*, which avoids semantically-hard-to-handle *inter-level* transitions. So-called *links* from entry points and to exit points define which substates get active after an entry and need to be active in order to enable the exit, respectively. This not only allows one to model the behavior of inter-level transitions but also subsumes *fork* and *join* transitions.

Due to orthogonality, it is possible that an input event fires more than one transition. More specifically, it fires a maximal non-conflicting and priority-respecting set of enabled transitions. For instance, two enabled transitions in different regions of a common super-state will both fire, but if two transitions originate from two states where one is nested in the other, only the outer transition will fire. This kind of priority, following Harel statecharts rather then UML state machines, is necessary for proper top-down development: transitions in subcomponents – to be specified later in the development process – do not change the behavior of a more global context. If there is no transition enabled for the provided event, it is discarded.

To keep our technical presentation as readable as possible, we will not consider variables or transition guards. However, in Section 6.6 we will discuss how variables and further features can be added and handled semantically.

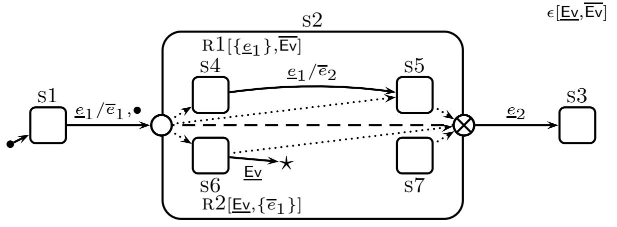

s2
$\epsilon[\underline{\mathsf{Ev}},\overline{\mathsf{Ev}}]$
$\mathrm{R1}[\{\underline{e}_1\},\overline{\mathsf{Ev}}]$
s4
$\underline{e}_1/\overline{e}_2$
s5
s1
$\underline{e}_1/\overline{e}_1,\bullet$
$\underline{e}_2$
s3
$\underline{\mathsf{Ev}}$
s6
s7
$\mathrm{R2}[\underline{\mathsf{Ev}},\{\overline{e}_1\}]$

Figure 6.1: Graphical notations. In this example, $\underline{\mathsf{Ev}} = \{\underline{e}_1, \underline{e}_2\}$ and $\overline{\mathsf{Ev}} = \{\overline{e}_1, \overline{e}_2\}$. States are represented by rounded rectangles with their names attached outside. Regions are separated by dashed lines and region names are specified inside the region, together with possible event constraints. Transitions are represented by solid arrows, labeled with one input event and a set of output events. Entry and exit points are drawn as $\bigcirc$ and $\otimes$, respectively. Both entry and exit links are represented by dotted arrows. Initial state $\mathtt{s1}$ is marked by a small arrow originating from a dot. Open events of a state are specified as labels of an arrow leading to $\star$.

Since we aim at top-down development we extend the syntax by notations for underspecification. A state can have *open* input events (specified in a component opn), which means that transitions from this state with such events may still be added (and removed). An entry or exit point is *open* with respect to a region r, if it has no entry links leading to region r, or exit links originating from region r, respectively. In this case, the choice of entry or exit links, with respect to this region, is left open.

Furthermore, a region can define *input* and *output event constraints*, specifying that this region can only have transitions labeled with the allowed input and output events (as given in components $\underline{\mathsf{con}}$ and $\overline{\mathsf{con}}$, see below). Finally, a special output event $\bullet$ in a transition label allows one to add arbitrary output events to this transition.

Refer to Figure 6.1 for graphical notations for our variant of state diagrams. If a region's event constraint is omitted in a drawing, the event constraints of its parent region are assumed (in case of the top-level region ϵ, $\underline{\mathsf{Ev}}$ and $\overline{\mathsf{Ev}}$ are assumed). Substates automatically inherit their parent state's open events; in the drawing, these may be omitted for the substate.

Formally, a *state diagram* with respect to an input event set $\underline{\mathsf{Ev}}$ and an output event set $\overline{\mathsf{Ev}}$ is a tuple $\mathcal{D} = (\mathcal{S}, \mathcal{S}^0, \mathcal{P}^\bigcirc, \mathcal{P}^\otimes, \mathcal{R}, \mathsf{par}, \mathcal{T}, \mathfrak{L}^\bigcirc, \mathfrak{L}^\otimes, \underline{\mathsf{con}}, \overline{\mathsf{con}}, \mathsf{opn})$, where $\mathcal{S}$ is a set of *states*, $\mathcal{S}^0 \subseteq \mathcal{S}$ is a subset of *initial states*, $\mathcal{P}^\bigcirc$ is a set of *entry points*, $\mathcal{P}^\otimes$ is a set of *exit points*, $\mathcal{R}$ is a set of *regions*[1] (with distinguished outermost region $\epsilon \in \mathcal{R}$), $\mathsf{par} : (\mathcal{V} \cup \mathcal{R} \setminus \{\epsilon\}) \to (\mathcal{V} \cup \mathcal{R})$ (with $\mathcal{V} \overset{\text{def}}{=} \mathcal{S} \cup \mathcal{P}^\bigcirc \cup \mathcal{P}^\otimes$ being the so-called vertices) is the *parent function* defining the hierarchy of vertices and regions, $\mathcal{T} \subseteq (\mathcal{S} \cup \mathcal{P}^\otimes) \times \underline{\mathsf{Ev}} \times (\mathcal{P}(\overline{\mathsf{Ev}} \cup \{\bullet\})) \times (\mathcal{S} \cup \mathcal{P}^\bigcirc)$ is the set of *transitions*, $\mathfrak{L}^\bigcirc \subseteq \mathcal{P}^\bigcirc \times (\mathcal{S} \cup \mathcal{P}^\bigcirc)$ defines the *entry links*, $\mathfrak{L}^\otimes \subseteq (\mathcal{S} \cup \mathcal{P}^\otimes) \times \mathcal{P}^\otimes$ defines the *exit links*, $\underline{\mathsf{con}} : \mathcal{R} \to \mathcal{P}(\underline{\mathsf{Ev}})$ defines the *input event constraint* of a region, $\overline{\mathsf{con}} : \mathcal{R} \to \mathcal{P}(\overline{\mathsf{Ev}})$ defines the *output event constraint* of a region, and $\mathsf{opn} : \mathcal{S} \to \mathcal{P}(\underline{\mathsf{Ev}})$ defines the *open events* of a state. A state diagram is required to satisfy the well-formedness conditions listed in Table 6.1, in which $\mathsf{state}(v)$, for $v \in \mathcal{V}$, denotes the state of v, i.e.

$$\mathsf{state}(v) \overset{\text{def}}{=} \begin{cases} \mathsf{par}(v) & \text{if } v \in \mathcal{P}^\bigcirc \cup \mathcal{P}^\otimes \\ v & \text{if } v \in \mathcal{S} \end{cases},$$

[1]Sets $\mathcal{S}$, $\mathcal{P}^\bigcirc$, $\mathcal{P}^\otimes$, and $\mathcal{R}$ are required to be pairwise disjoint.

Table 6.1: Well-formedness conditions for a state diagram.

The parent of every state is a region:	$\forall s \in \mathcal{S} : \mathsf{par}(s) \in \mathcal{R}$
The parent of every point is a state:	$\forall p \in \mathcal{P}^{\bigcirc} \cup \mathcal{P}^{\otimes} : \mathsf{par}(p) \in \mathcal{S}$
The parent of any non-top-level region is a state:	$\forall r \in \mathcal{R} \setminus \{\epsilon\} : \mathsf{par}(r) \in \mathcal{S}$
Every state has (at least) one region:	$\forall s \in \mathcal{S} : \exists r \in \mathcal{R} : \mathsf{par}(r) = s$
The parent function defines a tree with root ϵ and only branches of finite length:	$\forall x \in \mathcal{V} \cup \mathcal{R} : \exists n \in \mathbb{N} : \mathsf{par}^n(x) = \epsilon$
The initial states are closed wrt. parent states:	$\forall s \in \mathcal{S}^0 : \mathsf{par}(\mathsf{par}(s)) \in \mathcal{S}^0$
No two initial states are in conflict:	$\forall s_1, s_2 \in \mathcal{S}^0 : \neg\mathsf{conflict}(s_1, s_2)$
Event constraints are inherited by super-regions:	$\forall r \in \mathcal{R} : \underline{\mathsf{con}}(r) \subseteq \underline{\mathsf{con}}(\mathsf{par}(\mathsf{par}(r))) \wedge$ $\overline{\mathsf{con}}(r) \subseteq \overline{\mathsf{con}}(\mathsf{par}(\mathsf{par}(r)))$
Open transitions are inherited by substates:	$\forall s \in \mathcal{S} : \mathsf{par}(s) \neq \epsilon \Rightarrow$ $\mathsf{opn}(s) \supseteq \mathsf{opn}(\mathsf{par}(\mathsf{par}(s)))$
Transitions stay within a region:	$\forall (v, \underline{e}, \overline{E}, v') \in \mathcal{T} : \mathsf{reg}(v) = \mathsf{reg}(v')$
Entry links enter a direct child-region:	$\forall (v, v') \in \mathfrak{L}^{\bigcirc} : \mathsf{state}(v) = \mathsf{par}(\mathsf{reg}(v'))$
Exit links originate from a direct child-region:	$\forall (v, v') \in \mathfrak{L}^{\otimes} : \mathsf{par}(\mathsf{reg}(v)) = \mathsf{state}(v')$
Transitions conform to event constraints:	$\forall (v, \underline{e}, \overline{E}, v') \in \mathcal{T} : \underline{e} \in \underline{\mathsf{con}}(\mathsf{reg}(v)) \wedge$ $\overline{E} \subseteq \overline{\mathsf{con}}(\mathsf{reg}(v)) \cup \{\bullet\}$
Open transitions conform to event constraints:	$\forall s \in \mathcal{S} : \mathsf{opn}(s) \subseteq \underline{\mathsf{con}}(\mathsf{par}(s))$

and $\mathsf{reg} \overset{\text{def}}{=} \mathsf{par} \circ \mathsf{state}$ is a short notation for the surrounding region of a state or point. Furthermore, for $x \in \mathcal{V} \cup \mathcal{R}$, let $\mathsf{chdn}(x)$ be the set of direct children of x, i.e. $\mathsf{chdn}(x) \overset{\text{def}}{=} \{y \in \mathcal{V} \cup \mathcal{R} \mid \mathsf{par}(y) = x\}$; let $\mathsf{chdn}_{\mathcal{R}}(x)$ be the set of direct child-regions of x, i.e. $\mathsf{chdn}_{\mathcal{R}}(x) \overset{\text{def}}{=} \mathsf{chdn}(x) \cap \mathcal{R}$; and let $\mathsf{sub}(x)$ be the set of all sub-elements of x, i.e. $\mathsf{sub}(x) \overset{\text{def}}{=} \{y \in \mathcal{V} \cup \mathcal{R} \mid \exists n \in \mathbb{N} : \mathsf{par}^n(y) = x\}$. Finally, relation $\mathsf{conflict}$ defines whether two states s_1 and s_2 are *in conflict*, i.e. must not be active at the same time. This is the case if both are contained in a common region but one is not a substate of the other.

$$\begin{aligned}\mathsf{conflict}(s_1, s_2) \quad \overset{\text{def}}{\Leftrightarrow} \quad & (s_1 \in \mathsf{sub}(\mathsf{reg}(s_2)) \wedge s_1 \notin \mathsf{sub}(s_2)) \vee \\ & (s_2 \in \mathsf{sub}(\mathsf{reg}(s_1)) \wedge s_2 \notin \mathsf{sub}(s_1)).\end{aligned}$$

We use the convention that t stands for a transition $(v, \underline{e}, \overline{E}, v')$, i.e. it implicitly defines its components v, $\underline{e}$, $\overline{E}$, and v'. Accordingly, t_1 stands for $(v_1, \underline{e}_1, \overline{E}_1, v'_1)$, etc. Analogously, l stands for a link (v, v'), and l_1 stands for (v_1, v'_1), etc.

