Acknowledgements

After more than five years of work, there it is. Finally. The thesis contains more than I hoped for even one year ago.

Without the support of many colleagues, friends and relatives, I had not finished it. It is a good custom to mention my promotor and daily supervisor, Roel Wieringa, first. He is a great motivator and still allowed me to go my own way, even when I decided to do the last part of the work with Joost-Pieter Katoen, who is assistant-promotor now, and Holger Hermanns. My roommates, Rik Eshuis and Wijnand Derks, more and less quiet, often served as first hearers and commentators. Angelika Mader shared her research in the case study with me. I could mention many other names, mainly in the Information Systems and the Databases groups.

I would also like to thank the committee members for their reading and kind attention: Ed Brinksma, David Harel, Boudewijn Haverkort, Jozef Hooman, John-Jules Meyer, and Henk Zijm, in addition to the already mentioned Roel Wieringa, Joost-Pieter Katoen, and Holger Hermanns.

From the non-scientific side, I would like to put my relatives first: my parents, Käti and Klaas Jansen-Luginbühl, who supported and encouraged me. I thank them for their love and their education. During these last weeks, my brother Simon and his partner Christina Siu have helped me with the graphical design of the thesis.

I have made the acquaintance with and got support from many other people living in Twente, too many to mention. I only want to thank my paranymphs, Hans Boschker and Rob Kosman, here. Finally, I hope never to forget the One who stands at the beginning and end of everyone and everything, without whom we could not live and, for example, write a Ph. D. thesis.

The research in this thesis was partially supported by the DEIRDRE project, NWO 612-323-419, and the Esprit Working Group ASPIRE, contract number 22704.
## Contents

1 Introduction ........................................... 11
   1.1 Statecharts ........................................... 11
      1.1.1 History ........................................ 12
      1.1.2 Use and Interpretation of Statecharts .......... 12
   1.2 Real Time ........................................... 14
   1.3 Probabilities and Randomness ......................... 15
      1.3.1 Uses of Probability .............................. 15
      1.3.2 Some Distinctions ................................ 16
   1.4 Model Checking ....................................... 17
   1.5 Contributions of this Thesis ......................... 17
      1.5.1 Probabilistic Choice and Stochastic Timing .... 17
      1.5.2 A Property Language for Statecharts .......... 18
      1.5.3 Case Study .................................... 18
      1.5.4 In Brief ....................................... 19

2 Statecharts and the UML ............................... 21
   2.1 Input–Output Automata and Output Automata .......... 21
      2.1.1 Composition of Input–Output Automata ........... 22
      2.1.2 Open and Closed Systems ......................... 23
   2.2 Statecharts: Syntax .................................. 23
      2.2.1 Syntax .......................................... 24
      2.2.2 Drawing a Statechart ............................. 25
   2.3 Statecharts: Semantics ................................ 26
      2.3.1 Step Construction ................................. 26
      2.3.2 Step Execution .................................. 28
      2.3.3 Input–Output Automaton Semantics ............... 28
   2.4 Extensions of the above Statecharts ................. 31
      2.4.1 Local Variables ................................ 31
      2.4.2 Real Time ...................................... 31
   2.5 Unified Modelling Language ......................... 32
   2.6 TCM .................................................. 33
   2.7 Related Work ....................................... 33
      2.7.1 Statecharts .................................... 33
      2.7.2 UML ........................................... 35
## Contents

3 Real-Time and Action Language: ATCTL

3.1 Introduction ................................................. 37
3.2 Computation Tree Logic ..................................... 38
  3.2.1 Semantics .............................................. 39
  3.2.2 CTL Model Checking .................................. 40
3.3 System Model ................................................. 40
3.4 Action Logic ................................................. 40
3.5 Real Time Model ............................................. 41
  3.5.1 Timed Input–Output Automata ......................... 42
3.6 Action-Based Timed CTL .................................... 43
3.7 Timed CTL ................................................... 46
3.8 Reduction of ATCTL to TCTL ....................... 47
3.9 Action-Based CTL ........................................... 51
3.10 Reduction of ATCTL to ACTL ................. 53
  3.10.1 Translation of Models ................................. 54
  3.10.2 Translation of States, Steps and Runs ............ 55
  3.10.3 Translation of Formulas ............................. 56
  3.10.4 Equivalence theorem ................................. 56
3.11 Alternative Translation .................................. 57
3.12 Comparison of the Reductions .................. 58
3.13 Example: Internal Travel Office .............. 59
3.14 Related Work ............................................... 62
3.15 Conclusion ................................................ 62

4 Probabilistic Extension: P-Statecharts

4.1 Introduction ................................................. 65
4.2 Probabilistic UML Statecharts ..................... 66
4.3 P-Statechart Semantics .................................... 67
  4.3.1 Step Construction ..................................... 68
  4.3.2 Order of Probabilistic and Nondeterministic Choices in Algorithm 4.3.1 ................................. 70
  4.3.3 Step Execution ......................................... 72
  4.3.4 BPTS Semantics ....................................... 72
  4.3.5 Markov Decision Process Semantics ................ 75
4.4 Comparison to Non-Probabilistic Semantics .... 78
4.5 Property Specification for P-Statecharts ......... 83
4.6 Example: Hawks and Doves ......................... 85
  4.6.1 Scenarios checked ..................................... 86
4.7 Example: Gambling Machine ...................... 88
  4.7.1 Desired Properties Checked ......................... 91
4.8 Related Work ............................................... 93
4.9 Discussion and Conclusion ......................... 93
Chapter 1

Introduction

1.1 Statecharts

“Statecharts” are a graphical language to describe the behaviour of a discrete-state system. They are based on the exchange of messages, or “events”, between the system and its environment. A statechart is a drawing that shows how the system reacts to the reception of an event or a specific sequence of events and what events the system generates.

Statecharts are often used to specify the desired behaviour of a computer system under construction, i.e., they are used to describe what a system that will be built should do.

We illustrate this by means of an example: Figure 1.1 on the next page contains a statechart for a workflow system that supports a car damage assessor. He has to assess on behalf of an insurance company whether a damaged car should be repaired and whether the garage offers an acceptable price for the repair. If the estimated damage is beyond € 1 000, the assessor has to inspect the car physically before repair. The workflow system will support the car damage assessor in his task and should reflect the work procedure prescribed to the assessor by his employer.

The statechart has the following intuitive meaning: Each state is drawn as a box with rounded corners; arrows represent possible state changes. The workflow system starts in state Idle. When it receives the event start, it proceeds to Contacting garage. In this state, it waits for the event estimate damage. On receipt of the event, it checks a condition: if the estimated damage is less than € 1 000, the system proceeds to state Phone assessment. Otherwise, the system proceeds to Physical assessment. Both states are substates of Assessing; the statechart language shows this by drawing the superstate as a larger box that encloses the substates. Superstates can be used to simplify the presentation of common behaviour: in both states, the workflow system reacts to the event assessed in the same way. There are two arrows leaving the state Assessing and labelled with (input) event assessed. These two arrows are in conflict, as they point to different states to be entered. So, if the system receives the event assessed, it chooses (nondeterministically) one of the two arrows and executes it. Accordingly, it either generates the event write off and sends it to the component with identifier 2, or it generates the event repair and sends it to the same component. Then, it enters the state Repairing and some of its substates. Repairing is split into two by a dashed line. This indicates parallel behaviour: if the system is in state Repairing, it is in both substates – Invoice handling and Reporting – also. In each of these substates, an initial
state is indicated: Waiting for invoice and Writing report, respectively. The initial state is entered if the arrow does not point explicitly to a different state.

1.1.1 History

Statecharts were developed by Harel [43] as an extension of Mealy and Moore state machines [79, 82]. Statecharts extend Mealy machines in two ways, so that the behaviour model can be structured: With parallelism, one can clearly separate the behaviours of (almost) independent components of a system. By means of hierarchy, one can refine a behaviour: first, one describes the main structure of a system, and later, one adds substates to describe details.

These extensions make statecharts suitable in situations where simple state diagrams would become cluttered.

Full statecharts contain more features, like entry and exit actions or history connectors. Entry actions are similar to how Moore machines produce output.

1.1.2 Use and Interpretation of Statecharts

Statecharts were originally defined in the context of structured analysis, but they have been adopted by the Unified Modelling Language (UML) [83] to model the behaviour of objects. The UML is a set of languages to describe the design and implementation of a software system. It is often used in object-oriented software engineering. As statecharts are one of the parts of the UML, they are well-known to professional software engineers. Also in other contexts, statecharts are used widely, as can be seen from the extensive literature on the subject. A choice can be found in the related work section 2.7.
1.1. Statecharts

Requirements- and implementation-level. There are two fundamentally different ways to use statecharts, illustrated here with the CASE tools STATEMATE and Rhapsody. These ways to use are related to two different semantics: requirements-level and implementation-level. The uses intended by Harel [43] become clear from the CASE tool STATEMATE [46]. With STATEMATE, one models a system using several activities, of which a single one is the control activity. It controls the other activities; how it does so is described using a (single) statechart. STATEMATE is intended to specify the essential, i.e., the implementation-independent behaviour of a system: every implementation of the system has to satisfy the given description. Therefore, STATEMATE uses a requirements-level semantics of statecharts. A requirements-level semantics abstracts from limitations of current computers in speed and memory to simplify the conceptual model of a system. This is sometimes called the “perfect technology” assumption [78].

In the UML, the use of statecharts can be illustrated with the CASE tool Rhapsody [88]. In Rhapsody, a statechart can be attached to each object class. The objects in a class all behave according to its statechart. So, the behaviour of the overall system is the composition of all these statecharts. Rhapsody is intended to support the coding of a system; therefore, it uses an implementation-level semantics of statecharts, mapping a statechart to a C++ program fragment. As opposed to a requirements-level semantics, an implementation-level semantics models the behaviour of a computer and its limitations as faithfully as possible.

Variety of statechart semantics. There are more choices in statechart semantics than the distinction between requirements- and implementation-level semantics. This give rise to a large number of semantics, each adapted to a specific situation (see von der Beeck [8] for an overview of some). Consequently, this leads to misunderstandings when several engineers have different semantics in mind while speaking about the same statechart.

Our semantic choices. In this dissertation, we have taken the semantics of Eshuis and Wieringa [33] as a basis. It is an object-oriented requirements-level semantics that is based on the STATEMATE semantics of statecharts [46] and its formalisation by Damm et al. [21]. We have taken this approach because it is in accordance with our intended use of the UML: we use the UML to provide specifications of computer systems.

The object-oriented character can be seen from the following choices: we do not describe the behaviour centrally, in a single statechart, but describe each object’s behaviour individually. The object comprehends both behaviour and data, accessible as local variables to the statechart. Messages are always directed towards a single receiving object (as opposed to STATEMATE, where messages are sent to a channel and are received by every component that reads that channel).

Several semantic choices are consequences of the fact that we use statecharts at the requirements level and assume “perfect technology”: The system has unlimited memory and is infinitely fast\(^1\). Communication is instantaneous and arrives always. There is a single global clock that always shows the correct time. Actions are executed instantaneously.

\(^1\)In practice, the latter means: the system is so fast that the environment does not perceive any relevant delay.
and a arbitrary number of actions can be executed concurrently. A statechart reacts to all inputs it receives at once; therefore, no input queue is needed.

We have, in addition, made some choices that are intuitive to us, but are not directly related to the object-oriented or requirements-level aspects. Our semantics is event-driven. That means, a statechart reacts to an event it receives immediately. Some other semantics are clock-driven: the statecharts would collect events until the clock ticks and then react to all collected events at once. (Eshuis and Wieringa [33] also define a clock-driven variant of their semantics.) The perfect technology assumption and the event-driven character together imply that our semantics is a super-step semantics, i.e., it handles all internally generated events immediately, before waiting for more input from the environment.

Our semantics is sequential-step, i.e., if the system generates an internal event in one step of the statecharts, it reacts to that event in the next step (and not in the same step, as some others do). Therefore, all communication is asynchronous.

Because the extensions of statecharts defined in this thesis do not depend heavily on these semantic choices, we think that they can be adapted easily to other semantics.

1.2 Real Time

In addition to statecharts to describe models of discrete-state systems, we use logics to describe desired properties of these systems. Many logics and models we use are temporal in nature, that means, they express some development over time. With “real-time”, we express the fact that a logic or a model is able to quantify the amount of time that passes between events.

Real-time is needed in situations where not only the order of events is important, but also the amount of time between events. For example, in the damage assessor workflow, it is not only important that the assessor and the garage do their work, but that they finish it within some reasonable time, say, four days.