Example 6.1. *We illustrate the definitions with respect to the state diagram presented in Figure 6.1. Here, the state set $\mathcal{S}$ is $\{\text{S}1, \ldots, \text{S}7\}$. The initial state set $\mathcal{S}^0$ is the singleton $\{\text{S}1\}$. State* S2 *has one entry point on the left and one exit point on the right. From the entry point we have entry links to* S4*,* S5*, and* S6*; exit links to the exit point originate from* S5*,* S6*, and* S7*. State* S2 *has two orthogonal regions* R1 *and* R2*, whereas all other state just have a single default region, whose name is not given in the drawing. Region* R1 *has input event constraint $\{e_1\}$ and output event constraint $\overline{\mathsf{Ev}}$; region* R2 *has input event constraint $\underline{\mathsf{Ev}}$ and output event constraint $\{\overline{e}_1\}$. The only state that has open events is* S6*; it is open with respect to all input events $\underline{\mathsf{Ev}}$. The parent function defines, e.g., the following parent relationships: state* S6 *has as parent region* R2 *which in turn has as parent state* S2 *which in turn has as parent the outermost region ϵ. The entry point and the exit point both have parent* S2*. The set of direct children of state* S2*, $\mathsf{chdn}(\text{S}2)$, contains the entry point, the exit point, and the two regions* R1*,* R2*. The set*

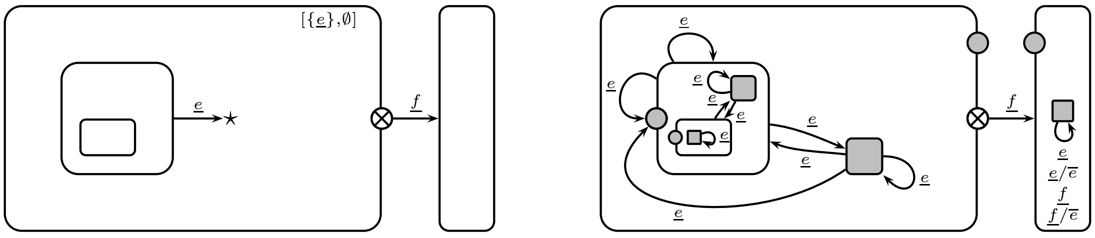


Figure 6.2: Illustration of transformation reduce. Here, $\underline{\mathsf{Ev}} = \{\underline{e}, \underline{f}\}$ and $\overline{\mathsf{Ev}} = \{\overline{e}\}$. The left state diagram is transformed to the right reduced state diagram. In the latter, introduced helper components are shaded gray for easier readability. The helper state of the outermost region ϵ, which is not reachable, and its outgoing transitions are omitted in this drawing.

of all sub-elements, $\mathsf{sub}(\mathrm{S2})$*, also contains the states* $\{\mathrm{S4}, \ldots, \mathrm{S7}\}$ *and their contained default regions. Examples for conflicting state pairs are* $(\mathrm{S4},\ \mathrm{S5})$ *and* $(\mathrm{S1},\ \mathrm{S5})$*. State pairs* $(\mathrm{S2},\ \mathrm{S4})$ *and* $(\mathrm{S4},\ \mathrm{S6})$ *are not in conflict, i.e. these are allowed to be active at the same time.*

6.3 Semantics

We give the semantics of a given state diagram in two steps. First, we eliminate underspecification notations (open input events, open output event sets, and event constraints) by expressing them using standard notations like nondeterministic transitions. This yields a *reduced state diagram* for which the semantics in terms of an I/O-transition system is easier to define, via structural operational semantics (SOS) rules.

6.3.1 Transformation into a reduced state diagram

A state diagram is called *reduced* if its event constraints are as general as possible, i.e. $\forall r \in \mathcal{R} : \underline{\mathsf{con}}(r) = \underline{\mathsf{Ev}} \wedge \overline{\mathsf{con}}(r) = \overline{\mathsf{Ev}}$, it does not have states that are open with respect to certain events, i.e. $\forall s \in \mathcal{S} : \mathsf{opn}(s) = \emptyset$, and all output event sets are closed, i.e. $\forall (v, \underline{e}, \overline{E}, v') \in \mathcal{T} : \bullet \notin \overline{E}$. Usually, the three last components of a reduced state diagram are omitted.

A state diagram is transformed into a reduced state diagram using function reduce, explained in the following. Open output event sets are eliminated by several transitions, one transition for each output event set that is allowed by the notation and the given event constraint. Open input events are eliminated using additional helper states in each region, $\square_r$, which offer all transitions allowed by the given event constraints. The state (in region r) that has open input events then is equipped with transitions to $\square_r$, labeled by these input events and any subset of (by event constraint) permitted output events. Furthermore, it gets transitions to all other states in its region, and it also has to descend into possible substates of these. For formalizing this we need to introduce an additional open entry state $\bigcirc_s$ for each state. Figure 6.2 shows an example of transformation reduce. Note that the openness with respect to $\underline{e}$ in the example still allows discarding of event $\underline{e}$, simulated by a loop to an entry point of the same state, which then, since it is open, allows re-entry into the current (sub-)configuration.

Formally,

$$\mathsf{reduce} : (\mathcal{S}, \mathcal{S}^0, \mathcal{P}^{\bigcirc}, \mathcal{P}^{\otimes}, \mathcal{R}, \mathsf{par}, \mathcal{T}, \mathfrak{L}^{\bigcirc}, \mathfrak{L}^{\otimes}, \underline{\mathsf{con}}, \overline{\mathsf{con}}, \mathsf{opn}) \mapsto (\widehat{\mathcal{S}}, \mathcal{S}^0, \widehat{\mathcal{P}}^{\bigcirc}, \mathcal{P}^{\otimes}, \widehat{\mathcal{R}}, \widehat{\mathsf{par}}, \widehat{\mathcal{T}}, \widehat{\mathfrak{L}}^{\bigcirc}, \mathfrak{L}^{\otimes}),$$

where

- $\widehat{\mathcal{S}}$ gets new helper states $\Box_r$ for each region:
$$\widehat{\mathcal{S}} \stackrel{\text{def}}{=} \mathcal{S} \mathbin{\dot\cup} \{\Box_r \mid r \in \mathcal{R}\},$$
- $\widehat{\mathcal{P}}^{\bigcirc}$ contains new default entry points $\bigcirc_s$ for each state:
$$\widehat{\mathcal{P}}^{\bigcirc} \stackrel{\text{def}}{=} \mathcal{P}^{\bigcirc} \mathbin{\dot\cup} \{\bigcirc_s \mid s \in \mathcal{S}\},$$
- the helper states $\Box_r$ get default regions ρ_r:
$$\widehat{\mathcal{R}} \stackrel{\text{def}}{=} \mathcal{R} \mathbin{\dot\cup} \{\rho_r \mid \Box_r \in \widehat{\mathcal{S}}\},$$
- the parent function arranges the new helper components' positions in the hierarchy tree:
$$\widehat{\mathsf{par}} \stackrel{\text{def}}{=} \mathsf{par} \cup \{(\Box_r, r) \mid r \in \mathcal{R}\} \cup \{(\bigcirc_s, s) \mid s \in \mathcal{S}\} \cup \{(\rho_r, \Box_r) \mid r \in \mathcal{R}\},$$
- there are new transitions with all (by event constraint) permitted subsets of output events and all targets inside the same region (i) from states with an open event, which then defines the input event of the transition, and (ii) from helper states $\Box_r$, in which case the input event may be any event allowed by the event constraint of the surrounding region. Furthermore, we replace open output event sets by all event sets allowed by the notation and the given event constraint:
$$\begin{aligned} \widehat{\mathcal{T}} \stackrel{\text{def}}{=} \; & \mathcal{T} \cup \{(v, \underline{e}, \overline{E}, v') \mid v \in \widehat{\mathcal{S}} \cup \widehat{\mathcal{P}}^{\bigcirc} \cup \mathcal{P}^{\otimes} \wedge v' \in \widehat{\mathcal{S}} \cup \widehat{\mathcal{P}}^{\bigcirc} \wedge \mathsf{reg}(v') = \mathsf{reg}(v) \wedge \\ & \overline{E} \subseteq \overline{\mathsf{con}}(\mathsf{reg}(v)) \wedge (\underline{e} \in \mathsf{opn}(v) \vee (\underline{e} \in \underline{\mathsf{con}}(\mathsf{reg}(v)) \wedge v = \Box_{\mathsf{par}(v)}))\} \cup \\ & \{(v, \underline{e}, \overline{E}, v') \mid \exists \overline{E}' : (v, \underline{e}, \overline{E}', v') \in \mathcal{T} \wedge \bullet \in \overline{E}' \wedge \overline{E}' \setminus \{\bullet\} \subseteq \overline{E} \subseteq \overline{\mathsf{con}}(\mathsf{reg}(v))\}. \end{aligned}$$

Here, $\Box_r$, ρ_r, and $\bigcirc_s$ are fresh symbols. Note that several newly introduced transitions in the transformation could have been left out in case they are "overridden" by another transition leaving a less deeply nested state. This would lead to a smaller and semantically equivalent reduced state diagram but complicates the definition of reduce.

6.3.2 Structural operational semantics

As semantic model we employ the refinement setting $\mathbb{IOTS}$, as introduced in Section 2.3.3. It fits the requirements in the context of state diagrams well, because (i) state diagrams have input and output events and (ii) state diagrams are input-enabled (every event is either processed or discarded, but never rejected). For better differentiation between state diagram states and I/O-transition system states, we call I/O-transition system states *configurations* in the following.

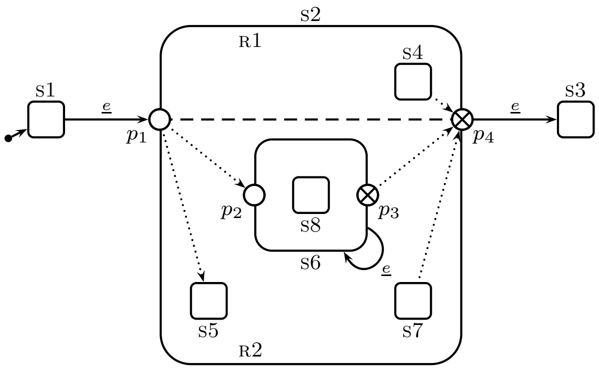


Figure 6.3: Example of a reduced state diagram.

We define the semantics of a reduced state diagram. Since the openness notations for open events in states, event constraints and open output event sets have already been reduced, we do not need to take care of these. However, entry and exit points can still be open, if there are no links from/to one of their parent's regions. Formally, predicates $\mathsf{opn}_{\bigcirc}$ and $\mathsf{opn}_{\otimes}$ define, for a point p and a region r of their parent, whether p is open with respect to r:

$$\mathsf{opn}_{\bigcirc}(p,r) \stackrel{\text{def}}{=} \forall v \in p.\mathfrak{L}^{\bigcirc} : \mathsf{reg}(v) \neq r \qquad \mathsf{opn}_{\otimes}(p,r) \stackrel{\text{def}}{=} \forall v \in \mathfrak{L}^{\otimes}.p : \mathsf{reg}(v) \neq r$$

Example 6.2. *In the state diagram of Figure 6.3, entry point p_1 is open with respect to region* R1*, but not open with respect to region* R2*. Exit point p_3 is open with respect to the single default region of state* S6*. Exit point p_4 is not open, neither with respect to region* R1 *nor with respect to region* R2*.*

Predicate $\longrightarrow_{\otimes} \subseteq \mathcal{P}(\mathcal{S}) \times \mathcal{V}$ defines whether v is a lead-out vertex of state set S (written $S \longrightarrow_{\otimes} v$), i.e. whether S, as a set of currently active states, enables vertex v (typically an exit point) for the firing of outgoing transitions. $\longrightarrow_{\otimes}$ is defined via two SOS rules: the first states that every $s \in S$ is enabled in this sense, and the second states that an exit point is enabled if every region has at least one enabled vertex linked to the exit point:

$$\frac{s \in S}{S \longrightarrow_{\otimes} s} \qquad \frac{p \in \mathcal{P}^{\otimes} \quad \forall r \in \mathsf{chdn}_{\mathcal{R}}(\mathsf{state}(p)) : \exists v \in (\mathfrak{L}^{\otimes}.p) \cap (S.\longrightarrow_{\otimes}) : \mathsf{reg}(v) = r}{S \longrightarrow_{\otimes} p}$$

Example 6.3. *In the state diagram of Figure 6.3, p_4 is a lead-out vertex of* {S2, S4, S7}*. p_4 is* not *a lead-out vertex of* {S2, S4, S6} *or* {S2, S4, S6, S8}.

If there are open exit points along which a vertex v could become a lead-out vertex of a state set S, it is possible but not guaranteed that in an implementation v is a lead-out vertex of S. This possibility is captured by predicate $\dashrightarrow_{\otimes} \subseteq \mathcal{P}(\mathcal{S}) \times \mathcal{V}$, in the definition of which it is also allowed to leave states using open exit points:

$$\frac{s \in S}{S \dashrightarrow_{\otimes} s} \qquad \frac{\begin{array}{c} p \in \mathcal{P}^{\otimes} \quad \forall r \in \mathsf{chdn}_{\mathcal{R}}(\mathsf{state}(p)) : \\ ((\exists v \in (\mathfrak{L}^{\otimes}.p) \cap (S.\dashrightarrow_{\otimes}) : \mathsf{reg}(v) = r) \vee \mathsf{opn}_{\otimes}(p,r)) \end{array}}{S \dashrightarrow_{\otimes} p}$$

Note that $\longrightarrow_{\otimes} \subseteq \dashrightarrow_{\otimes}$.

Example 6.4. *In the state diagram of Figure 6.3, we have* $\{\mathrm{S2}, \mathrm{S4}, \mathrm{S6}, \mathrm{S8}\} \dashrightarrow_{\otimes} p_4$.