There are several ways to express the real-time aspect [64]. We make the following distinctions:

Discrete or continuous? Time can be measured in a discrete or continuous way: In discrete real-time, the time is modelled similar to the natural numbers. Actions only take place at one of these time points. This is a suitable manner to model time for synchronised systems. For example, most hardware chips change state only at the moment they receive a clock pulse. A discrete time model is sometimes also chosen to simplify reasoning.

In continuous real-time, the time is modelled similar to the real numbers. Actions can take place at any moment. Continuous real-time can be used to model the environment of a system or to describe a system at a different level of abstraction.

We have chosen to use the continuous time model because it is nearer to our intuitive understanding of time.

Branching or linear? The possible futures from some state of the system can be constructed in two ways: In branching temporal logics, we may choose at every moment in
time out of the various possible futures. The corresponding structures can be seen as trees: at each branching, there is a choice.

In linear temporal logics, we choose once and for all a single run (a complete future) and then treat time as if each moment in time has a unique possible future.

There is much debate on the advantages and disadvantages of both choices. For example, the computational complexity of branching time model checking is lower than of linear time model checking [95]. Vardi [100], however, argues that this difference has little relevance in practice and that the limitations of branching time make it less suitable for model checking than linear time. We choose to use branching time logics for pragmatic reasons: the model checkers we use are based on branching time.

1.3 Probabilities and Randomness

Probabilistic descriptions quantify the amount of (un)certainty in some situation. Probabilities are normally assigned to some outcome of an experiment, for example the outcome “one is up” of the experiment “throwing a die”. Values are real numbers in the range 0 . . . 1: an impossible outcome has probability 0, a certain outcome 1. The values in between show how (un)certain the respective outcome is. See appendix A for a short introduction into mathematical probability theory.

1.3.1 Uses of Probability

Probabilistic description can be useful in requirements engineering and specification of software systems as follows:

Quality of service. This subsumes aspects like the expected duration of some process, how likely the process will terminate within a given time, and the probability of success. For example, the insurance company that uses the workflow in figure 1.1 might require that its damage assessors finish their work within 4 days for 95 % of the cases.

Varying workload. The environment imposes some workload on the system. Often, this workload is not constant, but varies with time. For example, car accidents happen more often in bad weather. A probabilistic description of the environment can account for this variance.

Randomised algorithms. In some protocols for distributed systems, randomisation is used to make a symmetric situation asymmetric. For example, the so-called “root contention” in IEEE 1394 [97] is a problem where asymmetry (namely, there is a single root in the network) has to be produced out of a symmetric situation (namely, there are two candidates for the root).

Abstraction. In some cases, it is enough to describe the likelihood that some particular choice is made, without detailing the exact data dependencies. For example, we can abstract from the exact damage estimate by just stating that a physical inspection is needed in 30 % of the damage cases, without detailing which accidents exactly.
Chapter 1. Introduction

Unreliable environments and fault-tolerant systems. A fault-tolerant system needs to interact correctly with an environment that only behaves most of the time. We can quantify “most of the time” using probabilities.

1.3.2 Some Distinctions

Probabilistic choice and stochastic timing. In this dissertation, we concentrate on two forms of randomness: In probabilistic choice, it is uncertain which possibility out of a discrete set the system will take. A probability space over this set describes how likely the single possibilities are. Throwing a die or tossing a coin are examples of probabilistic choice experiments. Probabilistic choice is typically used in randomised algorithms.

In stochastic timing, it is uncertain how long a specific process will take. The timing is then described by a probability space over the (nonnegative) real numbers. Stochastic timing is typically used when evaluating quality of service or varying workload. For example, how long somebody phones typically varies with time; a stochastic description can serve to predict the probability of his future behaviour. Figure 1.2 shows a possible distribution of phone calls over time.

Both forms can be described in general by standard probability theory. We are going to use both concepts: in chapter 4, we will define an extension of statecharts by probabilistic choice; in chapter 5, we will define an extension by stochastic timing.

Probability and (non)determinism. A random phenomenon to which a probability can be assigned is called probabilistic. A trivial probabilistic choice (where a single outcome has probability 1) is called deterministic. If one cannot or does not want to assign a probability to a phenomenon while it is not deterministic, the phenomenon is called nondeterministic.

We will allow for both probabilistic and nondeterministic choice in our extensions of statecharts.

System and environmental randomness. Each reactive system is exposed to external stimuli that exhibit some kind of randomness. We call this environmental randomness. On
the other hand, the system behaviour itself may ask for a probabilistic description, either because the system embodies a randomised algorithm, or because probability is used for abstraction. We call this system randomness.

Chapter 4 will address system randomness; chapter 5 will address environmental randomness.

1.4 Model Checking

In the past 20 years, model checking has been developed as a technique for verifying properties of computer systems [30, 19, 18]. Model checking consists of verifying whether a property (typically expressed in some modal extension of propositional logic) is true in a Kripke structure or labelled transition system (LTS), which represents the possible behaviours of a software system.

One of the advantages of model checking is that a negative answer always takes the form of a counterexample of the property to be verified, usually in the form of a trace through the transition system. This counterexample gives a lot of information to the analyst about the LTS.

Another advantage is that model checking does not require a creative effort from the verification engineer. The proof that the desired property is true, or the proof that it is false, is found automatically. (Still, the verification engineer has to enter the correct parameters to optimise the process; this may require quite some effort, as Ruys [90] has shown.)

There are several model checkers available, each with its own languages to specify the LTS and the desired properties. The property languages are usually variants of computation tree logic (CTL) [30] or sometimes linear temporal logic (LTL) [75].

1.5 Contributions of this Thesis

The goal of this thesis is:

To use statecharts to render model checking more widely usable.

We show this in two respects: For probabilistic model checking, we provide an extension of statecharts as input language. For traditional real-time statecharts, we provide a property language that fits nicely with the features of statecharts.

1.5.1 Probabilistic Choice and Stochastic Timing

Probabilistic and stochastic modelling has often been separated from functional behaviour modelling [37]: there are separate models for the performance aspect and the functional aspect of a system under development. However, this has the disadvantage that there may be small semantic differences between these two models.

We want to provide software engineers with a language that is suitable to both functional modelling and modelling of performance and other aspects mentioned in section 1.3.1. To this end, we define an extension of statecharts with probabilistic choice (in chapter 4) and stochastic timing (in chapter 5). Traditional statecharts neither include
Chapter 1. Introduction

probabilistic choice nor stochastic timing, although, interestingly enough, Harel mentioned the possibility of probabilistic choice already in his seminal article [43].

The extensions we provide have formal semantics which extend a formal semantics of (plain) statecharts. With these formal semantics, we can verify a model formally against a desired property. In both extensions, we use model checking as our verification method.

The probabilistic extension can serve to describe randomised algorithms and abstraction. For example, the hawk–dove-game in section 4.6 is modelled using probabilistic statecharts.

The stochastic extension can serve to describe quality of service and varying workload. For example, we can modify the damage assessor workflow in figure 1.1 so that it includes the relevant probabilistic and stochastic information. We will see this in section 5.2.2. Then, we can verify a requirement of the extended statechart like: “The damage assessor finishes his work within 4 days in 95% of the damage cases.”

One could object that there are already specialised languages to describe probabilistic systems, for example Markov chains and Markov decision processes [86], and these languages can (in principle) also be used for a functional model. However, these languages provide no means to structure a specification like statecharts do with hierarchy and parallelism. Therefore, Markov chains and Markov decision processes can only be used in very simple system specifications.

This work has been published together with Holger Hermanns and Joost-Pieter Katoen [56, 55, 58].

1.5.2 A Property Language for Statecharts

Formally specifying desired properties of statecharts is not so easy, as the logics currently in use are not tailored to the rich language of statecharts. A suitable logic should include means to express properties of states and of actions. For real-time statecharts, it should also be able to express real-time requirements.

Most current logics allow for the expression of state and action properties only in a half-hearted way: some choose for state properties as the basic notion and require to circumscribe action properties. Other logics choose for actions as the basic notion and require to express state properties by special test actions.

Chapter 3 describes a combination of logics, called ATCTL ("action + time CTL"), that provides all these means to the engineer: action properties, state properties, and real-time properties. It is based on computation tree logic (CTL), a logic that is widely used for model checking, as it allows for rather efficient model checkers. We show that ATCTL properties can be verified against (the semantic model of) a statechart using model checking.

This work has been published together with Roel Wieringa [60, 104].

1.5.3 Case Study

In chapter 6, we apply the extensions to a somewhat larger case study. The @HA project of the DIES group at the Universiteit Twente aims to integrate home appliances into one coherent distributed architecture. One of its main objectives is to design, build and evaluate one common integrated network for entertainment (e.g., video), control (e.g., heat-
ing) and information (e.g., WWW). The HOTnet or RTnet protocol [92, 91] is a real-time communication protocol designed to be used in the @HA network.

The RTnet protocol is token-based. The node that has the token may use the network for some predefined time and then passes it on to the next node. If the intended next node has died, the sender will find out after some time. In that case, the sender removes the next node from the list of participants.

Given that communication between the nodes fails with a certain probability, it may happen that a node is removed from the list of participants in error. We will investigate the probability that this happens.

On the other hand, the RTnet protocol includes an announcement protocol mainly intended to add new nodes to the list of participants. Nevertheless, the announcement protocol can also be used to reintegrate nodes into the network that have been removed erroneously. Also here, we will investigate the probability that this happens.

This work has not been published so far.

1.5.4 In Brief

Summarizing, the main contributions of this thesis are:

- A logic for requirements of statecharts (including actions, states, and real-time) – chapter 3
- An extension of statecharts with probabilistic choice – chapter 4
- An extension of statecharts with stochastic timing – chapter 5

In all three cases, we show that formal verification of the models (statecharts or extended statecharts) against desired properties (in ATCTL or in a probabilistic logic) is possible: model checking is the verification method we use.

The usability is shown at the hand of several smaller examples and a somewhat larger, practical case study – chapter 6.
Chapter 1. Introduction
Chapter 2

Statecharts and the UML

In this chapter, we introduce the basic statechart notation with a semantics adapted from Eshuis and Wieringa [33] which is based on the STATEMATE semantics of statecharts [46] and its formalisation by Damm et al. [21]. Originally, Eshuis and Wieringa translated a collection of statecharts to a Kripke structure; we have chosen to translate it to an input–output automaton (a Kripke structure is a kind of simple input–output automaton). We also show how statecharts are embedded in the UML.

2.1 Input–Output Automata and Output Automata

Before we introduce statecharts, we present the basic automaton model that we will use in the rest of the thesis. It also serves to define a formal semantics of statecharts. We adapt the definition of an input–output automaton given by Lynch and Tuttle [74] to our needs. An input–output-automaton (IOA) is a sextuple \((S, A, \Delta, AP, I, s_0)\) where:

- \(S\) is a finite, non-empty set of states.
- \(A\) is a set of actions, partitioned into input and output actions \(A^\text{in}\) and \(A^\text{out}\).
- \(\Delta \subseteq S \times A \times S\) is the transition relation. It has to satisfy input enabledness: For each state \(s \in S\) and each input action \(a \in A^\text{in}\), there exists a state \(t \in S\) such that \((s, a, t) \in \Delta\). We will sometimes write \(s \xrightarrow{a} t\) instead of \((s, a, t) \in \Delta\).
- \(AP\) is a set of atomic propositions.
- \(I : S \rightarrow P (AP)\) is the interpretation of the atomic propositions.
- \(s_0 \in S\) is the initial state.

As we do not need internal actions, we have omitted them from the definition. Internal actions are defined by Lynch and Tuttle [74] and behave almost the same as output actions.

An output automaton is an input–output automaton with an empty set of input actions: \(A^\text{in} = \emptyset\) and \(A = A^\text{out}\). Output automata are similar to Kripke structures; however, it is often required that a Kripke structure be total, i.e., in every state, at least one (output) transition is enabled.
Behaviour described by an input–output automaton. Speaking informally, an input–output automaton is always in one of its states \( s \). One of the enabled transitions that start in \( s \) may be selected for execution. If no transition is enabled, time may pass until some transition will be enabled (if any).

An output transition (labelled with some \( a \in A_{\text{out}} \)) is always enabled, an input transition (labelled with some \( a \in A_{\text{in}} \)) is enabled only if \( a \) is offered by the environment.

The system proceeds from \( s \) to the target indicated by the transition. On taking an output transition, the corresponding action is produced and offered to the environment.

Runs and complete runs. A run of an input–output automaton is a (possibly infinite) sequence of consecutive transitions, where the target of each transition is the source of the next one. We write this as:

\[
s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \rightarrow \ldots
\]

Note that the definition does allow the run to start in a state which is not the initial state.

A complete run is a run that cannot be continued, either because it is infinite or because there is no transition starting in its final state.

2.1.1 Composition of Input–Output Automata

Input–output automata are defined in a way that allows an intuitive composition: the outputs produced by one automaton serve as inputs to the other one, and vice versa. The result of the composition is again an input–output automaton.

If we want to compose two input–output automata \((S_i, A_i, \Delta_i, AP_i, I_i, s_{0,i})\), for \( i = 1, 2 \), the outputs of the automata must not overlap. We call the two automata compatible if \( A_{\text{out}}^1 \cap A_{\text{out}}^2 = \emptyset \).

If the two automata are compatible, their composition is the input–output automaton \((S, A, \Delta, AP, I, s_0)\), defined by:

- \( S = S_1 \times S_2 \)
- \( A = A_1 \cup A_2 \), where
  \[
  A_{\text{out}} = A_{\text{out}}^1 \cup A_{\text{out}}^2 \quad \text{and} \quad A_{\text{in}} = (A_{\text{in}}^1 \cup A_{\text{in}}^2) \setminus A_{\text{out}}
  \]
- The transition relation \( \Delta \) is defined by the rules\(^1\):
  \[
  \begin{align*}
  a &\in A_1 \cap A_2 \quad (s_1, a, s_1') \in \Delta_1 \quad (s_2, a, s_2') \in \Delta_2 \quad \Rightarrow (s_{1,2}, a, (s_1', s_2')) \in \Delta \\
  a &\in A_1 \setminus A_2 \quad (s_1, a, s_1') \in \Delta_1 \quad (s_2, a, s_2') \in \Delta_2 \quad \Rightarrow (s_{1,2}, a, (s_1, s_2')) \in \Delta \\
  a &\in A_2 \setminus A_1 \quad (s_1, a, s_1') \in \Delta_1 \quad (s_2, a, s_2') \in \Delta_2 \quad \Rightarrow (s_{1,2}, a, (s_1', s_2)) \in \Delta \\
  a &\in A_1 \setminus A_2 \quad (s_1, a, s_1') \in \Delta_1 \quad (s_2, a, s_2') \in \Delta_2 \quad \Rightarrow (s_{1,2}, a, (s_1, s_2')) \in \Delta 
  \end{align*}
  \]
- \( AP \) is the disjoint union \( AP_1 \cup AP_2 \).

\(^1\)For a definition of how we write and use rules, see appendix A.
2.2. Statecharts: Syntax

- The interpretation of atomic propositions is defined by the rules:
  \[
  p \in \mathcal{I}_1(s_1) \quad p \in \mathcal{I}_2(s_2) \quad p \in \mathcal{I}(s_1, s_2)
  \]

- \( s_0 = (s_{0,1}, s_{0,2}) \)

2.1.2 Open and Closed Systems

Input–output automata describe an open system: They receive input from their environment and may react to it. Therefore, open systems can be composed to larger systems.

Output automata, on the other hand, are a model to describe closed systems: They are assumed to be complete and do not react to input from their environment. Some analysis methods are more easily applied to closed systems, for example model checking of CTL properties.

In some situations, we assume that an input–output automaton \( IoA \) is not subject to any input from its environment (or alternatively, all the relevant environment is already accounted for in the model). We can then define an output automaton that corresponds to \( IoA \) by dropping all input transitions of the automaton.

Given an input–output automaton \( IoA = (S, A, \Delta, AP, I, s_0) \), we define its closure to be the output automaton \( (S, A', \Delta', AP, I, s_0) \), where

- The set of actions is \( A' = A \setminus A^{in} \), partitioned into:
  \[
  A^{in} = \emptyset \quad A^{out} = A^{out}
  \]

- The transition relation is
  \[
  \Delta' = \Delta \cap (S \times A' \times S)
  \]

In chapters 3 and 4, we will mainly treat closed systems. Therefore, the difference between input and output is less relevant there. Chapter 5 will deal with open systems and their composition again.

2.2 Statecharts: Syntax

To avoid unnecessary repetition of similar definitions, we define basic statecharts here. In the following chapters, we will only define the differences to this definition in detail and refer to this section for more details of the common ground. These extensions are: local variables, real-time, probabilistic choice and stochastic timing.

The definition of statecharts and the semantics given here are adapted from Eshuis and Wieringa [33] who based their work on Harel and Naamad [46] and Damm et al. [21]. They simplify full UML statecharts by omitting some minor aspects like deferred events, history and synchronisation states. We have further simplified it by omitting synchronous communication, local variables, actions on initial transitions and entry and exit actions. All of them can be easily added, but they clutter the semantics. We will describe the extension with local variables in section 2.4.1. We only provide an event-driven, requirements-level semantics.
Eshuis and Wieringa used Kripke structures as semantic model; we have changed this to input–output automata. This preserves the distinction between input and output found in the statechart.

In the following definitions, we assume that there is an object for each \( i \in \{1, \ldots, n\} \). The associated statechart is denoted \( SC_i \).

### 2.2.1 Syntax

A single statechart \( SC_i \) consists of

- A finite set \( \text{Nodes}_i \) of nodes with a tree structure, described by a function \( \text{children}_i : \text{Nodes}_i \rightarrow \mathcal{P}(\text{Nodes}_i) \). (If \( x' \in \text{children}_i(x) \), then \( x \) is the parent of \( x' \). Of course, \( \text{children}_i \) has to fulfil several constraints to make it describe a tree structure. For simplicity, these are omitted here.) Descendants are children or children of descendants. The root is denoted \( \text{root}_i \).

  - The function \( \text{type}_i : \text{Nodes}_i \rightarrow \{\text{BASIC}, \text{AND}, \text{OR}\} \) assigns to every node its type. Nodes that are leaves of the tree have type BASIC; children of AND nodes must have type OR; \( \text{type}_i(\text{root}_i) = \text{OR} \); other nodes have type OR or AND.

  - The function \( \text{default}_i : \{x \in \text{Nodes}_i \mid \text{type}_i(x) = \text{OR}\} \rightarrow \text{Nodes}_i \) identifies for each OR node one of its children as the default (or initial) node.

- A finite set \( \text{Events}_i \) of events. We will use the symbol \( \perp \) to denote “no event required”; \( \perp \notin \text{Events}_i \).

- A set \( \text{Guards}_i \) of guard expressions. Guard expressions are boolean combinations of the atoms \( j.\text{isin}(x) \), for \( j \in \{1, \ldots, n\} \) and \( x \in \text{Nodes}_j \) (with the intuitive meaning “the statechart \( SC_j \) is in node \( x \)”).

- A set \( \text{Actions}_i \) of actions. Actions are \( \text{send} \; j.e \) (for \( j \in \mathbb{N} \) and \( e \in \text{Events}_j \)) with the intuitive meaning “send event \( e \) to the statechart \( SC_j \)”\(^2\).

  - If we consider open systems, it may be that the system itself doesn’t contain \( SC_j \). In that case, we should read “send event \( e \) to the external component identified by \( j \)”.

- A finite set \( \text{Edges}_i \) of edges. An edge is a quintuple \( (X, e, g, A, Y) \) where \( X \subseteq \text{Nodes}_i \) is a non-empty set of source state nodes, \( e \in \text{Events}_i \cup \{\perp\} \) is the triggering event, \( g \in \text{Guards}_i \) is a guard, \( A \subseteq \text{Actions}_i \) is the set of actions executed when the edge is taken, and \( Y \subseteq \text{Nodes}_i \) is a non-empty set of target state nodes.

Note that \( \text{Guards}_i \) and \( \text{Actions}_i \) are defined in terms of the other components. We will therefore denote a statechart simply by \( SC_i = (\text{Nodes}_i, \text{Events}_i, \text{Edges}_i) \). It may happen that several objects have got the same statechart. In that case, not all \( SC_i \) can be distinguished based on these sets; therefore, we will not assume that the system at hand is described by a set of statecharts.

\(^2\)We will abbreviate \( \text{send} \; j.e \) sometimes by \( j.e \).
2.2.2 Drawing a Statechart

We adopt the following drawing conventions. The root node is not drawn. Nodes that are not children of an AND-node are drawn as rectangles with rounded corners. A parent node encloses its children. Children of an AND-node partition the node by dashed lines. Each OR-node encloses a black dot and indicates its default node by an arrow directed from the dot to the default node. An edge with event $e$, guard $g$ and action set $A$ is denoted as an arrow from the source to the target node $e[|g|/A]$. If there are multiple source nodes or multiple target nodes, we draw a multi-tailed or multi-headed arrow.

If the event on an edge is $\bot$, we may omit it: $|g|/A$. Further, if the guard is true, we may omit it: $e/A$. Similarly an empty set of actions may be omitted: $e[|g|]$.

**Example 2.2.1** We repeat the example from the introduction: We want to model the work-flow of a car damage case for the damage assessor. He has to assess on behalf of an insurance company whether a damaged car should be repaired and whether the garage offers an acceptable price for the repair. If the estimated damage is beyond € 1000, the assessor has to inspect the car physically before repair.

Figure 2.1 shows a modified version of the workflow statechart from figure 1.1. In this statechart, we can see most of the elements described above: Idle, Assessing, Repairing, Invoice handling are nodes. Idle is a BASIC node, and it is the default node of root. Assessing is an OR node with two children. Repairing is an AND node with two children, namely Reporting and Invoice handling.

The statechart describes the following behaviour: The workflow starts in node Idle. When it receives the event start, it proceeds to node Contacting garage. Depending on how large the estimated damage is, the workflow changes to one of the nodes Phone assessment or Physical assessment, meaning that the damage assessor does the assessment. After he is...
done, the workflow decides whether to repair or to write off the car. The damage assessor
has to write a report on every case; if the car is repaired, he also has to check the invoice.
These two processes run in parallel; when both are finished, the workflow proceeds to
node Finished.

2.3 Statecharts: Semantics

In this section, we define a simple semantics for statecharts. It is an adaptation of the
requirements-level semantics defined by Eshuis and Wieringa [33] which is based on the
StateMate semantics of statecharts [46] and its formalisation by Damm et al. [21].

The chosen semantics combines the Statemate semantics with communication and
classification. We have chosen it because it is simple and it corresponds to our intended
use of UML models for user-friendly model checking. The most important semantic
choices have been mentioned in section 1.1.2.

Intuitive semantics for a single statechart. The intuitive behaviour of a statechart can
be described as follows. The statechart is always in some state (which consists of one or
several nodes). An edge is enabled if the statechart is in the source node(s), the event
of the edge happens and its guard holds. The system executes as many enabled edges
at once as possible without conflict: it leaves the source nodes, executes the actions and
enters the target nodes of the edge(s).

In the rest of this section, we first define what will be a step. This encompasses the res-
olution of nondeterminism and priorities within a single statechart. Subsequently, these
steps are used as the building blocks in a mapping of a collection of statecharts to an
input–output automaton.

2.3.1 Step Construction

A step is, basically, the set of edges taken simultaneously in one transition of a single
statechart $SC_i$. This section describes how a step is constructed.

A configuration $C_i$ of statechart $SC_i$ is a set of nodes that fulfils the conditions:

- $\text{root}_i \in C_i$.
- If an OR-node is in $C_i$, then exactly one of its children is in $C_i$.
- If an AND-node is in $C_i$, then all its children are in $C_i$.

The set of all configurations of $SC_i$ is denoted $Conf_i$. A state of $SC_i$ is a pair $(C_i, I_i)$ where $C_i$ is a configuration and $I_i \subseteq \text{Events}_i$ is a set of events to which the statechart still has to react. The validity of guard $g$ in a state may depend on the configurations $C_1, \ldots, C_n$. We write $(C_1, \ldots, C_n) \models g$ iff $g$ holds in the state of the collection of statecharts.

An edge $(X, e, g, A, Y)$ is enabled in state $(C_i, I_i)$ if $C_i$ contains its source state nodes $X$, the event $e$ is in the current input set $I_i$ and the guard $g$ holds: $(C_1, \ldots, C_n) \models g$.

We denote the set of enabled edges by $En(C_i, I_i)$.