Open entry points (with respect to region r) correspond to those linking to each possible target in r. Multiple links into a region express nondeterminism which has to be resolved upon state entry. This is formalized by *resolutions* that define, for each pair of entry point and region, a vertex linked from this entry point and located in this region. Formally, a *resolution* is a function $\varrho : \{(p, r) \mid p \in \mathcal{P}^{\bigcirc} \wedge r \in \mathsf{chdn}_{\mathcal{R}}(\mathsf{state}(p))\} \to \mathcal{V}$ satisfying

$$\forall p \in \mathcal{P}^{\bigcirc}, r \in \mathsf{chdn}_{\mathcal{R}}(\mathsf{state}(p)) : ((p, \varrho(p, r)) \in \mathfrak{L}^{\bigcirc} \vee \mathsf{opn}_{\bigcirc}(p, r)) \wedge \mathsf{reg}(\varrho(p, r)) = r.$$

Example 6.5. *A possible resolution, say* ϱ_1*, for the state diagram of Figure 6.3 maps* $(p_1, \mathrm{R1})$ *to* S4 *and* $(p_1, \mathrm{R2})$ *to* S5*. Another possible resolution, say* ϱ_2*, maps* $(p_1, \mathrm{R1})$ *to* S4 *and* $(p_1, \mathrm{R2})$ *to* p_2*. Every resolution maps* p_2 *together with the single default region of* S6 *to* S8*.*

We define $v \xrightarrow{\varrho}_{\bigcirc} v'$ iff v' is a successor state that gets active if a transition leading to v fires, given resolution ϱ. Relation $\longrightarrow_{\bigcirc}$ is defined by two SOS rules, where the first states that v itself becomes active and the second descends into subregions according to the given resolution:

$$\frac{v \in \mathcal{S} \cup \mathcal{P}^{\bigcirc}}{v \xrightarrow{\varrho}_{\bigcirc} v} \qquad \frac{v \xrightarrow{\varrho}_{\bigcirc} p \quad r \in \mathsf{chdn}_{\mathcal{R}}(\mathsf{state}(p))}{v \xrightarrow{\varrho}_{\bigcirc} \varrho(p, r) \quad v \xrightarrow{\varrho}_{\bigcirc} \mathsf{state}(\varrho(p, r))}$$

Example 6.6. *Considering the resolutions* ρ_1 *and* ρ_2 *of Example 6.5, we get* $p_1 \xrightarrow{\varrho_1}_{\bigcirc} \mathrm{S4}$ *and* $p_1 \xrightarrow{\varrho_1}_{\bigcirc} \mathrm{S5}$*. We do* not *get* $p_1 \xrightarrow{\varrho_1}_{\bigcirc} p_2$*, but* $p_1 \xrightarrow{\varrho_2}_{\bigcirc} p_2$*.*

A transition $t = (v, \underline{e}, \overline{E}, v')$ is must-enabled or may-enabled with respect to given current active states S and a provided event $\underline{e}'$, if the event matches the transition's event and the source of t is a guaranteed or possible lead-out vertex of S, respectively.

$$\begin{aligned} \mathsf{must\text{-}enabled}(t, S, \underline{e}') &\overset{\text{def}}{\Leftrightarrow} \underline{e} = \underline{e}' \wedge S \longrightarrow_{\otimes} v \\ \mathsf{may\text{-}enabled}(t, S, \underline{e}') &\overset{\text{def}}{\Leftrightarrow} \underline{e} = \underline{e}' \wedge S \dashrightarrow_{\otimes} v \end{aligned}$$

Example 6.7. *In the state diagram of Figure 6.3, we have* $\mathsf{must\text{-}enabled}((p_4, \underline{e}, \emptyset, s_3), \{\mathrm{S2}, \mathrm{S4}, \mathrm{S7}\}, \underline{e})$ *and* $\mathsf{may\text{-}enabled}((p_4, \underline{e}, \emptyset, s_3), \{\mathrm{S2}, \mathrm{S4}, \mathrm{S6}, \mathrm{S8}\}, \underline{e})$*.*

One transition $t_1 = (v_1, \underline{e}_1, \overline{E}_1, v_1')$ has priority over another transition $t_2 = (v_2, \underline{e}_2, \overline{E}_2, v_2')$, $\mathsf{prio}(t_1, t_2)$, if the state of v_2 is a substate of the state of v_1. Note that for technical reasons we define $\mathsf{prio}(t_1, t_2)$ to also hold if $v_1 = v_2$ and even if $t_1 = t_2$.

$$\mathsf{prio}(t_1, t_2) \overset{\text{def}}{\Leftrightarrow} \mathsf{state}(v_2) \in \mathsf{sub}(\mathsf{state}(v_1))$$

Example 6.8. *In the state diagram of Figure 6.3,* $(p4, \underline{e}, \emptyset, \mathrm{S3})$ *has priority over* $(\mathrm{S6}, \underline{e}, \emptyset, \mathrm{S6})$*.*

A set of transitions Γ is fireable with respect to given current active states S and a provided event $\underline{e}$ if (i) all transitions $t \in \Gamma$ are may-enabled, (ii) there is no must-enabled transition with higher priority, and (iii) must-enabled transitions not in Γ are in priority conflict with a transition in Γ.

$$\begin{aligned} \mathsf{fireable}(\Gamma, S, \underline{e}') \overset{\text{def}}{\Leftrightarrow}\ & \forall t \in \Gamma : (\mathsf{may\text{-}enabled}(t, S, \underline{e}') \wedge \\ & (\forall t_1 \in \mathcal{T} : v_1 = v \vee \neg\mathsf{must\text{-}enabled}(t_1, S, \underline{e}') \vee \neg\mathsf{prio}(t_1, t))) \wedge \\ & (\forall t_2 \in \mathcal{T} \setminus \Gamma : \neg\mathsf{must\text{-}enabled}(t_2, S, \underline{e}') \vee \exists t_3 \in \Gamma : \mathsf{prio}(t_2, t_3)) \end{aligned}$$

Example 6.9. *In the state diagram of Figure 6.3, we have* $\mathsf{fireable}(\{(p4,\underline{e},\emptyset,\mathrm{s3})\},\{\mathrm{s2},\mathrm{s4},\mathrm{s7}\},\underline{e})$ *and* $\mathsf{fireable}(\{(p4,\underline{e},\emptyset,\{\mathrm{s2},\mathrm{s4},\mathrm{s6},\mathrm{s8}\}\},\underline{e})$.

Now the semantics of a given reduced state diagram in terms of an I/O-transition system is defined as follows. The configurations are pairs of current active states and those output events that are still to be performed before the next state transition, i.e. $C \overset{\text{def}}{=} \{(S,\overline{E}) \mid S \subseteq \mathcal{S} \wedge \overline{E} \subseteq \overline{\mathsf{Ev}}\}$, and the initial configuration is defined to be $c^0 \overset{\text{def}}{=} (\mathcal{S}^0,\emptyset)$. The transition relation $\longmapsto$ is defined by the following SOS rules. Rule (FIRE) selects a fireable set of transitions and consumes the corresponding input event. It leads over to the succeeding state configuration, leaving all substates of states left by the transition and entering successor states, defined by any given resolution resolving entry point nondeterminism. Furthermore, it collects all output events to be generated. Rule (EXEC) makes sure these events are thereafter generated one by one (in nondeterministic order). Finally, rule (DISCARD) introduces self-loops, if an input event cannot be taken, which corresponds to event discarding.

$$\frac{\underline{e} \in \underline{\mathsf{Ev}} \quad \Gamma \subseteq \mathcal{T} \quad \mathsf{fireable}(\Gamma,S,\underline{e}) \quad f:\Gamma \to \mathcal{P}(\mathcal{V}) \quad \forall t \in \Gamma : \exists \varrho : f(t) = (v'.\xrightarrow{\varrho}_{\bigcirc})}{(S,\emptyset) \overset{\underline{e}}{\longmapsto} \left(\left(S \setminus \bigcup_{t\in\Gamma} \mathsf{sub}(\mathsf{state}(v))\right) \cup \left(\bigcup_{t\in\Gamma} f(t)\right), \bigcup_{t\in\Gamma} \overline{E}\right)} \quad (\textsc{fire})$$

$$\frac{\underline{e} \in \underline{\mathsf{Ev}} \quad \forall t \in \mathcal{T} : \neg\mathsf{must\text{-}enabled}(t,S,\underline{e})}{(S,\emptyset) \overset{\underline{e}}{\longmapsto} (S,\emptyset)} \quad (\textsc{discard})$$

$$\frac{\overline{e} \in \overline{E}}{(S,\overline{E}) \overset{\overline{e}}{\longmapsto} (S,\overline{E} \setminus \{\overline{e}\})} \quad (\textsc{exec})$$

Proposition 6.10. *In the semantics, all reachable configurations are conflict-free and closed with respect to parent states.*

Proof. Closedness with respect to parent states: By well-formedness, the initial configuration is closed with respect to parent states. Given a configuration that is closed with respect to parent states, the only rule to change the currently active states is rule (FIRE). Any firing transition removes all children of the source state of the transition and adds a parent-closed set of states inside the same region (this is because transitions are not allowed to change regions). Thus closedness with respect to parent states is preserved.

Conflict-freedom: By well-formedness, the initial configuration is conflict-free. Given a conflict-free configuration, assume that there are a successor configuration and active states s_1, s_2 with $\mathsf{conflict}(s_1,s_2)$. Without loss of generality, let $s_1 \in \mathsf{sub}(\mathsf{reg}(s_2)) \wedge s_1 \notin \mathsf{sub}(s_2)$. Since all parent states of s_1 are active as well (closedness with respect to parent states), there is an active state s_1' such that $\mathsf{reg}(s_1') = \mathsf{reg}(s_2)$. s_1' or s_2 must have (newly) become active in the last step, otherwise the previous configuration would have had a conflict as well. Now, we distinguish several cases and show that all of them lead to contradictions:

- Both s_1' and s_2 got active by transitions, or one got active by a transition and the other was active before: this is impossible since then the previous configuration would have had a conflict as well, because transitions do not change regions.
- Both s_1' and s_2 got active by an entry link: then, they must have become active via a common entry point, because otherwise two transitions would have fired in a common region (predicate prio would have held). However, in the common entry point, the resolution can only choose a single successor, so this is impossible.

- One of the two states s_1' and s_2 got active by an entry link and the other is unchanged or got active by a transition: then, the previous configuration would have had a conflict. $\square$

6.4 Refinement patterns

A *refinement pattern* is a model transformation that is correct by construction, i.e. when applying it to a model satisfying the pattern's preconditions, the resulting model is in fact a refinement of the original. Here, we present an expressive set of refinement patterns in a very general notation, using structural operational semantics rules to define a pattern relation $\rightsquigarrow$. In practice, one will want to use a user-friendly, higher-level transformation language, e.g. (hierarchical) graph transformations [DHP02] or the *Epsilon* transformation language [KPP08]. By using an SOS-like rule notation we chose a very general notation that can easily be translated into any higher-level language of choice. The names of some of the rules are underlined. These refinement patterns are actually equivalence patterns, i.e. they could be inverted and would still yield a refinement pattern. Not all such inversions are given here because we restrict ourselves to those patterns that are especially useful in the context of top-down development. This collection of patterns is far from "complete"; instead we aim for practical utility.

Set $\mathsf{fresh}(\mathcal{D})$, often used in preconditions of rules, denotes a set of fresh symbols, i.e. it satisfies $\mathsf{fresh}(\mathcal{D}) \cap (\mathcal{V} \cup \mathcal{R}) = \emptyset$.

(as) **<u>Add state.</u>** This pattern adds a state s, together with a default contained region r, to an existing surrounding region $\overline{r}$.

$$\frac{s, r \in \mathsf{fresh}(\mathcal{D}) \qquad \overline{r} \in \mathcal{R}}{\begin{array}{c}\mathcal{D} \rightsquigarrow (\mathcal{S} \cup \{s\}, \mathcal{S}^0, ..., \mathcal{R} \cup \{r\}, \\ \mathsf{par}[r \mapsto s, s \mapsto \overline{r}], ..., \underline{\mathsf{con}}[r \mapsto \underline{\mathsf{con}}(\overline{r})], \overline{\mathsf{con}}[r \mapsto \overline{\mathsf{con}}(r)], \mathsf{opn}[s \mapsto \underline{\mathsf{con}}(\overline{r})])\end{array}}$$

Remember that $f[a \mapsto b]$ denotes function f except that a is mapped to b, i.e. $(f \setminus \{(a, f(a))\}) \cup \{(a, b)\}$ if $a \in A$, and $f \cup \{(a, b)\}$ otherwise.

(ar) **<u>Add region.</u>** This pattern adds an additional region r to a non-initial state s, which only has entry points that are open with respect to each region. r inherits the event constraints of an existing region $\overline{r}$.

$$\frac{r \in \mathsf{fresh}(\mathcal{D}) \qquad s \in \mathcal{S} \setminus \mathcal{S}^0 \qquad \forall p \in \mathsf{chdn}(s) \cap \mathcal{P}^{\circ} : p.\mathfrak{L}^{\circ} = \emptyset \qquad \overline{r} \in \mathsf{chdn}(s)}{\mathcal{D} \rightsquigarrow (..., \mathcal{R} \cup \{r\}, \mathsf{par}[r \mapsto s], ..., \underline{\mathsf{con}}[r \mapsto \underline{\mathsf{con}}(\overline{r})], \overline{\mathsf{con}}[r \mapsto \overline{\mathsf{con}}(\overline{r})], ...)}$$

Note that no actual behavior may be fixed for state s in which an additional region is to be added, which is made sure by the required non-existence of specified (i.e., non-open) entry points or initial states. This is natural in the top-down development process which first fixes regions (together with their event constraints) and then continues to implement the contents of the regions.

The restriction that initial states cannot be added regions can be compensated by a simple, top-level initial state that then enters a more complex hierarchical configuration in the first (possibly silent) step.