The scope of an edge $(X, e, g, A, Y)$ is the smallest (in the parent–child-hierarchy) OR-
node that contains both the source nodes $X$ and the target nodes $Y$. We refer to it as $\text{scope}(X, Y)$. (The scope is the smallest node that is not affected when the edge is executed.)
A step is a set of enabled edges that are taken together as a reaction to events. The edges in a step for statechart $SC_i$ depend on its current state $(C_i, I_i)$ (and, for the guards, on the configurations of the other statecharts in the collection). A step has to obey several constraints [33]:

**Enabledness.** All edges in the step must be enabled.

**Consistency.** All edges in the step must be pairwise consistent. This means that they are either identical or that their scopes are different children of some AND-node or their descendants (in the latter case, the scopes are called orthogonal in [33]).

**Priority.** We assume a given priority scheme (a partial order on the edges) that resolves some of the inconsistencies: If an enabled edge $e$ is not in the step, then there must be an edge in the step that is not consistent with $e$ and does not have lower priority than $e$.

**Maximality.** A step must be maximal. This means that adding any edge leads to a violation of the above conditions.\(^3\)

Eshuis and Wieringa [33] have given the following algorithm $\text{nextstep}$ that constructs a step of a single statechart which satisfies the conditions above in a straightforward manner. (Note that this algorithm differs from Harel’s [43] algorithm $\text{nextstep}$, which constructs and executes a complete superstep.)

**Algorithm 2.3.1 Nextstep [33].**

Given a state $(C_i, I_i)$, we construct a set $T$ which is a possible step by:

1. Let $T := \emptyset$.

2. While there is an enabled edge in $\text{En}(C_i, I_i) \setminus T$ that is consistent with $T$:
   
   (a) Choose an edge $t \in \text{En}(C_i, I_i) \setminus T$ such that $t$ is consistent with $T$ and $t$ has maximal priority.

   (b) Let $T := T \cup \{t\}$.

3. $T$ is a possible step.

We define $\text{Steps}(\text{En}(C_i, I_i))$ to be the set of possible steps in the state $(C_i, I_i)$.

**Example 2.3.1** A possible state of the damage assessor model from example 2.2.1 is: 
$\langle \{\text{root, Assessing, Phone assessment}\}, \{\text{assessed}\} \rangle$. In this state, two edges are enabled: (a) one labelled assessed / repair and (b) one labelled assessed / write off. The two edges are inconsistent, as both have scope $\text{root}$. So, there are two possible steps: $\{(a)\}$ and $\{(b)\}$.

---

\(^3\)If there is no unique maximum, the system is nondeterministic.
2.3.2 Step Execution

After having settled how steps are selected within a single statechart, we now consider their joint execution in the collection \( \{SC_1, \ldots, SC_n\} \). On the level of a single statechart, executing a step consists of two parts: updating the events occurring in the actions and determining the new state. As the actions of one statechart may influence the sets of events of other statecharts, we describe the step execution of the complete collection of statecharts.

The default completion \( C' \) of some set of nodes \( C \) is the canonical superset of \( C \) such that \( C' \) is a configuration. If \( C' \) contains an OR-node \( x \) but \( C \) contains none of its descendants, \( C' \) contains its default node \( \text{default}_i(x) \).

Given configurations \((C_1, \ldots, C_n)\) and steps \((T_1, \ldots, T_n)\) we define for statechart \( SC_i \) the new state \((C'_i, I'_i)\) by:

- \( C'_i \) is the default completion of
  \[
  \left( \bigcup_{(X,e,g,A,Y) \in T_i} Y \right) \cup \{x \in C_i | \forall (X,e,g,A,Y) \in T_i: x \text{ is not a descendant of scope}(X,Y)\}.
  \]

  The first set consists of all target nodes that are entered; the second set contains all nodes that are left unchanged.

- \( I'_i = \bigcup_{k=1}^n \{e | \exists (X,e,g,A,Y) \in T_k: \text{send } i.e \in A\} \)

We denote this as: \( \text{Execute}(C_1, \ldots, C_n, T_1, \ldots, T_n) = ((C'_1, I'_1), \ldots, (C'_n, I'_n)) \).

Example 2.3.2 Above, we mentioned two possible steps in a specific state of example 2.2.1. If we take step \( \{a\} \), the new configuration is the default completion of \{root, Repairing\}. As Repairing is an AND node, both Invoice handling and Reporting are part of the default completion. These two are OR nodes, so their default nodes (Waiting for invoice and Writing report, respectively) are part of the default completion. The new configuration, then, is: \{root, Repairing, Invoice handling, Waiting for invoice, Reporting, Writing report\}.

Also, external component 2 receives the event repair.

2.3.3 Input–Output Automaton Semantics

The semantics of a collection of statecharts is an input–output automaton.

Assume given a finite collection of statecharts \( SC_1, \ldots, SC_n \). The input–output-automaton model of the collection of statecharts is \( (S, A, \Delta, AP, I, s_0) \), where:

- \( S = \times_{i=1}^n (\text{Conf}_i \times \text{P} (\text{Events}_i)) \)
• The set of actions \( \mathcal{A} \) is defined by:

\[
\mathcal{A}^\text{in} = P \left( \bigcup_{i=1}^{n} \text{Events}_i \right) \setminus \{\emptyset\}
\]

\[
\mathcal{A}^\text{out} = P \left( \bigcup_{i=1}^{n} \{j.e \mid j \notin \{1, \ldots, n\} \land \exists (X,e,g,A,Y) \in \text{Edges}_i : \text{send } j.e \in A \} \right)
\]

All events may be received from the environment; but only events \( j.e \) that are not internal (not directed to a statechart \( SC_j, j \in \{1, \ldots, n\} \)) are output.

• Input transitions are defined by the rule:

\[
E \in \mathcal{A}^\text{in} \quad s_i = (C_i, I_i) \quad s'_i = (C_i, I_i \cup (E \cap \text{Events}_i)), \text{ for all } i = 1, \ldots, n
\]

\[
(s_1, \ldots, s_n) \xrightarrow{E} (s'_1, \ldots, s'_n)
\]

Input transitions only store the events received in the respective statechart’s set of input events.

• Output transitions are defined by the rule:

\[
s_i = (C_i, I_i) \quad T_i \in \text{Steps}(\text{En}(s_i)), \text{ for all } i = 1, \ldots, n
\]

\[
(s_1, \ldots, s_n) \xrightarrow{E} \text{Execute}(C_{1..n}, T_{1..n})
\]

where \( E = \bigcup_{i=1}^{n} \{j.e \mid j \notin \{1, \ldots, n\} \land \exists (X,e,g,A,Y) \in T_i : \text{send } j.e \in A \} \). Output transitions actually construct a step and execute it.

• The set of atomic propositions is \( \mathcal{AP} = \bigcup_{i=1}^{n} \{i.\text{isin}(x) \mid x \in \text{Nodes}_i\} \).

• The interpretation of atomic propositions is defined by the rule:

\[
x \in C_{i..} \text{ for some } i \in \{1, \ldots, n\}
\]

\[
i.\text{isin}(x) \in \mathcal{I}((C_1, I_1), \ldots, (C_n, I_n))
\]

• The initial state is \( s_0 = ((C_{0,1}, \emptyset), \ldots, (C_{0,n}, \emptyset)) \), where \( C_{0,i} \) is the default completion of \( \{\text{root}_i\} \).

Note that the communication mechanism between statecharts differs from the communication between IOA: all statecharts may take their step and produce output at once, while in a IOA, only one of the components is allowed to take an output transition.

**Example 2.3.3** A large fragment of the input–output automaton that corresponds to example 2.2.1 is shown in figure 2.2. Each state is drawn as a box with the configuration and the set of input events inscribed. In the initial state, all input transitions for the three events assessed, report finished and start are shown. In the other states, (almost) only relevant input transitions are drawn to avoid cluttering of the diagram.
Figure 2.2: Semantics of the damage assessor workflow (fragment)
2.4 Extensions of the above Statecharts

I have tried to present the statecharts as simply as possible. Therefore, I have omitted two extensions that are often used: local variables and real time.

2.4.1 Local Variables

It is possible to add a set of local variables to a statechart. We extend guards and actions to check and set the local variables:

- In addition to guards of the form $j.isin(x)$ (for $j \in \{1, \ldots, n\}$ and $x \in \text{Nodes}_j$), we allow comparisons like $expr \leq expr$ and $expr \geq expr$, for arithmetic expressions made up from the variables and constants.

- In addition to actions of the form $send j.e$ (for $j \in \mathbb{N}$ and $e \in \text{Events}_j$), we allow assignments like $v := expr$, where $v \in \text{Vars}_i$.

A state, then, consists of a triple: configuration, set of input events, and a valuation of the local variables. Edges are only enabled when the guards on local variables hold. When the statechart takes a step, the assignments may change the values of the local variables.

With this extension, it is possible to describe infinite-state systems using a statechart with a variable with infinite range. Therefore, we will sometimes only allow bounded integer variables.

Inconsistent assignments. It may happen that several edges with inconsistent assignments are enabled at once, or even that the engineer has added several inconsistent assignments to a single edge. In that case, at least part of the assignments must be ignored.

There are several solutions to this problem. For example, Eshuis and Wieringa [33] then choose one of the assignments at random and ignore the others. Others declare that a set of edges is only consistent if all assignments are consistent, so, there are fewer edges in a step.

As this problem is orthogonal to the extensions we describe in the following chapters, we have decided not to solve this problem in detail. Any existing solution can be combined with our extensions.

2.4.2 Real Time

Real-time statecharts can quantify the time that passes between some events. To this end, they include two pseudo-events, which can be used as edge labels instead of events:

- after($delay$) is enabled $delay$ time units after its source state was entered. If the source state is left and reentered, the delay restarts.

- when($time$) is enabled at time $time$.

A real-time statechart can be given a semantics in form of a timed input–output automaton. Timed input–output automata will be defined in section 3.5.1. Each time a state with an after($delay$) edge is entered, a clock is set; when it reaches $delay$, the edge gets enabled. For when($time$) edges, we assume one central clock.
2.5 Unified Modelling Language

The Unified Modelling Language (UML) is a set of nine languages to describe several aspects of a software system in an object-oriented way. In the beginning of object-oriented software engineering, there were a multitude of methods, each with its own language that competed with the others. To reduce the differences to the more essential points, the UML was introduced. Originally, it was a proposal by the “three amigos” Booch, Jacobson, and Rumbaugh [12, 54, 89], but now, the Object Management Group (a consortium of “virtually every large company in the computer industry”, according to its website) is continuing the development of the language.

Diffusion/Dissemination of the UML. The UML has been successfully introduced into the market of professionally used software specification languages. So, we can safely assume that the languages in the UML are well-known to professional software engineers.

How others use the UML. The UML originally served to describe (the implementation of) an object-oriented software system.

Typically, not the full UML is used; there is some redundancy. For example, collaboration diagrams essentially present the same information as sequence diagrams, so most users only draw one of the two diagrams. Also, the UML does not come linked to a specific software development method. As several methods lay different accents, they also may choose to only use parts of the UML. Finally, not for every software development project, all aspects that can be modelled by the UML are relevant.

How we use the UML. We use parts of the UML for requirements engineering, where one describes the essential architecture of a (software) system. We mainly use three languages from the UML and reinterpret them for requirements engineering:

Class and object diagrams describe the structure of a system. They show its decomposition into objects by describing the classes that objects may possibly have, and the absolute and relative cardinalities of the objects.

Statecharts describe the behaviour of an object. All classes defined in the class diagram have an associated statechart; each object behaves according to the statechart associated to its class.

Communication and collaboration diagrams describe the possible communication between objects of a system. While a collaboration diagram exemplifies how objects communicate in a single behaviour, the communication diagram (a variant of the collaboration diagram defined by Wieringa [102]) generalises this to all possible communication.

In the following chapters, we will give a system overview by a class diagram, object diagram, or communication diagram. For the relevant classes, we will provide a statechart that describes the behaviour.

Depending on the analysis method we use, we may restrict the number of objects to a finite number, a bounded number, or even to a fixed number.
Semantics. The exact semantics of the languages in the UML is still under discussion. The UML does not have a fully defined formal semantics; some points are intentionally left open, but more often, subtle issues are not clarified [87]. Many of them arise from the interaction of features. For example, the UML does not specify in what order events received by an object are processed; but then, one should make the fairness assumption that an event cannot be delayed indefinitely.