(a$\bigcirc$) **Add open entry point.** This pattern adds an entry point p to state s. It is open with respect to all regions of s because it does not have any links to subvertices yet.

$$\frac{p \in \mathsf{fresh}(\mathcal{D}) \qquad s \in \mathcal{S}}{\mathcal{D} \rightsquigarrow (..., \mathcal{P}^{\bigcirc} \cup \{p\}, ..., \mathsf{par}[p \mapsto s], ...)}$$

(a$\otimes$) **Add open exit point.** This pattern adds an exit point p to state s. It is open with respect to all regions of s because it does not have any links from subvertices yet.

$$\frac{p \in \mathsf{fresh}(\mathcal{D}) \qquad s \in \mathcal{S}}{\mathcal{D} \rightsquigarrow (..., \mathcal{P}^{\otimes} \cup \{p\}, ..., \mathsf{par}[p \mapsto s], ...)}$$

(s$\bigcirc$) **Specify entry links.** This pattern closes an entry point p that is open with respect to region r by defining the set of linked subvertices in r, namely V. Even though it is closed in this sense, some nondeterminism still remains existent as long as $|V| > 1$.

$$\frac{p \in \mathcal{P}^{\bigcirc} \qquad r \in \mathcal{R} \qquad \mathsf{par}(r) = \mathsf{state}(p) \qquad \mathsf{opn}_{\bigcirc}(p, r) \qquad V \subseteq \{v \in \mathcal{S} \cup \mathcal{P}^{\bigcirc} \mid \mathsf{reg}(v) = r\}}{\mathcal{D} \rightsquigarrow (..., \mathfrak{L}^{\bigcirc} \cup \{(p, v) \mid v \in V\}, ...)}$$

(s$\otimes$) **Specify exit links.** This pattern closes an exit point p that is open with respect to region r by defining the set of linked subvertices in r, namely V.

$$\frac{p \in \mathcal{P}^{\otimes} \qquad r \in \mathcal{R} \qquad \mathsf{par}(r) = \mathsf{state}(p) \qquad \mathsf{opn}_{\otimes}(p, r) \qquad V \subseteq \{v \in \mathcal{S} \cup \mathcal{P}^{\otimes} \mid \mathsf{reg}(v) = r\}}{\mathcal{D} \rightsquigarrow (..., \mathfrak{L}^{\otimes} \cup \{(v, p) \mid v \in V\}, ...)}$$

(r$\bigcirc$) **Remove open entry point.** This pattern removes an entry point p that is open with respect to all regions and connects transitions and links leading to p with the state of p.

$$\frac{p \in \mathcal{P}^{\bigcirc} \qquad p.\mathfrak{L}^{\bigcirc} = \emptyset}{\begin{array}{c}\mathcal{D} \rightsquigarrow (..., \mathcal{P}^{\bigcirc} \setminus \{p\}, ..., \{t \mid (t \in \mathcal{T} \land v' \neq p) \lor ((v, \underline{e}, \overline{E}, p) \in \mathcal{T} \land v' = \mathsf{state}(p))\}, \\ \{l \mid (l \in \mathfrak{L}^{\bigcirc} \land v' \neq p) \lor ((v, p) \in \mathfrak{L}^{\bigcirc} \land v' = \mathsf{state}(p))\}, ...)\end{array}}$$

(r$\otimes$) **Remove open exit point.** This pattern removes an exit point p that is open with respect to all regions and connects transitions and links originating from p with the state of p.

$$\frac{p \in \mathcal{P}^{\otimes} \qquad \mathfrak{L}^{\otimes}.p = \emptyset}{\begin{array}{c}\mathcal{D} \rightsquigarrow (..., \mathcal{P}^{\otimes} \setminus \{p\}, ..., \{t \mid (t \in \mathcal{T} \land v \neq p) \lor ((p, \underline{e}, \overline{E}, v') \in \mathcal{T} \land v = \mathsf{state}(p))\}, \\ \{l \mid (l \in \mathfrak{L}^{\otimes} \land v \neq p) \lor ((p, v') \in \mathfrak{L}^{\otimes} \land v = \mathsf{state}(p))\}, ...)\end{array}}$$

(rx) **Remove open exiting transition.** This pattern removes an exit point p that is open with respect to at least one region, along with all transitions and links originating from p.

$$\frac{p \in \mathcal{P}^{\otimes} \qquad \exists r \in \mathsf{chdn}_{\mathcal{R}}(\mathsf{state}(p)) : \mathsf{opn}_{\otimes}(p, r)}{\mathcal{D} \rightsquigarrow (..., \mathcal{P}^{\otimes} \setminus \{p\}, ..., \{t \in \mathcal{T} \mid v \neq p\}, ..., \{l \in \mathfrak{L}^{\otimes} \mid v \neq p\}, ...)}$$

Note that the corresponding rule for open entering transitions, removing an entry point together with transitions and links leading to the entry point, would be unsound in general. This is because the incoming transition could be required by its source state.

(n$\bigcirc$) **Reduce nondeterminism in entry links.** This pattern removes a link originating from an entry point p that has at least one further link to the same region.

$$\dfrac{(p, v_1), (p, v_2) \in \mathfrak{L}^{\bigcirc} \quad v_1 \neq v_2 \quad \mathsf{reg}(v_1) = \mathsf{reg}(v_2)}{\mathcal{D} \rightsquigarrow (..., \mathfrak{L}^{\bigcirc} \setminus \{(p, v_1)\}, ...)}$$

(at) **Add transition.** This pattern adds a new transition $(v, \underline{e}, \overline{E}, v')$ originating from a vertex v that is open with respect to $\underline{e}$.

$$\dfrac{\begin{array}{c} v \in \mathcal{S} \cup \mathcal{P}^{\otimes} \quad v' \in \mathcal{S} \cup \mathcal{P}^{\bigcirc} \quad \mathsf{reg}(v) = \mathsf{reg}(v') \quad \underline{e} \in \mathsf{opn}(\mathsf{state}(v)) \\ \overline{E} \subseteq \overline{\mathsf{con}}(\mathsf{reg}(v)) \cup \{\bullet\} \end{array}}{\mathcal{D} \rightsquigarrow (..., \mathcal{T} \cup \{(v, \underline{e}, \overline{E}, v')\}, ...)}$$

(rt) **Remove transition.** This patterns removes a transition $(v, \underline{e}, \overline{E}, v')$ originating from a vertex v that is open with respect to $\underline{e}$.

$$\dfrac{(v, \underline{e}, \overline{E}_1, v') \in \mathcal{T} \quad \underline{e} \in \mathsf{opn}(\mathsf{state}(v))}{\mathcal{D} \rightsquigarrow (..., \mathcal{T} \setminus \{(v, \underline{e}, \overline{E}_1, v')\}), ...}$$

(nt) **Reduce nondeterminism in transitions.** This pattern removes a transition $(v, \underline{e}, \overline{E}, v')$ if there is another transition with input event $\underline{e}$ originating from v.

$$\dfrac{(v, \underline{e}, \overline{E}_1, v_1'), (v, \underline{e}, \overline{E}_2, v_2') \in \mathcal{T} \quad \overline{E}_1 \neq \overline{E}_2 \vee v_1' \neq v_2'}{\mathcal{D} \rightsquigarrow (..., \mathcal{T} \setminus \{(v, \underline{e}, \overline{E}_1, v_1')\}), ...}$$

(ct) **Close for transitions.** This pattern removes openness with respect to events from a state s by restricting the set of open events to a subset E. If s is not a top-level state, i.e. is not in region ϵ, E must remain a super-set of the surrounding state's open events.

$$\dfrac{s \in \mathcal{S} \quad E \subseteq \mathsf{opn}(s) \quad \mathsf{par}(s) \neq \epsilon \Rightarrow E \supseteq \mathsf{opn}(\mathsf{par}(\mathsf{par}(s)))}{\mathcal{D} \rightsquigarrow (..., \mathsf{opn}[s \mapsto E])}$$

(ao) **Add output events.** This pattern adds a set of output events E_1 to a transition t having an open output event set ($\bullet \in \overline{E}$).

$$\dfrac{t \in \mathcal{T} \quad \bullet \in \overline{E} \quad E_1 \subseteq \overline{\mathsf{con}}(\mathsf{reg}(v))}{\mathcal{D} \rightsquigarrow (..., (\mathcal{T} \setminus \{t\}) \cup \{(v, \underline{e}, \overline{E} \cup E_1, v')\}, ...)}$$

(co) **Close for output events.** This pattern closes a transition's output event set.

$$\dfrac{t \in \mathcal{T} \quad \bullet \in \overline{E}}{\mathcal{D} \rightsquigarrow (..., (\mathcal{T} \setminus \{t\}) \cup \{(v, \underline{e}, \overline{E} \setminus \{\bullet\}, v')\}, ...)}$$

(sc) **Strengthen event constraint.** These two patterns strengthen the input or output event constraint of a region r by restricting the event set to a subset E, which however needs to remain a super-set of the constraints of all subregions. For output events, set

E must not be stricter than existing transitions in r. For input events, set E also must not be stricter than existing open events in substates of r.

$$\frac{\begin{array}{c} r \in \mathcal{R} \quad E \subseteq \underline{\mathsf{con}}(r) \quad \forall \overline{r} \in \mathcal{R} : \mathsf{par}(\mathsf{par}(\overline{r})) = r \Rightarrow E \supseteq \underline{\mathsf{con}}(\overline{r}) \\ \forall s \in \mathsf{chdn}(r) : \mathsf{opn}(s) \subseteq E \wedge \forall t \in \mathcal{T} : v = s \Rightarrow \underline{e} \in E \end{array}}{\mathcal{D} \rightsquigarrow (..., \underline{\mathsf{con}}[r \mapsto E], ...)}$$

$$\frac{\begin{array}{c} r \in \mathcal{R} \quad E \subseteq \overline{\mathsf{con}}(r) \quad \forall \overline{r} \in \mathcal{R} : \mathsf{par}(\mathsf{par}(\overline{r})) = r \Rightarrow E \supseteq \overline{\mathsf{con}}(\overline{r}) \\ \forall t \in \mathcal{T} : v \in \mathsf{chdn}(r) \Rightarrow \overline{E} \subseteq E \cup \{\bullet\} \end{array}}{\mathcal{D} \rightsquigarrow (..., \overline{\mathsf{con}}[r \mapsto E], ...)}$$

(ds) **Duplicate state.** This pattern duplicates a state s by copying its complete substructure X including entry and exit points of s, subregions and substates. Transitions, links, event constraints and open events in the copied substructure are set as in the original, with transitions and links entering and leaving X also entering and leaving its copy. In addition, transitions between the copies are introduced if they are well-formed. If initial states are copied, their copy is not an initial state, in order to avoid a conflicting start configuration.

$$\frac{s \in \mathcal{S} \quad X = \mathsf{sub}(s) \quad \iota \text{ maps } x \in X \text{ to some fresh } \iota(x) \text{ and } x \notin X \text{ to } x}{\begin{array}{c} \mathcal{D} \rightsquigarrow (\mathcal{S} \cup \iota(X \cap \mathcal{S}), \mathcal{S}^0, \mathcal{P}^{\bigcirc} \cup \iota(X \cap \mathcal{P}^{\bigcirc}), \mathcal{P}^{\otimes} \cup \iota(X \cap \mathcal{P}^{\otimes}), \mathcal{R} \cup \iota(X \cap \mathcal{R}), \\ \mathsf{par} \cup \{(\iota(s), \mathsf{par}(s))\} \cup \{(\iota(x), \iota(\mathsf{par}(x))) \mid x \in X \setminus \{s\}\}, \\ \{t \mid \exists t_1 \in \mathcal{T} : v \in \{v_1, \iota(v_1)\} \wedge \underline{e} = \underline{e}_1 \wedge \overline{E} = \overline{E}_1 \wedge v' \in \{v'_1, \iota(v'_1)\} \wedge \mathsf{reg}(v) = \mathsf{reg}(v')\}, \\ \{l \mid \exists l_1 \in \mathfrak{L}^{\bigcirc} : v \in \{v_1, \iota(v_1)\} \wedge v' \in \{v'_1, \iota(v'_1)\} \wedge \mathsf{state}(v) = \mathsf{par}(\mathsf{reg}(v'))\}, \\ \{l \mid \exists l_1 \in \mathfrak{L}^{\otimes} : v \in \{v_1, \iota(v_1)\} \wedge v' \in \{v'_1, \iota(v'_1)\} \wedge \mathsf{par}(\mathsf{reg}(v)) = \mathsf{state}(v')\}, \\ \underline{\mathsf{con}} \cup \{(\iota(r), \underline{\mathsf{con}}(r)) \mid r \in X \cap \mathcal{R}\}, \overline{\mathsf{con}} \cup \{(\iota(r), \overline{\mathsf{con}}(r)) \mid r \in X \cap \mathcal{R}\}, \\ \mathsf{opn} \cup \{(\iota(v), \mathsf{opn}(v)) \mid v \in X \cap \mathcal{S}\}) \end{array}}$$