Extensions. The UML includes a mechanism for extension by so-called profiles. The intention of a profile is to add structured annotations to UML diagrams that extend the semantics of a UML model without changing the plain semantics, so that a UML diagram with the annotations can also be analysed by tools that only know the plain semantics. The structured annotations contain information that could not be expressed in the plain diagram. A profile changes the semantics of some UML constructs; so, in our opinion, it is a strict extension. More on the UML profile for schedulability, performance, and time specification [99] can be found in the related work section of chapter 5.

2.6 TCM

The Toolkit for Conceptual Modelling is a collection of editors to draw, among others, UML diagrams. It is intended to become a simple CASE tool that allows the analyst to simulate and verify the UML model or to generate code from it.

TCM can be combined with model checking as follows: TCM provides the analyst with a simple and intuitive interface, for example to draw statecharts. The analyst then enters a desired property (for example in ATCTL, see chapter 3) and instructs TCM to start up the most suitable model checker.

Eshuis has implemented an interface to the model checkers SMV and nuSMV [32] that also presents the model checking result in the TCM diagram again. Cañete Valdeón at LSI, Universidad de Sevilla is currently working on an extension of Eshuis' work (not yet published). We implemented an interface to the model checker Kronos [14, 108].

TCM also was used to draw most images in this dissertation.

2.7 Related Work

2.7.1 Statecharts

The original definition of statecharts stems from Harel [43]. He defined them as an extension of state machines that is relevant to the specification and design of complex discrete-event systems. The article does not give formal semantics in full detail, but it seems that there is a precise idea behind it.

There are several papers describing statechart semantics. Harel et al. [42] described the original semantics; the fact that it was not compositional led many others to propose improvements. Pnueli and Shalev [85] gave a semantics which is equivalent under some conditions. The Statemate semantics is described by Harel and Naamad [46]. Damm et al. [21] defined a formal, compositional statechart semantics, which served as a basis for the semantics of Eshuis and Wieringa [33]. Hoorman, Ramesh and de Roever [51] describe an
axiomatization of statecharts based on a textual representation of statecharts by Huizing, Gerth and de Roever [52]. Leveson et al. [72, 71] describe a variant of statecharts called RSML used for requirements engineering. RSML contains several features not found in standard statecharts, for example, directed communication. An overview of how these semantics differ can be found in von der Beeck [8]. Another comparison of how several statechart variants define steps has been given by John [61].

Our semantics is similar to the abovementioned in the following points:

- We define a requirements-level semantics that abstracts from some imperfections of currently existing computer systems: it assumes “perfect technology”, i.e., unlimited speed and memory. All of the abovementioned semantics assume perfect technology. The assumption simplifies the semantics, as one does not need to include provisions for possible delays between the reception of an event and the reaction to it.

- We define a sequential step semantics, that reacts to internal events generated by one step in the following step. This is in accordance with Damm et al. [21], but differs from Pnueli and Shalev [85]. Sequential step avoids the problem that the effect of a transition may be contradictory to its cause (problem 4 of von der Beeck [8]), as the effect is delayed until the next step.

- Our semantics is a superstep semantics, that does not wait for extra input from the environment before reacting to internally generated events. (This is opposed to single-step behaviour, where the system always waits some time after having executed a single step; in that time, the environment may send extra inputs to the system.) Many semantics let the user choose between superstep and single-step behaviour.

Latella, Majzik and Massink [69] define a statechart semantics by a translation to hierarchical automata. These hierarchical automata can be input to a model checker [38, 70]. There is also a stochastic extension of statecharts and their hierarchical automata semantics [39]. It is similar to our work in the sense that it identifies a subset of essential UML statechart features and translates them.

David, Möller, and Yi [25] have defined an extension of UML statecharts with real-time. They use as semantic model hierarchical timed automata, which are similar to the hierarchical automata mentioned above. This work is linked to the real-time model checker Uppaal [68].

Statemate. The case tool Statemate embeds the use of statecharts in additional system descriptions that describe the conceptual decomposition into activities (“activity charts”) and the physical decomposition (“module charts”).

An activity chart is somewhat similar to an object or class diagram, that also shows the conceptual decomposition. However, all activities in the activity chart are coordinated by a central statechart that receives input from the activities or the system environment and sends commands to them. We use statecharts in a decentralised way: every class in the

4 Unfortunately, the presentation of the semantics defined by Eshuis and Wieringa [33] is incorrect.
2.7. Related Work

A class diagram may have its own statechart. The behaviour of the system is defined by the composition of all these statecharts.

As this thesis is not concerned with the (physical) implementation of a system, we do not define a structure that corresponds to module charts.

2.7.2 UML

Ten years after Harel’s first article, he and Gery [44] redefined statecharts to be used in the context of the upcoming UML. They use a statechart to describe the behaviour of an object in an implementation, dropping the perfect technology assumption. The method they describe is implemented in the case tool Rhapsody. Rhapsody uses an object model diagram, a combination of object diagram and communication diagram, to describe the communication relations between objects.

The OMG has continued this work and defined an informal UML statechart semantics [83].

Our semantics is similar to the UML semantics in the following points:

- We define an event-driven semantics, i.e., the system starts its reaction immediately after reception of an event, without waiting for a clock tick.

- We define an object-oriented semantics, where each object in the system with non-trivial behaviour has its own statechart. So, the behaviour of the complete system is the composition of the single statecharts’ behaviours. (In Statemate, as a contrast, the behaviour is centralised in one large statechart.)

The main difference between our work and the UML semantics is: we define a requirements-level semantics, because we use the UML to describe a requirements model of a system. This is in accordance with the event-driven, object-oriented statechart semantics of Eshuis and Wieringa [33].

Wieringa and Broersen [103] have described a small subset of the UML called lightweight UML (LUML) and defined a semantics for LUML models in a version of order-sorted dynamic logic. The main elements of LUML are lightweight class diagrams and state–transition diagrams; the latter can be understood as very simple statecharts. This choice of languages out of the UML is similar to our choice, but Wieringa and Broersen have reduced the expressiveness of the language considerably: state–transition diagrams contain neither hierarchy nor parallelism.
Chapter 3

Real-Time and Action Language: ATCTL

3.1 Introduction

This chapter is an updated version of [60], which is based on the conference proceedings [59].

Specifying desired properties of statecharts formally is not so easy, as the logics currently in use are not adapted to the rich language of statecharts. A suitable logic should include means to express properties of states and of actions. For real-time statecharts, it should also be able to express real-time requirements.

Most current logics allow for the expression of state and action properties only in a half-hearted way: some choose for state properties as the basic notion and require to circumscribe action properties. For example, the property “action $a$ happens” can only be expressed as “the effects of action $a$ on the current state occur”. Other logics choose for actions as the basic notion and require to write “it is possible to execute the action $x$” instead of “the system is in state $x$”.

We have designed a combination of logics, called ATCTL, that provides all these means to the engineer. It is based on computation tree logic (CTL), a logic that is widely used for model checking, as it allows for rather efficient model checkers.

ATCTL combines two known extensions of CTL, namely action CTL (ACTL) and timed CTL (TCTL). So, ATCTL can express properties of actions and real-time properties. Properties of states can be expressed by the standard propositional basis of CTL (which was abandoned in the original definition of ACTL).

ATCTL’s semantic structures are Timed Output Automata (TOA), a simple variant of timed input–output automata, suited for closed systems. A real-time statechart model can be simplified to a TOA; therefore, we can use ATCTL to express desired properties of (the semantic model of) statecharts.

We show that ATCTL can be reduced to ACTL as well as to TCTL, and therefore also to CTL (with a small incompleteness in the reduction to ACTL); see figure 3.1. This gives us a choice of tools for model checking. SMV [17] is a CTL model checker. ACTL as well as TCTL can be checked by a reduction to CTL. In addition, specific TCTL and ACTL model checkers have been written: Yovine [108, 14] has implemented Kronos for TCTL.
3.2 Computation Tree Logic

All logics presented in this chapter are extensions of CTL (Computation Tree Logic). Therefore, we first give a short definition of the syntax and semantics of CTL.

The logic CTL was first proposed by Emerson and Clarke [30] to describe requirements on synchronisation between modules in a specification. CTL can express properties of output automata, a simplification of input–output automata. Satisfiability of CTL is decidable (in exponential time) [31]. CTL models can be checked against a CTL formula in time linear in the size of the automaton and the formula [19]. So, the logic is suitable for automatic verification.

Syntax. Given a set of atomic propositions \( AP = \{p, q, \ldots\} \), we define the syntax of CTL as:

\[
\varphi, \psi ::= \bot \mid p \mid \neg \varphi \mid \varphi \land \psi \mid \exists X \varphi \mid \forall X \varphi \mid \exists (\varphi U \psi) \mid \forall (\varphi U \psi)
\]

The non-modal formulas have their usual meaning. \( \exists X \varphi \) states that there is some step from the current state to a state where \( \varphi \) holds. \( \forall X \varphi \) states that every step from the current state leads to a state where \( \varphi \) holds.

\( \exists (\varphi U \psi) \) holds in a state where there is at least one run which starts with zero or more states satisfying \( \varphi \) and then has a state satisfying \( \psi \). \( \forall (\varphi U \psi) \) holds in a state where each complete run has this form.

Abbreviations. The disjunction, implication, and other modal operators are defined as usual:

\[
\begin{align*}
\varphi \lor \psi & ::= \neg (\neg \varphi \land \neg \psi) \\
\exists \lozenge \varphi & ::= \exists (\top U \varphi) \\
\forall \lozenge \varphi & ::= \forall (\top U \varphi) \\
\lozenge \varphi & ::= \varphi \lor \neg \varphi \\
\exists \square \varphi & ::= \exists \neg \lozenge \neg \varphi \\
\forall \square \varphi & ::= \neg \exists \lozenge \neg \varphi
\end{align*}
\]
3.2. Computation Tree Logic

3.2.1 Semantics

We interpret CTL formulas over an output automaton $K = (S, A, \Delta, AP, I, s_0)$. Recall that an output automaton is an input–output automaton where $A^\text{in} = \emptyset$.

A state $s$ of $K$ satisfies formula $\varphi$ (noted $K, s \models \varphi$) if:

- $K, s \models \bot$ never.
- $K, s \models p$ if $p \in I(s)$.
- $K, s \models \neg \varphi$ if not $K, s \models \varphi$.
- $K, s \models \varphi \land \psi$ if $K, s \models \varphi$ and $K, s \models \psi$.
- $K, s \models \exists X \varphi$ iff there are $a \in A$ and $s' \in S$ such that $(s, a, s') \in \Delta$ and $K, s' \models \varphi$.
- $K, s \models \forall X \varphi$ if all complete runs starting in $s$ have the form $s \xrightarrow{a_1} s' \xrightarrow{a_2} \cdots$ and satisfy $K, s' \models \varphi$.
- $K, s \models (\varphi \mathrel{U} \psi)$ if there is a run $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \cdots$ of $K$ that starts in $s = s_1$ and satisfies the following condition:

**The until condition.** There is an $i \geq 1$ such that

- $K, s_j \models \varphi$, for all $j < i$.
- $K, s_i \models \psi$.
- $K, s \models (\varphi \mathrel{U} \psi)$ if every complete run starting in $s$ satisfies the above until condition.

We say that an output automaton satisfies a formula, $K \models \varphi$, if its initial state satisfies the formula: $K, s_0 \models \varphi$.

**Unlabelled output automata.** In the above satisfaction relation, the action labelling does not influence the truth value of any formula. We could as well dispose of the labels and define the following, somewhat simpler, structure: An unlabelled output automaton has a singleton action set, denoted $A = A^\text{out} = \{\ast\}$. In that case, we sometimes omit the set $A$ at all: $K = (S, \Delta, AP, I, s_0)$.

---

1One could also give an alternative definition: “... iff there is a complete run starting in $s$ that has the form $s \xrightarrow{a_1} s' \xrightarrow{a_2} \cdots$ and $K, s' \models \varphi$.”

The two definitions are equivalent: the original definition implies the alternative because we can construct a complete run that starts with the indicated transition $(s, a, s') \in \Delta$. On the other side, the alternative definition implies the original one because every complete run starts with a transition, in this case $(s, a_1, s') \in \Delta$.

The same argument can, of course, be applied to $\exists \mathrel{U}$ formulas.
3.2.2 CTL Model Checking

Clarke, Emerson and Sistla [19] describe a model checker algorithm. This algorithm takes as input a total\(^2\) unlabelled output automaton and a CTL formula and outputs in which states of the output automaton the formula holds. The algorithm is both linear in the length of the formula and the size of the output automaton (i.e., the number of states + the number of transitions).

The main idea of the algorithm is: Given an output automaton \(K = (S, \Delta, AP, I, s_0)\) and a CTL formula \(\varphi\), we proceed in phases: the first phase processes all subformulas of \(\varphi\) of length 1, the second phase all subformulas of length 2, and so on. In phase \(i\), we assign labels to each state telling which subformulas of length \(i\) hold in that state. After the last phase, the states where \(\varphi\) holds are labelled.

For example, to assign the correct labels for \(\exists(\varphi \cup \psi)\), we first label all states where \(\psi\) holds (i.e., all states labelled with \(\psi\)) and then extend the labelling recursively to their \(\Delta\)-predecessors that satisfy \(\varphi\). – To assign the correct labels for \(\forall(\varphi \cup \psi)\), we first label all states where \(\psi\) holds and then extend the labelling recursively to all states that satisfy \(\varphi\), have \(\Delta\)-successors, and of which all \(\Delta\)-successors are labelled.

Clarke et al. also introduce an extension of the model checker algorithm that includes fairness.

Although this algorithm is (in principle) linear in the size of the output automaton, it is still too slow for larger systems. There, many optimisations are used, for example symbolic model checking [16].

3.3 System Model

We base our system models on input–output automata with some extensions:

- Real-time aspects come in from timed automata [1]. Basically, the time how long a system is in a certain state is quantified. To this end, timed automata include clocks that may be reset when a transition is taken. Clock constraints express that some transitions are only enabled during some specific time interval.

- The transitions are labelled similar to input–output automata or labelled transition systems (LTSs).

- We use propositional logic as a model for data. In traditional LTSs, there is no possibility to distinguish states that allow the same behaviours.

- Action logic is similar to dynamic logic [45, 15]. It serves to express constraints on actions.

3.4 Action Logic

Action logic serves to specify constraints on sets of actions. Its atomic formulas are action symbols. Assume given a finite set of action symbols \(A = \{a, b, \ldots\}\). Action terms \(a, b, \ldots\)

---

\(^2\)An output automaton is total if in every state, at least one (output) transition is enabled.
3.5. Real Time Model

are then defined by the syntax:
\[ \alpha, \beta ::= \text{any} \mid a \mid -\alpha \mid \alpha + \beta \]

For the sake of simplicity, we identify action symbols with their interpretation, actions. A set of actions \( A, B, \ldots \in \mathcal{P}(A) \) is said to satisfy an action term, \( A \models \alpha \), in the following cases:

- \( A \models \text{any} \).
- \( A \models a \) iff \( a \in A \).
- \( A \models -\alpha \) iff \( A \not\models \alpha \).
- \( A \models \alpha + \beta \) iff \( A \models \alpha \) or \( A \models \beta \) (or both).

Speaking informally, any set is allowed by \textit{any}. The term \textit{a} requires a set containing that action; \( \textit{aaa} \) requires a set that is forbidden by \textit{aaa}; and \( \textit{aaa} + \textit{bbb} \) requires a set satisfying (at least) one of the constraints. We define the abbreviation: \( \textit{aaa} \& \textit{bbb} ::= \textit{aaa} + \textit{bbb} \).

The definition looks quite similar to a Boolean algebra, and in fact, if one defines \([\alpha] := \{ A \subseteq A \mid A \models \alpha \} \), the sets \([\alpha] \) become elements of the Boolean algebra \( \mathcal{P}(\mathcal{P}(A)) \) with set union, intersection and complement as its operations.\(^3\)

### 3.5 Real Time Model

As we have discussed in section 1.2, several choices need to be made when modelling the real-time aspect of a system. We have chosen a continuous, branching time model. So, we model time by the non-negative real numbers.

We use \textit{clocks} to measure time. Clocks run forward and may be reset to zero at any moment. All clocks are assumed to run exactly at the same speed. We assume a fixed finite set of clock symbols \( C = \{ c, c', \ldots \} \).

Clocks may enable or disable state changes in a system. To describe this, we use \textit{clock constraints}. Clock constraints \( \xi, \eta \in CC \) are given by the syntax: (for any \( n \in \mathbb{N} \))

\[ \xi, \eta ::= c \leq n \mid c \geq n \mid -\xi \mid \xi \land \eta \]

Other operators are defined as the usual abbreviations, e.g. \( n < c := -c \leq n \).

A \textit{time stamp} is a valuation of the clock symbols: \( \chi : C \rightarrow \mathbb{R}^+_0 \). A time stamp can easily be extended to expressions over clock symbols and to clock constraints; we define that \( \chi(\xi) \) is satisfied whenever the valuation \( \chi \) makes the clock constraint \( \xi \) true.

Later on, we will also use \textit{partial time stamps}, which are partial functions \( \nu : C \rightarrow \mathbb{R}^+_0 \).

We define two abbreviations, \( \chi[C := 0] \) (where \( C \) is a set of clock symbols) and \( \chi + \delta \) (where \( \delta \in \mathbb{R}^+_0 \)), by:

\[
(\chi[C := 0])(c) := \begin{cases} 0 & \text{if } c \in C \\ \chi(c) & \text{otherwise} \end{cases}
\]

\[
(\chi + \delta)(c) := \chi(c) + \delta
\]

\(^3\)If \( A \) were infinite – which we have forbidden – this language’s standard model were the Boolean algebra \( \mathcal{P}_{\mathcal{C}}(\mathcal{P}(A)) \), where \( \mathcal{P}_{\mathcal{C}}(S) = \{ T \subseteq S \mid T \text{ is finite or } S \setminus T \text{ is finite} \} \) denotes all finite and cofinite subsets of \( S \).
3.5.1 Timed Input–Output Automata

We extend the input–output automata model defined in section 2.1 by real-time to timed input–output automata.

A timed input–output automaton (TIOA) is a septuple \((L, C, A, \Delta, AP, \mathcal{J}, l_0)\) where:

- \(L\) is a finite, non-empty set of locations. For each location \(l \in L\), there is a location invariant \(i_l \in CC\). (The location invariant serves to enforce some transitions: the system cannot stay in the location too long, but it will take action before some deadline.)
- \(C\) is a finite set of clock symbols.
- \(A\) is a set of labels, partitioned into input and output labels \(A_{\text{in}}\) and \(A_{\text{out}}\).
- \(\Delta \subseteq L \times A \times P(C) \times L\) is the transition relation. For each transition \((l, a, C, l')\), there is a transition invariant \(i_{(l, a, C, l')} \in CC\). We sometimes write \(l \xrightarrow{a, C} l'\) instead of \((l, a, C, l') \in T\) and \(\xi = i_{(l, a, C, l')}\).
- \(AP\) is a set of atomic propositions.
- \(\mathcal{J} : L \rightarrow P(AP)\) is the interpretation of the atomic propositions.
- \(l_0 \in L\) is the initial location.

**States, steps, runs.** A state of a TIOA is a pair \((l, \chi)\) that consists of a location \(l \in L\) and the timing information, given by a time stamp \(\chi\).

A step describes a possible state change. There are two kinds of steps:

**Time steps** relate a state with a similar state in the future. They are described using a positive delay \(\delta\):

\[
\begin{align*}
\chi' &= \chi + \delta \\
(l, \chi) &\xrightarrow{\delta} (l, \chi')
\end{align*}
\]

for some \(\delta \in \mathbb{R}^+\)

**Action steps** relate a state via a transition to another state:

\[
\begin{align*}
l \xrightarrow{a, C} l' &\quad \chi = \chi[C := 0] \\
(l, \chi) &\xrightarrow{a} (l', \chi')
\end{align*}
\]

If we write about steps in general, we may write \((l, \chi) \xrightarrow{a, C} (l', \chi')\) where either \(\delta^- \in \mathbb{R}^+\) and \(a^T = T\) (the \(T\) indicates the Time step), or \(\delta^- = -\) and \(a^T \in A\). (the \(-\) indicates that an action step produces no delay, so the numeric value of \(-\) is assumed to be 0).

A run of a TIOA is a sequence of steps, where the target of each step is the source of the next one. We write this as:

\((l_1, \chi_1) \xrightarrow{a_1^T, \delta_1^-} (l_2, \chi_2) \xrightarrow{a_2^T, \delta_2^-} (l_3, \chi_3) \rightarrow \cdots\)
It is possible to take several time steps without doing anything in between, or to take multiple action steps without waiting in between. This allows us to represent, say, a superstep in the semantics of a real-time statechart as a sequence of steps in a run.

A complete run is a run that cannot be continued, either because it is infinite or because there is no step starting in its final state.

To identify a certain position in a run, we use the format \((i, d)\), where \(i \in \{1, 2, 3, \ldots\}\) is the index and \(0 \leq d \leq d_{i+1}\) is the local delay. At position \((i, d)\), the system is in state \((l_i, c_{i+d})\). The time consumed up to position \((i, d)\) is:

\[
T_i(i, d) := d + \sum_{j=1}^{i-1} \delta_j
\]

We define that position \((j, \varepsilon)\) lies before position \((i, d)\) if either \(j < i\) or \((j = i \land \varepsilon < \delta)\).

**Timed output automata.** As with standard input–output automata, we define a simplification: a timed output automaton (TOA) is a TIOA that has no input actions: \(A^\text{in} = \emptyset\).

In this chapter, we will use (timed) output automata with a special set of labels \(A\): we assume that it is the powerset of a set of action symbols \(A\), so \(A = A^\text{out} = \mathcal{P}(A)\). This enables us to express properties of steps using the action logic defined above. (Note that we use the term “labels” where we have used the term “actions” in section 2.1 to avoid confusion with action symbols.)

Similar to the unlabelled output automata defined in section 3.2.1, we also define unlabelled timed output automata with a singleton set of labels: \(A = A^\text{out} = \{\ast\}\), or equivalently, let the set of action symbols be \(A = \emptyset\).

### 3.6 Action-Based Timed CTL

We now present a logic which may be used to specify properties of TOAs.

**Syntax.** Assume given a set of proposition symbols \(AP = \{p, q, \ldots\}\), a set of action symbols \(A = \{a, b, \ldots\}\) and a set of clock symbols: \(C = \{c, c', \ldots\}\). (These three sets correspond to the sets of atomic propositions, the set of action symbols and the set of clocks in the TOA.) The syntax of ATCTL is then defined by: (for \(n \in \mathbb{N}\))

\[
\varphi, \psi ::= \bot \mid p \mid \neg \varphi \mid \varphi \land \psi \mid c \leq n \mid c \geq n \mid \text{reset } c \text{ in } \varphi \mid \\
\exists X_a \varphi \mid \forall X_a \varphi \mid \exists (\varphi \cup \psi) \mid \forall (\varphi \cup \psi) \mid \exists (\varphi \cup_f \psi) \mid \forall (\varphi \cup_f \psi)
\]

The common formulas have their usual meaning. \(c \leq n\) and \(c \geq n\) hold in a state where the time stamp says that clock \(c\) is in the indicated range. reset \(c\) in \(\varphi\) is a sort of hypothetical reasoning: “If \(c\) were zero (ceteris paribus), then \(\varphi\) would hold.” \(\exists X_a \varphi\) states that there is some action step from this state satisfying the constraints \(a\), such that \(\varphi\) holds afterwards. \(\forall X_a \varphi\) states that every step from this state is an action step satisfying the

\[\text{footnote text}\]

\[\text{footnote text}^\text{4}\]
constraint $a$ and $\varphi$ will hold afterwards. $\exists (\varphi \cdot a U \psi)$ is valid in a state where it is possible to do zero or more $a$ steps (during which $\varphi$ holds), and after these steps, $\psi$ becomes true. $\exists (\varphi \cdot a U_b \psi)$ is valid in a state where it is possible to do zero or more $a$ steps (during which $\varphi$ holds), and finally do one $b$ step, such that $\psi$ is valid immediately afterwards.

We include two different “until” operators to get full expressivity. The operators $\forall a U$ and $\forall a U_b$ cannot be reduced to each other.$^5$

Similarly, the two formulas $\neg \exists X a \top$ and $\forall X a \bot$ are not equivalent: the first requires that action $a$ is impossible in the current state, and the second requires that no action can be taken in the current state. Therefore, it is not possible to reduce $\forall X$ formulas simply to $\exists X$ formulas. A more involved translation is still possible: $\forall X_a \varphi$ is equivalent to $\neg \exists X_{\neg a} \top \land \neg \exists X_a \neg \varphi$.