(gs) **Group states.** This pattern groups a set of states S, all sharing the same surrounding region r, in a new state $\overline{s}$, which has new default region $\overline{r}$. Transitions and links entering and leaving S (collected in sets $\Gamma_{\bigcirc}$, $\Gamma_{\otimes}$, $\Lambda_{\bigcirc}$, $\Lambda_{\otimes}$) have to cross the new state's border via new entry and exit points, denoted by p_t for crossing transitions and p_l for crossing links. If there is an initial state in S, the new state also becomes an initial state. Event constraints for the new state's region $\overline{r}$ are inherited by the surrounding region r, and the new state is open with respect to the intersection of all open events of states in S.

$$\frac{\begin{array}{c} r \in \mathcal{R} \quad S \subseteq \mathsf{chdn}(r) \quad \widetilde{S} = \bigcup_{s \in S} \mathsf{sub}(s) \quad \overline{s}, \overline{r} \in \mathsf{fresh}(\mathcal{D}) \\ \Gamma_{\bigcirc} = \{t \in \mathcal{T} \mid v \notin \widetilde{S} \wedge v' \in \widetilde{S}\} \quad \Gamma_{\otimes} = \{t \in \mathcal{T} \mid v \in \widetilde{S} \wedge v' \notin \widetilde{S}\} \\ \Lambda_{\bigcirc} = \{l \in \mathfrak{L}^{\bigcirc} \mid v \notin \widetilde{S} \wedge v' \in \widetilde{S}\} \quad \Lambda_{\otimes} = \{l \in \mathfrak{L}^{\otimes} \mid v \in \widetilde{S} \wedge v' \notin \widetilde{S}\} \\ \forall x \in \Gamma_{\bigcirc} \cup \Gamma_{\otimes} \cup \Lambda_{\bigcirc} \cup \Lambda_{\otimes} : p_x \in \mathsf{fresh}(\mathcal{D}) \end{array}}{\begin{array}{c} \mathcal{D} \rightsquigarrow (\mathcal{S} \cup \{\overline{s}\}, \mathcal{S}^0 \cup \{s \mid s = \overline{s} \wedge \mathcal{S}^0 \cap S \neq \emptyset\}, \mathcal{P}^{\bigcirc} \cup \{p_t \mid t \in \Gamma_{\bigcirc}\} \cup \{p_l \mid l \in \Lambda_{\bigcirc}\}, \\ \mathcal{P}^{\otimes} \cup \{p_t \mid t \in \Gamma_{\otimes}\} \cup \{p_l \mid l \in \Lambda_{\otimes}\}, \mathcal{R} \cup \{\overline{r}\}, \mathsf{par}[\overline{s} \mapsto r, \overline{r} \mapsto \overline{s}] \cup \{(s, \overline{r}) \mid s \in S\}, \\ \{t \mid t \in \mathcal{T} \setminus (\Gamma_{\bigcirc} \cup \Gamma_{\otimes}) \vee (\exists t_1 \in \Gamma_{\bigcirc} : t = (v_1, \underline{e}_1, \overline{E}_1, p_{t_1})) \vee \\ (\exists t_2 \in \Gamma_{\otimes} : t = (p_{t_2}, \underline{e}_2, \overline{E}_2, v'_2))\}, \\ \{l \mid l \in \mathfrak{L}^{\bigcirc} \setminus \Lambda_{\bigcirc} \vee (\exists t_1 \in \Gamma_{\bigcirc} : l = (p_{t_1}, v'_1)) \vee (\exists l_1 \in \Lambda_{\bigcirc} : l = (v_1, p_{l_1}) \vee l = (p_{l_1}, v'_1))\}, \\ \{l \mid l \in \mathfrak{L}^{\otimes} \setminus \Lambda_{\otimes} \vee (\exists t_1 \in \Gamma_{\otimes} : l = (v_1, p_{t_1})) \vee (\exists l_1 \in \Lambda_{\otimes} : l = (v_1, p_{l_1}) \vee l = (p_{l_1}, v'_1))\}, \\ \underline{\mathsf{con}}[\overline{r} \mapsto \underline{\mathsf{con}}(r)], \overline{\mathsf{con}}[\overline{r} \mapsto \overline{\mathsf{con}}(r)], \mathsf{opn}[\overline{s} \mapsto \bigcap_{s \in S} \mathsf{opn}(s)]) \end{array}}$$

(u$\bigcirc$) **Unify entry points.** If there are two entry points p_1, p_2 such that all transitions and links leading to these points are "equal" up to p_1 and p_2, then this pattern removes p_2 and adds entry links leaving p_1 that originally left p_2.

$$\frac{s \in \mathcal{S} \quad p_1, p_2 \in \mathcal{P}^{\bigcirc} \cap \mathsf{chdn}(s) \quad p_1 \neq p_2 \quad \mathcal{T}.p_1 = \mathcal{T}.p_2 \quad \mathfrak{L}^{\bigcirc}.p_1 = \mathfrak{L}^{\bigcirc}.p_2}{\begin{array}{c}\mathcal{D} \rightsquigarrow (\ldots, \mathcal{P}^{\bigcirc} \setminus \{p_2\}, \ldots, \mathsf{par} \setminus \{(p_2, s)\}, \{t \in \mathcal{T} \mid v' \neq p_2\}, \ldots, \\ \{l \mid (l \in \mathfrak{L}^{\bigcirc} \wedge v \neq p_2 \neq v') \vee ((p_2, v') \in \mathfrak{L}^{\bigcirc} \wedge v = p_1)\}, \ldots)\end{array}}$$

(u$\otimes$) **Unify exit points.** If there are two exit points p_1, p_2 such that all transitions and links originating from these points are "equal" up to p_1 and p_2, then this pattern removes p_2 and adds exits links leading to p_1 that originally lead to p_2.

$$\frac{s \in \mathcal{S} \quad p_1, p_2 \in \mathcal{P}^{\otimes} \cap \mathsf{chdn}(s) \quad p_1 \neq p_2 \quad p_1.\mathcal{T} = p_2.\mathcal{T} \quad p_1.\mathfrak{L}^{\otimes} = p_2.\mathfrak{L}^{\otimes}}{\begin{array}{c}\mathcal{D} \rightsquigarrow (\ldots, \mathcal{P}^{\otimes} \setminus \{p_2\}, \ldots, \mathsf{par} \setminus \{(p_2, s)\}, \{t \in \mathcal{T} \mid v \neq p_2\}, \ldots, \\ \{l \mid (l \in \mathfrak{L}^{\otimes} \wedge v \neq p_2 \neq v') \vee ((v, p_2) \in \mathfrak{L}^{\otimes} \wedge v' = p_1)\}, \ldots)\end{array}}$$

Theorem 6.11. *The refinement patterns are sound, i.e. for every state diagram $\mathcal{D}_1$, and $\mathcal{D}_2$ with $\mathcal{D}_1 \rightsquigarrow \mathcal{D}_2$, $\mathcal{D}_2$ is a state diagram and $\mathcal{D}_2 \leq \mathcal{D}_1$.*

Proof. It is easily checked that $\mathcal{D}_2$ is in fact a (well-formed) state diagram.

Patterns (at), (rt), (ao), (a$\bigcirc$), (a$\otimes$), (u$\bigcirc$), and (u$\otimes$) do not change the semantics at all, as discussed in the following: The added transition in (at) would have been introduced by transformation reduce anyway. The removed transition in (rt) will still be introduced by transformation reduce. (ao) has no effect on the semantics, because $\bullet$ stays in the set of output events of the transition. Pattern (a$\bigcirc$) does not change the semantics, because the new entry point has the same behavior as helper component $\bigcirc_s$, which is introduced by transformation reduce. Applications of (u$\bigcirc$) or (u$\otimes$) also do not change the semantics, since for the definitions of relations $\longrightarrow_{\otimes}$ and $\longrightarrow_{\bigcirc}$, only the reachable transitions are relevant, not the link paths, by which they are reached.

The following patterns change the semantics but yield an equivalent semantic model: The added state in pattern (as) is not reachable by transition or by explicit link, also it is not linked to an exit point explicitly. It has the open events defined by the region's event constraint. Thus, it has the same behavior as the helper state $\square_{\overline{r}}$ that is introduced by transformation reduce. The added region in pattern (ar) implies a new helper state, which however has only a part of the behavior of the helper state of $\overline{r}$. For pattern (a$\otimes$), the new exit point has the same implicit transitions as its parent. Whenever an implicit transition from the exit point is enabled, the corresponding implicit transition of the parent is enabled, too. Thus, no new behavior is added. Pattern (r$\bigcirc$) leads to not entering state $\mathsf{state}(p)$. This behavior however already was described before by transformation reduce via the helper states introduced for each region. Refining those to only perform self-loops is equivalent to not entering the state. Pattern (gs) introduces a new state $\overline{s}$ which, in case it has no open events, is active if and only if one of the grouped states in S is active. If it has open events, these are defined by the intersection of open events in S, and transformation reduce introduces corresponding additional helper components with open behavior, which however do not introduce *new* open behavior and only reflects behaviour that has already been exhibited by the substates.

The following patterns lead to a proper refinement, i.e. not to an equivalent model: For pattern (s$\bigcirc$), note that an entry point that is open with respect to region r is equivalent to one linking to all possible targets in r. Specifying a subset thus results in a subset of outgoing transitions in the semantics. For pattern (s$\otimes$), note that implicit exit links are dealt with like may transitions. In the semantic translation they correspond to several branches with both the behavior that would be implied by the exit link and the behavior that would be implied if the exit link would not exist (this can be a self-loop or a lower-priorized transition enabled for the event). Both explicitly selecting ($v \in V$) and explicitly discarding ($v \notin V$) thus just corresponds to the removal of semantic transition branches. Patterns (r$\otimes$) and (rx) are special cases of (s$\otimes$): (r$\otimes$) corresponds to the implementation of all implicit may exit links, with respect to all regions, whereas (rx) corresponds to the removal of all implicit may exit links, with respect to all regions. Patterns (n$\bigcirc$) and (nt) semantically translate to removing a transition branch. Also patterns (ct) and (co) only remove transition branches. This is also the case for pattern (sc), although the transition branches removed there are implicit ones, introduced by transformation reduce. Finally, refinement for pattern (ds) can be shown by defining the refinement relation to relate configurations containing copied states with those of the corresponding originals. $\square$

6.5 Example: top-down development of an elevator control

We model the control unit of a simple elevator that only serves two floors. The input events of the model are $\underline{\mathsf{Ev}} = \{on, off, go, stopped, opened, closed, canceled, call, alarm, pickup, hangup\}$, and the output events are $\overline{\mathsf{Ev}} = \{\overline{up}, \overline{down}, \overline{stop}, \overline{open}, \overline{close}, \overline{d0}, \overline{d1}, \overline{dup}, \overline{ddown}, \overline{dalarm}, \overline{dclear}, \overline{ring}, \overline{connect}, \overline{disconnect}\}$. Input events *on* and *off* can be issued by an operator to turn the elevator on and off, respectively. Input event *go* is given by a user who therewith requests the cabin to move to the other floor. The modeled system controls the motor for vertical movement of the cabin using output events $\overline{up}$, $\overline{down}$ and $\overline{stop}$, respectively requesting the cabin to move up, down or stop immediately. The motor unit responds with an input event *stopped* as soon as it has come to a halt. The door mechanism is controlled using output events $\overline{open}$ and $\overline{close}$. The door mechanism responds with event *opened* or *closed*, if opening or closing is complete, respectively. It sends event *canceled* if the closing of the door could not be completed (e.g. if there is an obstacle in the door area). The display is controlled using output events $\overline{d0}$, $\overline{d1}$, $\overline{dup}$, $\overline{ddown}$, $\overline{dalarm}$, and $\overline{dclear}$, resulting in the display showing icons for “floor 0”, “floor 1”, “moving up”, “moving down”, “alarm”, or clearing the display, respectively. An alarm button inside the cabin generates event *alarm*. It shall perform an emergency stop: upon pressing the button, the cabin should halt immediately, the doors should be opened, and no further mechanical actions should be performed. Finally, there is a communication system: a button inside the cabin generates an input event *call*, requesting a call to an operator. Output event $\overline{ring}$ is then issued to inform the operator of a communication request. Event *pickup* occurs when the operator answers (then, $\overline{connect}$ is issued, which establishes a connection), and event *hangup* occurs when the operator ends the connection (then $\overline{disconnect}$ is issued which terminates the connection).

Figure 6.4 depicts the model of the elevator control, specified completely except for the handling of event *canceled* in substate CLOSING of the DOOR region. The rest of this section illustrates how this model can be developed top-down using the refinement patterns presented in Section 6.4.