**Abbreviations.** The disjunction $\lor$ and implication $\rightarrow$ are defined as usual. $\top := \neg \bot$. The paper by Alur et al. [2] defines combined operators; we may introduce them as abbreviation: $\exists (\varphi \cdot a U \leq n \psi) := \text{reset } c_{\text{new}} \in \exists (\varphi \cdot a U (c_{\text{new}} \leq n \land \psi))$, where $c_{\text{new}}$ denotes a new clock symbol. We can define similar abbreviations for $\geq, = \text{ etc.}$ In all our case studies, the combined operator could be used; we have chosen our definition because it separates real-time and actions more clearly. The modal operators $\Box, \Diamond$ are defined as usual, e.g.: $\exists a \Diamond_b \varphi := \exists (\top \cdot a U_b \varphi)$. Finally, the two abbreviations can be combined to formulas like $\exists a \Diamond_b \leq^n \varphi := \text{reset } c_{\text{new}} \in \exists (\top \cdot a U_b (c_{\text{new}} \leq n \land \varphi))$.

**Semantics.** Assume given a TOA $M = (L, C, A, \Delta, AP, J, I_0)$. We will interpret a formula in a state $(l, \chi)$ of the model together with a partial time stamp $v : C \rightarrow R_0$, which serves to interpret subformulas of a reset $c$ in $\cdot$ operator.$^6$

The satisfaction relation $(M, l, \chi) \models_v \varphi$ is defined by:

- $(M, l, \chi) \models_v \bot$ never.
- $(M, l, \chi) \models_v p$ iff $p \in J(l)$.
- $(M, l, \chi) \models_v \neg \varphi$ iff $(M, l, \chi) \not\models_v \varphi$.
- $(M, l, \chi) \models_v \varphi \land \psi$ iff $(M, l, \chi) \models_v \varphi$ and $(M, l, \chi) \models_v \psi$.
- $(M, l, \chi) \models_v c \leq n$ iff $f(c) \leq n$, where $f(c) = \begin{cases} v(c) & \text{if } v(c) \text{ is defined} \\ \chi(c) & \text{otherwise} \end{cases}$.
- $(M, l, \chi) \models_v c \geq n$ iff $f(c) \geq n$, where $f(c)$ is defined as above.
- $(M, l, \chi) \models_v \text{reset } c$ in $\varphi$ iff $(M, l, \chi) \models_{v[\{c\}:=0]} \varphi$.

---

$^5$The operator $\exists a U_b$ can be defined as an abbreviation: $\exists (\varphi \cdot a U_b \psi) \equiv \exists (\varphi \cdot a U \exists b \varphi)$.

$^6$Henzinger et al. [48] call $v$ a clock environment.

This interpretation, involving two time stamps, has the effect that it is no more necessary to define formula clocks (Formula clocks are clocks which cannot be reset by the model, used by some authors to express real-time constraints.).
3.6. Action-Based Timed CTL

- \((M, l, \chi) \models \exists X_a \varphi\) iff there is a run\(^7\) \((l, \chi) \xrightarrow{A} (l', \chi') \cdots\) which satisfies \(A \models \alpha\) and \((M, l', \chi') \models \varphi\).

- \((M, l, \chi) \models \forall X_a \varphi\) iff every complete run starting in \((l, \chi)\) has the form \((l, \chi) \xrightarrow{A} (l', \chi') \cdots\) and satisfies \(A \models \alpha\) and \((M, l', \chi') \models \varphi\).

- \((M, l, \chi) \models \exists (\varphi \mathcal{U} \psi)\) iff there is a run \((l_1, \chi_1) \xrightarrow{A_1^{t_1}} (l_2, \chi_2) \xrightarrow{A_2^{t_2}} (l_3, \chi_3) \cdots\) starting in \((l, \chi) = (l_1, \chi_1)\) which satisfies the simple until condition: there is a position \((i, \delta)\) in the run \((i \geq 1)\) such that
  - \((M, l_i, \chi_i + \delta) \models \varphi_{+T_i(i,\delta)} \psi;\) and
  - \((M, l_j, \chi_j + \epsilon) \models \varphi_{+T_i(j,\epsilon)} \varphi \vee \psi,\) for every position \((j, \epsilon)\) before \((i, \delta)\); and
  - Either step \(j\) is a time step \((i.e., \delta_j \neq -)\), or \(A_j^t \models \alpha\), for every \(j\) with \(1 \leq j < i\).

- \((M, l, \chi) \models \forall (\varphi \mathcal{U} \psi)\) iff every complete run starting in \((l, \chi)\) satisfies the simple until condition.

- \((M, l, \chi) \models \exists (\varphi \mathcal{U} \beta \psi)\) iff there is a run \((l_1, \chi_1) \xrightarrow{A_1^{t_1}} (l_2, \chi_2) \xrightarrow{A_2^{t_2}} (l_3, \chi_3) \cdots\) starting in \((l, \chi) = (l_1, \chi_1)\) which satisfies the double until condition: there is a position \((i, 0)\) in the run \((i > 1)\) such that
  - \((M, l_i, \chi_i) \models \varphi_{+T_i(i,0)} \psi;\) and
  - \((M, l_j, \chi_j + \epsilon) \models \varphi_{+T_i(j,\epsilon)} \varphi,\) for every position \((j, \epsilon)\) before \((i, 0)\); and
  - \(A_j^t \models \beta;\) and
  - Either step \(j\) is a time step, or \(A_j^t \models \alpha,\) for every \(j\) with \(1 \leq j < i - 1\).

- \((M, l, \chi) \models \forall (\varphi \mathcal{U} \beta \psi)\) iff every complete run starting in \((l, \chi)\) satisfies the double until condition.

In the simple until condition, the begin of the run has to satisfy \(\varphi \vee \psi\). This is to allow for conditions like \(\exists(c \leq 5 \text{ any } \varphi \psi(c > 5))\), which are intuitively true. But one cannot produce a position in any run before which \(c \leq 5\) is true and at which the formula \(c > 5\) is true, as \(c > 5\) describes a left-open interval of time values. For the double until condition of \(\exists(\varphi \mathcal{U} \beta \psi)\), this is not necessary, for there is a clear point \((viz.,\) immediately after the execution of a \(\beta\) action, at position \((i, 0)\)) where the validity of \(\psi\) is tested.

The double until condition requires that the run contain at least one \(\beta\) action.

**Example 3.6.1** An ATCTL formula corresponding to “It is possible to reach a blue state without executing a green action” is:

\[ \exists_{\text{green}} \Diamond \text{blue} \]

---

\(^7\)One could also define: “... iff there is a complete run ...” The two definitions are equivalent for the same reasons as the CTL definitions are equivalent.
Chapter 3. Real-Time and Action Language: ATCTL

or, without abbreviations:

\[ \exists (\neg \perp, \neg \mathsf{green} \mathcal{U} \, \mathsf{blue}) \]

The following formula corresponds to “In a red state, it is not possible to reach a yellow state within 3 time units”:

\[ \text{reset } c \text{ in } (\text{red} \rightarrow \neg \exists (\text{any} \diamond c \leq 3 \land \text{yellow})) \]

### 3.7 Timed CTL

TCTL is a temporal logic without action modalities introduced by Alur et al. [2, 48]. It extends CTL [30] by real time. There is a model checker Kronos [14, 108] for TCTL, so if we can translate ATCTL formulas and models to TCTL, we can check ATCTL formulas.

We can see TCTL as ATCTL over the empty action symbol set (A = \(\emptyset\)) and without the operators \(X_a\) or \(\mathcal{U}_B\). They are omitted because without action labels, there is no unique “next state” notion in a continuous-time setting. So, TCTL syntax is defined as:

\[ \varphi, \psi ::= \bot \mid p \mid \neg \varphi \mid \varphi \land \psi \mid c \leq n \mid c \geq n \mid \text{reset } c \text{ in } \varphi \mid \exists (\varphi \mathcal{U} \psi) \mid \forall (\varphi \mathcal{U} \psi) \]

The operators have a similar semantics to the corresponding ATCTL operators (see below). We may define abbreviations in the style of Alur et al. [2], e.g., \(\exists (\varphi \mathcal{U} \leq n \psi) \equiv \text{reset } c_{\text{new}} \text{ in } \exists (\varphi \mathcal{U} c_{\text{new}} \leq n \land \psi)\).

**TCTL models.** TCTL models are unlabelled TOA. Here too, we interpret a formula \(\varphi\) over a state \((l, \chi)\) of an unlabelled TOA \(T = (L, C, \Delta, \mathcal{J}, l_0)\) and a partial time stamp \(v : C \rightarrow \mathbb{R}_0^+\). The definition of the satisfaction relation \((T, l, \chi) \models v \varphi\) is a simple adaptation of the corresponding definition for ATCTL:

- \((T, l, \chi) \models v \bot \text{ never.}\)
- \((T, l, \chi) \models v p \iff p \in \mathcal{J}(l).\)
- \((T, l, \chi) \models v \neg \varphi \iff (T, l, \chi) \not\models v \varphi.\)
- \((T, l, \chi) \models v \varphi \land \psi \text{ iff } (T, l, \chi) \models v \varphi \text{ and } (T, l, \chi) \models v \psi.\)
- \((T, l, \chi) \models v c \leq n \text{ iff } f(c) \leq n, \text{ where } f(c) = \begin{cases} v(c) & \text{if } v(c) \text{ is defined} \\ \chi(c) & \text{otherwise.} \end{cases}\)
- \((T, l, \chi) \models v c \geq n \text{ iff } f(c) \geq n, \text{ where } f(c) \text{ is defined as above.}\)
- \((T, l, \chi) \models v \text{ reset } c \text{ in } \varphi \text{ iff } (T, l, \chi) \models v[(c) := 0] \varphi.\)
- \((T, l, \chi) \models v \exists (\varphi \mathcal{U} \psi) \text{ iff there is a run } (l_1, \chi_1) \xrightarrow{\delta_1} (l_2, \chi_2) \xrightarrow{\delta_2} (l_3, \chi_3) \cdots \text{ starting in } (l, \chi) = (l_1, \chi_1) \text{ which satisfies the until condition: there is a position } (i, \delta) \text{ in the run such that}
- \quad (T, l_i, \chi_i + \delta) \models v_{+T_i(i, \delta)} \psi; \text{ and}
- \quad (T, l_j, \chi_j + \epsilon) \models v_{+T_j(i, \delta)} \varphi \lor \psi, \text{ for every position } (j, \epsilon) \text{ before } (i, \delta).\)
- \((T, l, \chi) \models v \forall (\varphi \mathcal{U} \psi) \text{ iff every complete run starting in } (l, \chi) \text{ satisfies the until condition.}\)
3.8 Reduction of ATCTL to TCTL

We now define a reduction of ATCTL formulas and models to TCTL. It extends the reduction from ACTL to CTL defined by De Nicola and Vaandrager [28]. Most of the operators of ATCTL have a direct correspondence in TCTL. However, we had to find a workaround for the operators $X_{a}$ and $\mu_{\beta}$.

**Action locations.** Because transitions of TCTL models are unlabelled, we introduce an action location for every transition and action propositions, as De Nicola and Vaandrager [28] do. In every action location, the action propositions hold that correspond to the original transition’s label. Further, there is one special atomic proposition $ACT$ which holds in all action locations to distinguish them from non-action locations. See figure 3.2 for an example ($\cdot^T$ denotes the translation).

To ensure that actions are instantaneous, we have to add one clock symbol $c_{new}$. Clock constraints in the location invariant forbid to wait in the action location. Transitions are split up into two parts: For every transition $tr = l \xrightarrow{\mathcal{A}} C:0 \rightarrow l'$ of $\mathcal{M}$, we define two transitions in $\mathcal{M}^T$, namely $l \xrightarrow{\mathcal{C} \cup \{c_{new}\} := 0} tr$ and $tr \xrightarrow{\{c_{new} = 0\}} l'$. To remember the action $A$, the corresponding action propositions hold in state $tr$.