Top-down development. The left side of Figure 6.5 shows the most general model possible,

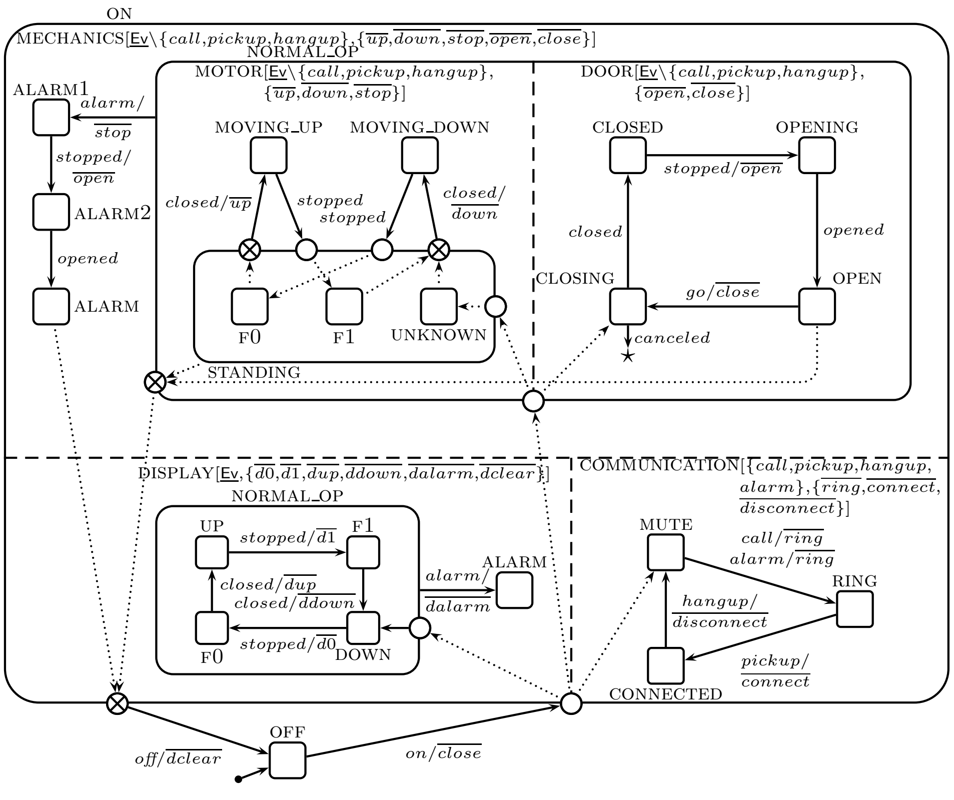


Figure 6.4: A model of an elevator control unit. Here, $\underline{\mathsf{Ev}}$ = {*on*, *off*, *go*, *stopped*, *opened*, *closed*, *canceled*, *call*, *alarm*, *pickup*, *hangup*}, and $\overline{\mathsf{Ev}}$ = {$\overline{up}$, $\overline{down}$, $\overline{stop}$, $\overline{open}$, $\overline{close}$, $\overline{d0}$, $\overline{d1}$, $\overline{dup}$, $\overline{ddown}$, $\overline{dalarm}$, $\overline{dclear}$, $\overline{ring}$, $\overline{connect}$, $\overline{disconnect}$}.

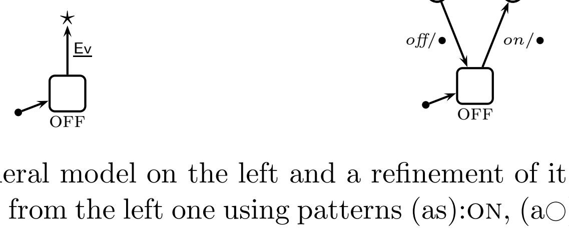


Figure 6.5: The most general model on the left and a refinement of it on the right. The right model evolves from the left one using patterns (as):ON, (a○), (a⊗), 2×(at), (ct):ON, (ct):OFF.

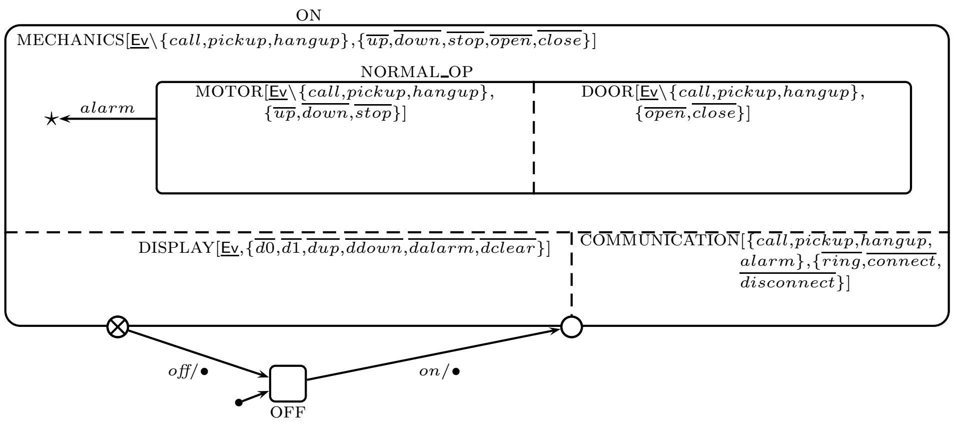

ON
MECHANICS[$\underline{\mathsf{Ev}}\setminus\{call,pickup,hangup\},\{\overline{up},\overline{down},\overline{stop},\overline{open},\overline{close}\}$]
NORMAL_OP
MOTOR[$\underline{\mathsf{Ev}}\setminus\{call,pickup,hangup\},\{\overline{up},\overline{down},\overline{stop}\}$]
DOOR[$\underline{\mathsf{Ev}}\setminus\{call,pickup,hangup\},\{\overline{open},\overline{close}\}$]
alarm
DISPLAY[$\underline{\mathsf{Ev}},\{\overline{d0},\overline{d1},\overline{dup},\overline{ddown},\overline{dalarm},\overline{dclear}\}$]
COMMUNICATION[$\{call,pickup,hangup,alarm\},\{\overline{ring},\overline{connect},\overline{disconnect}\}$]
off/●
on/●
OFF

Figure 6.6: A refinement of the previous model (Figure 6.5, right) which evolves using patterns (ar):DISPLAY, (ar):COMMUNICATION, 2×(sc):MECHANICS, (sc):DISPLAY, 2×(sc):COMMUNICATION, [(as),(ct)]:NORMAL_OP, (ar):DOOR, (sc):MOTOR, (sc): DOOR.

from which any model can be refined. As a first step, we refine it to the model shown on the right of Figure 6.5. The basic idea is that the elevator control will have an OFF state which is initial, and an ON state which will be refined further. The OFF state will be left on input event *on*, leading to a further-to-be-refined configuration in state ON (expressed by an open entry point), and the ON state will be left on event *off* leading to state OFF, provided it is in a further-to-be-refined configuration in state ON (expressed by an open exit point). We do not yet specify which output events should be generated when taking the transitions and therefore use the open event set notation •.

The next refinement is shown in Figure 6.6, where we refine the ON state. It will have three regions, where MECHANICS will control the mechanical parts of the elevator, i.e. the motor for cabin movements and the door mechanism. By event constraint, it may react on all input events, except for events *call*, *pickup*, and *hangup*. It may generate only the output events $\overline{up}$, $\overline{down}$, $\overline{stop}$, $\overline{open}$, and $\overline{close}$. A further region is for the DISPLAY inside the cabin which may only control the display device and generate no further events. However, it may react to any event since it may be reasonable to illustrate on the display any event that might happen. The third region is for the COMMUNICATION system between the cabin and the operator. It may only react on input events and generate output events related to communication.

In Figure 6.6, the MECHANICS state is refined further. It has a state for normal operation which may be left in case of *alarm*. NORMAL_OP has two regions where one of them, MOTOR, controls the vertical movement of the cabin by issuing events $\overline{up}$, $\overline{down}$, and $\overline{stop}$. The other region, DOOR, controls the opening and closing of the DOOR by issuing events $\overline{open}$ and $\overline{close}$.

As next steps, the regions can be refined independently from one another. For example, the communication system could be modeled, as shown in Figure 6.7.

Now, it is possible to check properties dealing only with the communication system. For example, we know that after an initial *on* and as long as *off* does not occur, *call* and *alarm* are the only events that lead to $\overline{ring}$. This is because no other region in ON may generate an event $\overline{ring}$ due to the given event constraints.

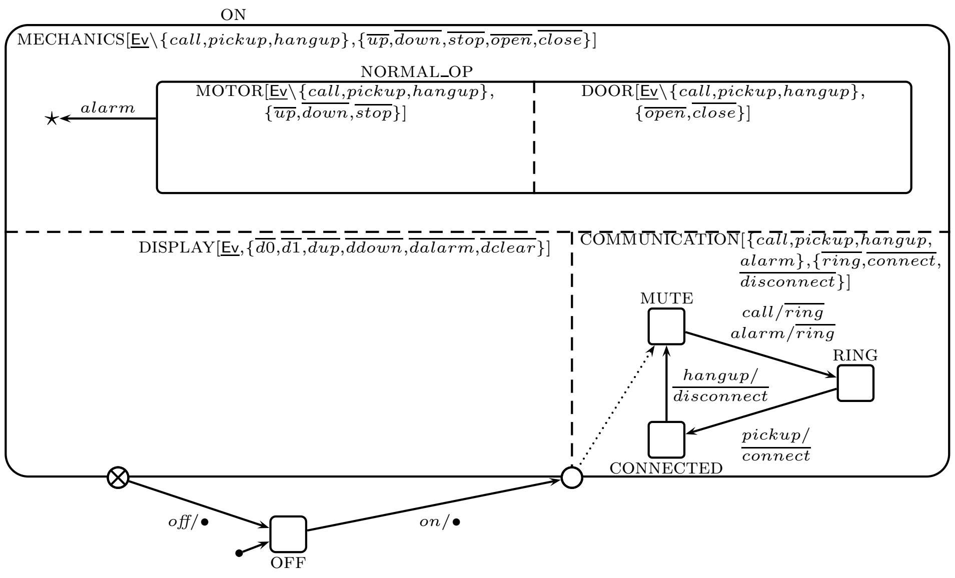

ON
MECHANICS[Ev\{call,pickup,hangup},{up,down,stop,open,close}]
NORMAL_OP
MOTOR[Ev\{call,pickup,hangup},{up,down,stop}]
DOOR[Ev\{call,pickup,hangup},{open,close}]
alarm
DISPLAY[Ev,{d0,d1,dup,ddown,dalarm,dclear}]
COMMUNICATION[{call,pickup,hangup,alarm},{ring,connect,disconnect}]
MUTE
call/ring
alarm/ring
RING
hangup/disconnect
pickup/connect
CONNECTED
off/•
on/•
OFF

Figure 6.7: A refinement of the previous model (Figure 6.6) which evolves using patterns 3×(as), (s○), 3×(at), 3×(ct).

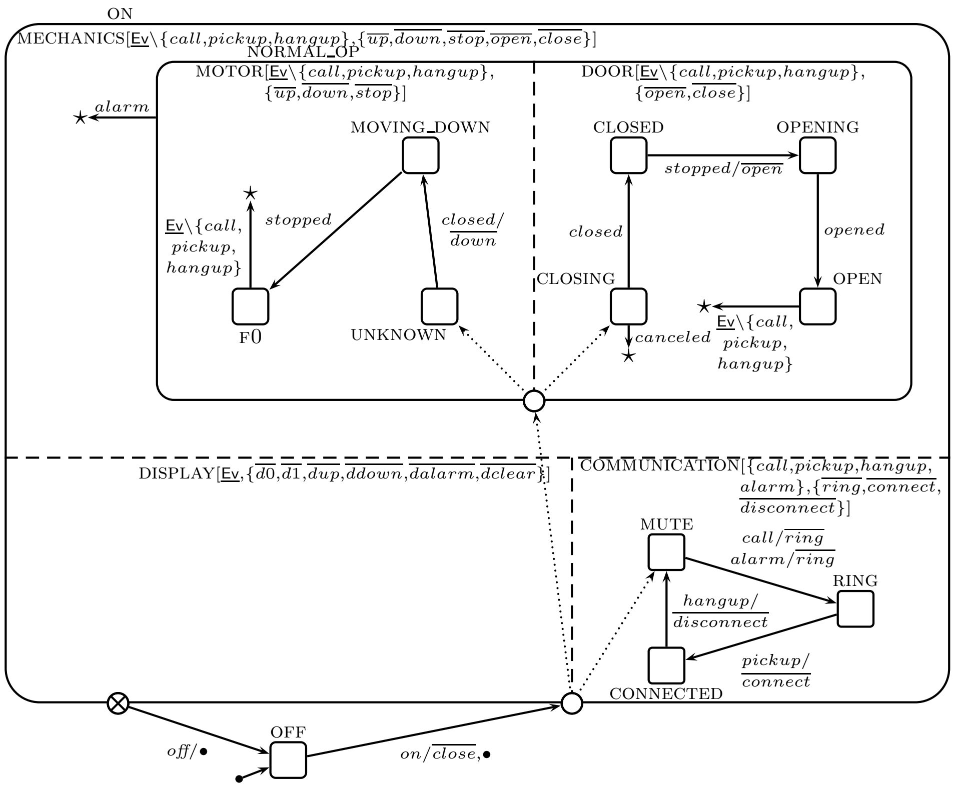

ON
MECHANICS[Ev\{call,pickup,hangup},{up,down,stop,open,close}]
NORMAL_OP
MOTOR[Ev\{call,pickup,hangup},{up,down,stop}]
DOOR[Ev\{call,pickup,hangup},{open,close}]
alarm
MOVING_DOWN
Ev\{call,pickup,hangup}
stopped
closed/down
F0
UNKNOWN
CLOSED
OPENING
stopped/open
closed
opened
CLOSING
OPEN
canceled
Ev\{call,pickup,hangup}
DISPLAY[Ev,{d0,d1,dup,ddown,dalarm,dclear}]
COMMUNICATION[{call,pickup,hangup,alarm},{ring,connect,disconnect}]
MUTE
call/ring
alarm/ring
RING
hangup/disconnect
pickup/connect
CONNECTED
OFF
off/•
on/close,•

Figure 6.8: A refinement of the previous model (Figure 6.7), implementing initial behavior. It evolves using patterns (ao), (a○), 7×(as), 3×(s○), 5×(at), 5×(ct).

Next, let us refine the heart of the control unit, region MECHANICS. Initially, we do not know the positions of the cabin and the door, so the first action after *on* will be to establish an orderly state for both the cabin and the door mechanism by closing the door, moving the cabin to the ground floor and opening the door there. The corresponding model is shown in Figure 6.8.

The described initial procedure can now be proven to occur as expected. We know that only *alarm* and *off* can interrupt the expected procedure (these transitions have priority over the transitions inside NORMAL_OP), although this is also not yet guaranteed.

Finally, we argue, exemplary for the MOTOR region, that the current model can evolve to the final model illustrated in Figure 6.4: first, we add states MOVING_UP and F1. Then, we add transitions from F0 to MOVING_UP, from MOVING_UP to F1, and from F1 to MOVING_DOWN. The new states are closed using pattern (ct). Now, we group states F0, F1, and UNKNOWN in a new state STANDING using pattern (gs). Pattern (u$\otimes$) allows us to save one exit point (F1 and UNKNOWN are now left via a common exit point). An exit point for NORMAL_OP is introduced and, due to the new grouping state STANDING, it only needs one link for the MOTOR region.

6.6 Possible extensions

Our modeling formalism features a subset of commonly available notations in several established dialects of state diagrams. We decided for this restriction because (i) a safe subset of semantically clear concepts should be preferred to a variety of different notations with different (maybe even ambiguous) semantic meanings in different dialects; (ii) the presentation is kept as simple as possible, not obscuring the general concepts of top-down development in state diagrams; and (iii) our subset is meaningful for practical applications, as is demonstrated by our example. Nevertheless, there may of course be practical applications that require features not presented here. This section discusses such possible further extensions and shows how they can equivalently be described in our formalism or, if they cannot, how the semantics would have to be adapted in order to handle them.

Variables and guards. Our modeling formalism is lacking variable assignments and guards depending on these variables. Harel statecharts also allow more sophisticated guards, e.g. guard (in A) evaluating to true if state A is active, which however mostly can be reduced to variables, e.g. by setting a flag variable when A is entered and removing it when it is left. In order to make our model amenable to variables and guards, the configurations in the semantic model would have to hold an additional component for a global variable assignment, i.e. a function mapping variable symbols to their current values. Then, executing a transition may include changes in the variable assignment and enabledness of transitions may also depend on the evaluation of guards with respect to the current variable assignment.

Guards lead to slightly modified refinement patterns. For instance, it would be an additional pattern to strengthen a guard of a transition as long as its source state is still open with respect to the transition's input event, or as long as there is a second transition from the same source, with the same input event, and with an "overlapping" guard.

Global variables are difficult to handle in a refinement-sensitive setting. The problem is that variables changed in one part of a system may affect guards in another part, which makes it hard to follow a compositional top-down design approach. One would need extra notations that guarantee properties of variables, e.g. *contracts* that one part of the system fulfills and the other part can rely on.

Various pseudostates. Our formalism does not have notations for default or history-based state entry, or fork/join pseudostates. However, our formalism's entry and exit points, together with their attached links, can replace initial, fork, and join transitions. History pseudostates cannot be expressed in our formalism because they would, similar to variables, need extra information stored in semantic configurations, remembering substate configurations for later re-entry. Further pseudostates existing in UML state machines are junction and choice. Transitions composed using junction pseudostates can be expressed using several normal transitions. Choice pseudostates are primarily characterized by their handling of actions and guards which are not featured in our formalism, but they could be represented by several normal transitions and intermediate states, if variables and guards would be supported.

Entry, exit, and do actions. *Do activities*, as in UML state machines, can be modeled in our formalism using an orthogonal region performing the do actions. Entry and exit actions would have to be made explicit in transitions leading to and originating from entry/exit points, respectively.

Deferred events. In UML state machines, events can be deferred, i.e. in case they do not fire any transition, they can be stored for future handling instead of being discarded. To support deferred events, our semantic configurations would have to be extended by an additional component holding the (multi-)set of currently deferred events, and there would be an alternative version of SOS rule (DISCARD) which also performs a self-loop but stores the action in the component of deferred events. If there are deferred events, silent transitions (labeled, e.g., by a special, unobservable action τ) can fire, consuming the event from the pool of deferred events instead of requesting it from the environment.

Further extensions. There are further possible extensions that may be useful, especially in the context of refinement in the top-down development process. For example, if conflicting initial states were allowed, this would express initial nondeterminism (conflicts would have to be resolved as a first step). This could be handled by an initial state set in the semantic model, as is commonly done. Furthermore, one might want to specify open input events also with respect to output events or target states, which would make it possible to express that a transition with a specified input event may be added, but then only with (a subset or superset of) a defined set of output events or only to (a subset or superset of) a defined set of allowed target vertices. The concept of open entry and exit points could be refined by may-links, which would allow certain links in an implementation and disallow others (open points always allow *all* possible links with respect to a region). A further possible design choice would be to allow introducing event constraints that prohibit events that already occur in the region. Then, an event constraint would only restrict the possibility to add transitions in the future.

6.7 Related work

There are numerous papers on the semantics of statecharts, in their more classical variant, e.g. [HN96], and in their newer variant as defined in the UML, e.g. [HK04, FS07c]. An overview is given in [CD05a]. Fewer papers, as listed in the following, can be found that deal with refinement issues. The notions of refinement vary among these papers. [Sch98] and [LB98] use a kind of trace inclusion, [Rum96, RK96] employs an inclusion of so-called stream processing functions. [MNB04] is the only paper that takes a coalgebraic approach as we do, defining a refinement notion based on simulation. All these papers give refinement patterns (also called rules or transformations). However, none of them improves the syntax of statecharts to allow for more flexible underspecification, together with the implied patterns, as we do. Furthermore, many of them lack important rules given in this paper, e.g. the state duplication pattern is missing in [Sch98, LB98, Rum96, RK96].

6.8 Conclusions

We defined a variant of statecharts that is especially designed for underspecification in a top-down development process, maximizing compositionality and minimizing notational overhead. We gave a formal semantics using the simulation-based refinement notion of $\mathbb{IOTS}$, as defined in Section 2.3.3. A set of basic refinement patterns was presented that allows for a structured and stepwise top-down development, as illustrated by an example. Our results pave the way for the implementation of a design tool for formally precise and correct-by-construction top-down development of state diagrams.

Chapter 7

Conclusions

This thesis established new results related to nondeterminism in semantic models. We formally characterized the uniform notion of a *refinement setting* which, roughly speaking, consists of a set of models, a refinement preorder and an embedding of implementations. We defined a total of 19 different refinement settings, largely taken from the literature. 12 of these refinement settings support only one kind of nondeterminism, namely *resolvable nondeterminism*, which introduces underspecification and therefore is essential for the top-down development of software systems. The proper semantic handling of this nondeterminism is required in order to ensure the correctness of underspecified models and the soundness of refinement steps, using formal methods.

Because of the large amount of available settings that support resolvable nondeterminism, it is hard for a designer to decide which setting is most suitable for the given application at hand. Our results presented in Chapter 3 support answering this question via expressiveness comparisons between the most popular settings. Not only is the set of results new, but so is the approach of comparison. Our approach does not compare the coarseness of refinement preorders, as is usually done, e.g. in [vG01, EF02], but it compares the sets of expressible sets of implementations. We established an expressiveness hierarchy using language-preserving transformations. These transformations can also be used to convert between settings and to reuse algorithms and tools across different settings.

Further, 6 of the 19 presented refinement settings support not only resolvable nondeterminism, but also *persistent nondeterminism* which remains existent in implementations and can, e.g., be used to express faulty or random behavior. Consequently, the implementations in these settings need not necessarily be deterministic. This thesis presented two practical application fields for semantic models with both persistent and resolvable nondeterminism.

First, Chapter 4 showed that both kinds of nondeterminism must be handled when giving a semantics to a process algebra for concurrent systems. This is because (i) the common interpretation of the choice operator concerns persistent nondeterminism, and (ii) concurrent execution may introduce resolvable nondeterminism, to be resolved by a scheduler of the operating system. We gave a structural operational semantics in terms of μ-automata, and also presented a sound and complete axiom system characterizing our refinement preorder.

The second application field for refinement settings supporting the two kinds of nondeterminism is state diagrams (i.e. statecharts [Har87], UML state machines [Obj07] and further variants). In Chapter 5, we discussed that using persistent nondeterminism is a suitable approach to model failures or random decisions in state diagrams. Resolvable nondeterminism is present by the underspecification inherent to state diagrams as abstract specifications. In order to handle both kinds of nondeterminism, we developed the refinement setting of ν-automata, which is dual to μ-automata [JW95] but allows for a more succinct description of state diagram semantics. The new setting is equipped with a satisfaction relation over the μ-calculus [Koz83]. Then we used ν-automata to give a formal semantics to a simple state diagram variant, showed how existing semantics can be adapted to also allow for persistent nondeterminism, and discussed how our approach can be extended with distributions for persistent nondeterminism, interpreted as random choice.

Chapter 6 continued our consideration of state diagram semantics but focused on resolvable nondeterminism and the refinement it expresses. A new state diagram variant was developed, with a syntax supporting the requirements in the top-down development process. It was given a formal semantics in a new refinement setting, introduced especially for this application. The setting is based on ready simulation, but distinguishes between input and output events, representing triggers and the sending of events, respectively. An expressive set of refinement patterns was given that allows for a structured stepwise top-down development. The patterns' use was illustrated by an example.

Future work. Possible future research with respect to the theoretical line of this thesis includes performing the expressiveness comparison in the style of Chapter 3, but for the settings that can express not only resolvable but also persistent nondeterminism. Also, settings that abstract from internal computation remain to be compared.

Concerning the practical results on state diagrams, our state diagram variant should be extended by further features, as discussed in Section 6.6, thereby improving its flexibility in the top-down development process. Especially the use of contracts, i.e. logical annotations expressing constraints for later refinement steps, appears promising. Finally, we leave it to future work to provide an actual tool and integrate it into existing, or yet to be designed development environments. This, however, is beyond the scope of this thesis.

Appendix A

The modal μ-calculus

The modal μ-calculus [Koz83], as refered to in this thesis, is generated by the following BNF grammar:

$$A ::= \mathtt{tt} \mid \mathtt{ff} \mid Z \mid A \wedge A \mid A \vee A \mid \langle a \rangle\, A \mid [a]\, A \mid \mu Z.A \mid \nu Z.A$$

where $a \in \mathcal{L}$ and Z represents an element from a variable set $\mathcal{V}ar$. For a transition system $M = (S, S^{\mathrm{i}}, \longrightarrow)$, the standard semantic function $[\![\_]\!]_{\_} : \mathcal{F} \times (\mathcal{V}ar \to \mathcal{P}(S)) \to \mathcal{P}(S)$ with respect to $\mathcal{A}$ is defined as:

$$[\![\mathtt{tt}]\!]_\rho = S \qquad [\![\mathtt{ff}]\!]_\rho = \emptyset \qquad [\![Z]\!]_\rho = \rho(Z)$$

$$[\![\phi_1 \wedge \phi_2]\!]_\rho = [\![\phi_1]\!]_\rho \cap [\![\phi_2]\!]_\rho \qquad [\![\langle a \rangle\, \phi]\!]_\rho = \{s \in S \mid \exists s' \in (s.\xrightarrow{a}) : s' \in [\![\phi]\!]_\rho\}$$

$$[\![\phi_1 \vee \phi_2]\!]_\rho = [\![\phi_1]\!]_\rho \cup [\![\phi_2]\!]_\rho \qquad [\![[a]\, \phi]\!]_\rho = \{s \in S \mid \forall s' \in (s.\xrightarrow{a}) : s' \in [\![\phi]\!]_\rho\}$$

$$[\![\mu Z.\phi]\!]_\rho = \bigcap\{\ddot{S} \mid [\![\phi]\!]_{\rho[Z \to \ddot{S}]} \subseteq \ddot{S}\} \qquad [\![\nu Z.\phi]\!]_\rho = \bigcup\{\ddot{S} \mid \ddot{S} \subseteq [\![\phi]\!]_{\rho[Z \to \ddot{S}]}\}$$

Bibliography

[AHK02] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.

[Bae05] Jos C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, 2005.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Comput. Netw. ISDN Syst.*, 14(1):25–59, 1987.

[BB94] Jos C. M. Baeten and Jan A. Bergstra. Process algebra with partial choice. In Bengt Jonsson and Joachim Parrow, editors, *CONCUR '94*, volume 836 of *LNCS*, pages 465–480. Springer, 1994.

[BBK87] Jos C. M. Baeten, Jan A. Bergstra, and Jan W. Klop. Ready-trace semantics for concrete process algebra with the priority operator. *Comput. J.*, 30(6):498–506, 1987.

[BHR84] Stephen D. Brookes, Charles A. R. Hoare, and Andrew W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.

[BIM95] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. *J. ACM*, 42(1):232–268, 1995.

[BK85] Jan A. Bergstra and Jan W. Klop. Algebra of communicating processes with abstraction. *Theor. Comput. Sci.*, 37:77–121, 1985.

[BL92] Gérard Boudol and Kim G. Larsen. Graphical versus logical specifications. *Theor. Comput. Sci.*, 106(1):3–20, 1992.

[BPS01] Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors. *Handbook of Process Algebra*. Elsevier, 2001.

[BS98] Sébastien Bornot and Joseph Sifakis. On the composition of hybrid systems. In Thomas A. Henzinger and Shankar Sastry, editors, *HSCC '98*, volume 1386 of *LNCS*, pages 49–63. Springer, 1998.

[CD05a] Michelle L. Crane and Jürgen Dingel. On the semantics of UML state machines: Categorization and comparison. Technical report, Queen's University, 2005.

[CD05b] Michelle L. Crane and Jürgen Dingel. UML vs. classical vs. Rhapsody statecharts: Not all models are created equal. In Lionel C. Briand and Clay Williams, editors, *MoDELS '05*, volume 3713 of *LNCS*, pages 97–112. Springer, 2005.

[CGJ$^+$03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking.* MIT Press, 2001.

[CH93] Rance Cleaveland and Matthew Hennessy. Testing equivalence as a bisimulation equivalence. *Form. Asp. Comput.*, 5:1–20, 1993.

[CS01] Rance Cleaveland and Oleg Sokolsky. Equivalence and preorder checking for finite-state systems. In Bergstra et al. [BPS01], chapter 6, pages 391–424.

[DC03] Jim Davies and Charles Crichton. Concurrency and refinement in the Unified Modeling Language. *Form. Asp. Comput.*, 15(2–3):118–145, 2003.

[DGG97] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM TOPLAS*, 19(2):253–291, 1997.

[DHP02] Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical graph transformation. *J. Comput. Syst. Sci.*, 64(2):249 – 283, 2002.

[DN04] Dennis Dams and Kedar S. Namjoshi. The existence of finite abstractions for branching time model checking. In *LICS '04*, pages 335–344. IEEE Computer Society Press, 2004.

[DN05] Dennis Dams and Kedar S. Namjoshi. Automata as abstractions. In Radhia Cousot, editor, *VMCAI '05*, volume 3385 of *LNCS*, pages 216–232. Springer, 2005.

[EF02] Rik Eshuis and Maarten M. Fokkinga. Comparing refinements for failure and bisimulation semantics. *Fundam. Inform.*, 52(4):297–321, 2002.

[Eng85] Joost Engelfriet. Determinacy $\rightarrow$ (observation equivalence $=$ trace equivalence). *Theor. Comput. Sci.*, 36:21–25, 1985.

[FdFELS09] Harald Fecher, David de Frutos-Escrig, Gerald Lüttgen, and Heiko Schmidt. On the expressiveness of refinement settings. In *FSEN '09*, 2009. To appear in LNCS.

[FG07] Harald Fecher and Immo Grabe. Finite abstract models for deterministic transition systems: Fair parallel composition and refinement-preserving logic. In Farhad Arbab and Marjan Sirjani, editors, *FSEN '07*, volume 4767 of *LNCS*, pages 1–16. Springer, 2007.

[FH06] Harald Fecher and Michael Huth. Ranked predicate abstraction for branching time: Complete, incremental, and precise. In Susanne Graf and Wenhui Zhang, editors, *ATVA '06*, volume 4218 of *LNCS*, pages 322–336. Springer, 2006.

[FHSS09] Harald Fecher, Michael Huth, Heiko Schmidt, and Jens Schönborn. Refinement sensitive formal semantics of state machines with persistent choice. In *AVoCS '07*, volume 250 of *ENTCS*, pages 71–86. Elsevier, 2009.

[FS05] Harald Fecher and Martin Steffen. Characteristic μ-calculus formula for an underspecified transition system. In *EXPRESS '04*, volume 128 of *ENTCS*, pages 103–116. Elsevier, 2005.

[FS07a] Harald Fecher and Heiko Schmidt. Comparing disjunctive modal transition systems with an one-selecting variant. *J. Logic Algebr. Progr.*, 77:20–39, 2007.

[FS07b] Harald Fecher and Heiko Schmidt. Process algebra having inherent choice: Revised semantics for concurrent systems. In *SOS '07*, volume 192 of *ENTCS*, pages 45–60. Elsevier, 2007.

[FS07c] Harald Fecher and Jens Schönborn. UML 2.0 state machines: Complete formal semantics via core state machines. In Luboš Brim, Boudewijn Houverkort, Meucker Leucker, and Jaco van de Pol, editors, *FMICS '06 and PDMC '06*, volume 4346 of *LNCS*, pages 244–260. Springer, 2007.

[FSKdR05] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem-Paul de Roever. 29 new unclarities in the semantics of UML 2.0 state machines. In Kung-Kiu Lau and Richard Banach, editors, *ICFEM '05*, volume 3785 of *LNCS*, pages 52–65. Springer, 2005.

[GJ03] Patrice Godefroid and Radha Jagadeesan. On the expressiveness of 3-valued models. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *VMCAI '03*, volume 2575 of *LNCS*, pages 206–222. Springer, 2003.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *CAV '97*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[GTW02] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *LNCS*. Springer, 2002.

[Har87] David Harel. Statecharts: A visual formulation for complex systems. *Sci. Comput. Program.*, 8:231–274, 1987.

[HH06] Altaf Hussain and Michael Huth. Automata games for multiple-model checking. In *MFPS '05*, volume 155 of *ENTCS*, pages 401–421. Elsevier, 2006.

[HJ90] Hans Hansson and Bengt Jonsson. A calculus for communicating systems with time and probabilities. In *Real-Time Systems Symposium '90*, pages 278–287. IEEE Computer Society Press, 1990.

[HJ94] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Asp. Comput.*, 6(5):512–535, 1994.

[HK04] David Harel and Hillel Kugler. The Rhapsody semantics of statecharts. In Hartmut Ehrig, Werner Damm, Martin Große-Rhode, Wolfgang Reif, Eckehard Schnieder, and Engelbert Westkämper, editors, *Integration of Software Specification Techniques for Applications in Engineering*, volume 3147 of *LNCS*, pages 325–354. Springer, 2004.

[HM85] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.

[HN96] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[Hoa80] Charles A. R. Hoare. A model for communicating sequential processes. In *On the Construction of Programs*, pages 229–254. Cambridge University Press, 1980.

[HRW07] Hardi Hungar, Oliver Robbe, and Boris Wirtz. Safe UML - restricting UML for the development of safety-critical systems. In Eckehard Schnieder and Geza Tarnai, editors, *FORMS/FORMAT '07*, pages 467–475, 2007.

[JHK02] David N. Jansen, Holger Hermanns, and Joost-Pieter Katoen. A probabilistic extension of UML statecharts. In Werner Damm and Ernst-Rüdiger Olderog, editors, *FTRTFT '02*, volume 2469 of *LNCS*, pages 355–374. Springer, 2002.

[JL91] Bengt Jonsson and Kim G. Larsen. Specification and refinement of probabilistic processes. In *LICS '91*, pages 266–277. IEEE Computer Society Press, 1991.

[JW95] David Janin and Igor Walukiewicz. Automata for the modal μ-calculus and related results. In Jiří Wiedermann and Petr Hájek, editors, *MFCS '95*, volume 969 of *LNCS*, pages 552–562. Springer, 1995.

[KNP06] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Game-based abstraction for Markov decision processes. In *QEST '06*, pages 157–166. IEEE Computer Society, 2006.

[Koz83] Dexter Kozen. Results on the propositional μ-calculus. *Theo. Comput. Sci.*, 27:333–354, 1983.

[KPP08] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The Epsilon transformation language. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *ICMT '08*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008.

[LB98] Kevin Lano and Juan Bicarregui. UML refinement and abstraction transformations. In *Second Workshop on Rigorous Object Orientated Methods: ROOM 2*, 1998.

[LN04] Natalia López and Manuel Núñez. An overview of probabilistic process algebras and their equivalences. In Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, Joost-Pieter Katoen, and Markus Siegle, editors, *Validation of Stochastic Systems*, volume 2925 of *LNCS*, pages 89–123. Springer, 2004.

[LNW07] Kim G. Larsen, Ulrik Nyman, and Andrzej Wasowski. On modal refinement and consistency. In Luís Caires and Vasco T. Vasconcelos, editors, *CONCUR '07*, volume 4703 of *LNCS*, pages 105–119. Springer, 2007.

[LS91] Kim G. Larsen and Arne Skou. Bisimulation through probabilistic testing. *Inform. Comput.*, 94(1):1–28, 1991.

[LT88] Kim G. Larsen and Bent Thomsen. A modal process logic. In *LICS '88*, pages 203–210. IEEE Computer Society Press, 1988.

[LX90] Kim G. Larsen and Liu Xinxin. Equation solving using modal transition systems. In *LICS '90*, pages 108–117. IEEE Computer Society Press, 1990.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[MC01] Mila E. Majster-Cederbaum. Underspecification for a simple process algebra of recursive processes. *Theor. Comput. Sci.*, 266(1-2):935–950, 2001.

[MCR04] Clare E. Martin, Sharon A. Curtis, and Ingrid Rewitzky. Modelling nondeterminism. In Dexter Kozen, editor, *MPC '04*, volume 3125, pages 228–251. Springer, 2004.

[Mil80] Robin Milner. *A Calculus for Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[MNB04] Sun Meng, Zhang Naixiao, and Luis S. Barbosa. On semantics and refinement of UML statecharts: a coalgebraic view. In *SEFM '04*, pages 164–173. IEEE Computer Society Press, 2004.

[Obj07] Object Management Group. *UML 2.1.2 Superstructure Specification*, 2007.

[OH86] Ernst-Rüdiger Olderog and Charles A. R. Hoare. Specification-oriented semantics for communicating processes. *Acta Inform.*, 23(1):9–66, 1986.

[Par81] David Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, volume 104 of *LNCS*, pages 167–183, 1981. Springer.

[Phi87] Iain Phillips. Refusal testing. *Theor. Comput. Sci.*, 50(3):241–284, 1987.

[RK96] Bernhard Rumpe and Cornel Klein. Automata describing object behavior. In *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, pages 265–286. Kluwer Academic Publishers, 1996.

[RRS06] Atle Refsdal, Ragnhild K. Runde, and Ketil Stølen. Underspecification, inherent nondeterminism and probability in sequence diagrams. In Roberto Gorrieri and Heike Wehrheim, editors, *FMOODS '06*, volume 4037 of *LNCS*, pages 138–155. Springer, 2006.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme.* Dissertation, Technische Universität München, 1996.

[SAM96] F. G. Shi, J. M. Armstrong, and J. A. McDermid. A safe subset of statecharts for safety-critical applications. Technical report, University of York, 1996.

[Sch98] Peter Scholz. A refinement calculus for statecharts. In Egidio Astesiano, editor, *FASE '98*, volume 1382 of *LNCS*, pages 285–301. Springer, 1998.

[Seg06] Roberto Segala. Probability and nondeterminism in operational models of concurrency. In Christel Baier and Holger Hermanns, editors, *CONCUR '06*, volume 4137 of *LNCS*, pages 64–78. Springer, 2006.

[SG04] Sharon Shoham and Orna Grumberg. Monotonic abstraction-refinement for CTL. In Kurt Jensen and Andreas Podelski, editors, *TACAS '04*, volume 2988 of *LNCS*, pages 546–560. Springer, 2004.

[SG06] Sharon Shoham and Orna Grumberg. 3-valued abstraction: More precision at less cost. In *LICS '06*, pages 399–410. IEEE Computer Society Press, 2006.

[SHRS96] Sandeep K. Shukla, Harry B. Hunt III, Daniel J. Rosenkrantz, and Richard Edwin Stearns. On the complexity of relational problems for finite state processes (extended abstract). In Friedhelm Meyer auf der Heide and Burkhard Monien, editors, *ICALP '96*, volume 1099 of *LNCS*, pages 466–477. Springer, 1996.

[Tho87] Bent Thomsen. An extended bisimulation induced by a preorder on actions. Master's thesis, Aalborg University Centre, 1987.

[UML] UML 2 semantics project. `http://www.cs.queensu.ca/~stl/internal/uml2/`.

[Val98] Antti Valmari. The state explosion problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *LNCS*, pages 429–528. Springer, 1998.

[vdB02] Michael von der Beeck. A structured operational semantics for UML-statecharts. *Softw. Syst. Model.*, 1(2):130–141, 2002.

[VDN98] Simone Veglioni and Rocco De Nicola. Possible worlds for process algebras. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR '98*, volume 1466 of *LNCS*, pages 179–193. Springer, 1998.

[vG01] Rob J. van Glabbeek. The linear time – branching time spectrum I: the semantics of concrete, sequential processes. In Bergstra et al. [BPS01], chapter 1, pages 3–99.

[Wil01] Thomas Wilke. Alternating tree automata, parity games, and modal μ-calculus. *Bull. Soc. Math. Belg.*, 8(2), May 2001.

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4):221–227, 1971.

[WM97] Michal Walicki and Sigurd Meldal. Algebraic approaches to nondeterminism: an overview. *ACM Comput. Surv.*, 29(1):30–81, 1997.

Index