**Translation of models.** Assume given a TOA $\mathcal{M} = (L, C, \mathcal{A}, \Delta, AP, \mathcal{J}, l_0)$. We define the TCTL model $\mathcal{M}^T = (L^T, C^T, \Delta^T, AP^T, \mathcal{J}^T, l_0)$ by:

- The set of locations is $L^T = L \cup \Delta$: we add an action location for every transition. The location invariants are defined by: $i^T_l := i_l$, for $l \in L$ and $i^T_{tr} := (c_{new} = 0)$, for $tr \in \Delta$.
- The set $C^T$ of clock symbols is the disjoint union $C \cup \{c_{new}\}$: we add one new clock $c_{new}$ to the clocks of $\mathcal{M}$.
- The transition relation $\Delta^T$ is defined by the rules:

\[
\begin{align*}
&tr \in \Delta, \text{ where } tr = l \xrightarrow{\mathcal{A}} C:0 \rightarrow l' \quad l \xrightarrow{\mathcal{C} \cup \{c_{new}\} := 0} tr \\
&tr \in \Delta, \text{ where } tr = l \xrightarrow{\mathcal{A}} C:0 \rightarrow l' \quad tr \xrightarrow{\{c_{new} = 0\}} l'
\end{align*}
\]

\[\]If there are no other clocks, even $c_{new}$ may be suppressed, as it is no more possible to measure the time consumed by an action.
The reset of $c_{\text{new}}$, the location invariant of $tr$, and the transition invariant $c_{\text{new}} = 0$ jointly take care that the system does not spend time in the action location. Note that all transitions in $\Delta$ and in $\Delta^T$ are output transitions.

- The set of atomic propositions $AP^T$ is the disjoint union $AP \cup A \cup \{ACT\}$.

- The interpretation of the atomic propositions is defined by the rules:
  \[
  \begin{align*}
  p &\in J(l) \quad tr \in T \quad (l, A, C, l') \in T \quad a \in A \\
  p &\in J^T(l) \quad \text{ACT} \in J^T(tr) \\
  \end{align*}
  \]

- The initial location $l_0$ is the same for the ATCTL model and its translation.

Translation of states, steps and runs. Now, we continue with states: as the translated model has one new clock, there is a slight difference between states of $M$ and the states of $M^T$. In the translation of a state, one may choose the value of $c_{\text{new}}$ almost arbitrarily. Given a time stamp $\chi$, we define $\chi^T_r$, where $r$ is the value of $c_{\text{new}}$:

\[
\chi^T_r(c) = \begin{cases} 
  r & \text{if } c = c_{\text{new}} \\
  \chi(c) & \text{otherwise}
\end{cases}
\]

In the translation of steps, we distinguish action and time steps:

- $(l, \chi) \xrightarrow{A} (l', \chi')$ is translated to $(l, \chi^T_r) \xrightarrow{(l', A, l')} (l', \chi'^T_0)$ (where $r$ may be chosen freely).

- $(l, \chi) \xrightarrow{\delta} (l, \chi + \delta)$ is translated to $(l, \chi^T_r) \xrightarrow{\delta} (l, (\chi + \delta)^T_{r+\delta})$.

Action steps are split up into two parts, corresponding to the split transitions. Time steps are translated directly. A run, then, is translated stepwise. As the translation of each individual step is one or two step(s) in $M^T$, their composition is a run again.

This mapping is injective and almost surjective. Only for the new clock symbol $c_{\text{new}}$, not every value is hit. The following lemma makes this clear:

**Lemma 3.8.1** Given a value $r \in \mathbb{R}_0^+$, the extension of the above mapping to complete runs is a bijection between the complete runs of $M$ and the complete runs of $M^T$ starting in non-action locations and having $r$ as the initial value for $c_{\text{new}}$.

**Proof:**

1. The mapping is injective. Given two different complete runs of $M$, they differ in at least one state, so that the respective translations of the states differ, or they differ in at least one time step – but in the translation, the delay $\delta$ is preserved –, or they differ in at least one action step, so that the respective translations contain different action locations.
Translation of formulas. We reinterpret action formulas as a special kind of propositional formulas, for we have reinterpreted action symbols as a special kind of proposition symbols:

\[
\begin{align*}
\text{any}^T &= \text{ACT} \\
(-a)^T &= -(a^T) \\
(a + \beta)^T &= a^T \lor \beta^T
\end{align*}
\]

Propositional formulas are translated in a straightforward manner. ATCTL operators are mostly translated to the corresponding TCTL operators:

\[
\begin{align*}
\bot^T &= \bot \\
p^T &= p \\
(-\varphi)^T &= -\varphi^T \\
(\varphi \land \psi)^T &= \varphi^T \land \psi^T \\
(c \leq n)^T &= c \leq n \\
(c \geq n)^T &= c \geq n \\
\text{(reset } c \text{ in } \varphi)^T &= \text{reset } c \text{ in } \varphi^T \\
(\exists x_a \varphi)^T &= \exists (-\text{ACT } U=0 \exists (\text{ACT } \land a^T U=0 \varphi^T)) \\
(\forall x_a \varphi)^T &= \forall (-\text{ACT } U=0 \exists (\text{ACT } \land a^T U=0 \varphi^T)) \\
\exists (\varphi \land \psi)^T &= \exists ((\text{ACT } \land a^T) \lor (-\text{ACT } \land \varphi^T) \ U \neg \text{ACT } \land \psi^T) \\
\forall (\varphi \land \psi)^T &= \forall ((\text{ACT } \land a^T) \lor (-\text{ACT } \land \varphi^T) \ U \neg \text{ACT } \land \psi^T) \\
\exists (\varphi \land \psi)^T &= \exists ((\text{ACT } \land a^T) \lor (-\text{ACT } \land \varphi^T) \ U \exists (\text{ACT } \land \beta^T U=0 \neg \text{ACT } \land \psi^T)) \\
\forall (\varphi \land \psi)^T &= \forall ((\text{ACT } \land a^T) \lor (-\text{ACT } \land \varphi^T) \ U \exists (\text{ACT } \land \beta^T U=0 \neg \text{ACT } \land \psi^T))
\end{align*}
\]

The \( \exists (\varphi \ U=0 \ \psi) \) operator is an abbreviation for: reset \( c'_{\text{new}} \) in \( \exists (\varphi \ U \ (c'_{\text{new}} = 0 \land \psi)) \), where \( c'_{\text{new}} \) denotes a new clock symbol.

Example 3.8.1 We translate the formulas from example 3.6.1:

\[
\begin{align*}
\exists (\bot \land \neg \text{green} U \text{ blue})^T &= \exists ((\text{ACT } \land \neg \text{green}) \lor (-\text{ACT } \land \bot) U \neg \text{ACT } \land \text{blue}) \\
\text{reset } c \text{ in } (\text{red } \rightarrow \neg \exists (\bot \land \text{any } c \leq 3 \land \text{yellow}))^T &= \\
\text{reset } c \text{ in } (\text{red } \rightarrow \neg \exists ((\text{ACT } \land \text{ACT}) \lor (-\text{ACT } \land \bot) U \neg \text{ACT } \land (c \leq 3 \land \text{yellow})))
\end{align*}
\]
Chapter 3. Real-Time and Action Language: ATCTL

**Theorem 3.8.1** Let $\mathcal{M} = (L, C, A, D, AP, \mathcal{J}, J_0)$ be a TOA, $(l, \chi)$ one of its states, $\varphi$ an ATCTL formula, $v$ a partial time stamp, and $r \in \mathbb{R}_0^+$. Then,

$$(\mathcal{M}, l, \chi) \models_v \varphi \iff (\mathcal{M}^T, l, \chi^T_r) \models_v \varphi^T$$

**Proof:** By induction over the structure of the formulas. Base case:

- $(\mathcal{M}, l, \chi) \models_v \bot \iff (\mathcal{M}^T, l, \chi^T_r) \models_v \bot^T$: trivially.
- $(\mathcal{M}, l, \chi) \models_v p \iff (\mathcal{M}^T, l, \chi^T_r) \models_v p^T$: The left side holds iff $p \in \mathcal{J}(l)$, and the right side holds iff $p \in \mathcal{J}(l)^T$. The restriction of $\mathcal{J}^T$ to non-action locations is $\mathcal{J}$.

Induction step:

- $(\mathcal{M}, l, \chi) \models_v \neg \varphi \iff (\mathcal{M}^T, l, \chi^T_r) \models_v (\neg \varphi)^T$: By induction hypothesis.
- $(\mathcal{M}, l, \chi) \models_v \varphi \land \psi \iff (\mathcal{M}^T, l, \chi^T_r) \models_v (\varphi \land \psi)^T$: By induction hypothesis.
- $(\mathcal{M}, l, \chi) \models_v c \leq n \iff (\mathcal{M}^T, l, \chi^T_r) \models_v (c \leq n)^T$: The left side holds iff $f(c) \leq n$, where
  $$f(c) = \begin{cases} v(c) & \text{if } v(c) \text{ is defined,} \\ \chi(c) & \text{otherwise.} \end{cases}$$
  The right side holds iff $f^T(c) \leq n$, where $f^T(c)$ is defined similarly for $v$ and $\chi^T_r$. But by definition, $f(c) = f^T(c)$ for $c \neq c_{\text{new}}$.
- $(\mathcal{M}, l, \chi) \models_v c \geq n \iff (\mathcal{M}^T, l, \chi^T_r) \models_v (c \geq n)^T$: Analogous.
- $(\mathcal{M}, l, \chi) \models_v \text{reset } c \text{ in } \varphi \iff (\mathcal{M}^T, l, \chi^T_r) \models_v (\text{reset } c \text{ in } \varphi)^T$: The left side holds iff $(\mathcal{M}, l, \chi) \models_{v[c]=0} \varphi$. The right side holds iff $(\mathcal{M}^T, l, \chi^T_r) \models_{v[c]=0} \varphi^T$. By induction hypothesis, these two are equivalent.
- $(\mathcal{M}, l, \chi) \models_v \exists X_\mathbf{a}\varphi \iff (\mathcal{M}^T, l, \chi^T_r) \models_v (\exists X_\mathbf{a}\varphi)^T$: The left side holds (by definition of $\models$) iff there is a run $(l, \chi) \xrightarrow{\mathbf{A}} (l', \chi') \rightarrow \cdots$ such that $A \models \mathbf{a}$ and $(\mathcal{M}', l', \chi') \models_v \varphi$. This run can be extended to a complete run. Then, by lemma 3.8.1, there is a complete run of $\mathcal{M}^T$, namely $(l, \chi^T_r) \xrightarrow{\mathbf{A}} ((l, \mathbf{a}, l'), (\chi^T_r)_0) \xrightarrow{\mathbf{A}} (l', \chi'^T_0) \xrightarrow{\mathbf{A}} \cdots$. It satisfies $(\exists X_\mathbf{a}\varphi)^T \equiv \exists (\neg \mathbf{ACT} \cup \neg 0 \exists (\mathbf{ACT} \land \mathbf{a}^T \cup \varphi^T))$. This is the right-hand side. The converse (the right-hand side implies the left-hand side) uses the correspondence between complete runs of $\mathcal{M}$ and complete runs of $\mathcal{M}^T$ starting in non-action locations in a similar way.
- $(\mathcal{M}, l, \chi) \models_v \forall X_\mathbf{a}\varphi \iff (\mathcal{M}^T, l, \chi^T_r) \models_v (\forall X_\mathbf{a}\varphi)^T$: Analogous.
- $(\mathcal{M}, l, \chi) \models_v \exists (\mathbf{a} \cup \psi) \iff (\mathcal{M}^T, l, \chi^T_r) \models_v (\exists (\mathbf{a} \cup \psi)^T$: The left-hand side holds iff there is a run $(l_1, \chi_1) \xrightarrow{\delta_1} (l_2, \chi_2) \xrightarrow{\delta_2} (l_3, \chi_3) \cdots$ which satisfies the simple until condition. This run can be extended to a complete run; then, by lemma 3.8.1, there is a corresponding complete run of $\mathcal{M}^T$. Let $(i, \delta)$ be the position used in the definition of $\cdots \models_v \exists (\mathbf{a} \cup \psi)$, and let $(i^T, \delta)$ be the corresponding position in the translated run. The translated run satisfies the until condition for the right-hand side:
3.9. Action-Based CTL

ACTL is a logic with action modalities, but (originally) with ⊥ as only atomic proposition and without real time, introduced by De Nicola and Vaandrager [27, 28]. It adapts CTL [30] to action-oriented situations. We have added proposition symbols to the original ACTL, so we are able to describe it just as an extension of CTL. Another addition we have made is that the transitions are labelled with a single action.

We can see ACTL as ATCTL over the empty clock set: \( C = \emptyset \). Then, models, transitions and steps don’t bear any real-time information (or, stated differently, every positive real number is acceptable as delay of a time step). Time steps in ACTL models are similar to internal steps denoted by \( \tau \) in process algebra [80]. The clock constraints \( \cdot \leq n, \cdot \geq n \) and the operator reset \( \cdot in \varphi \) cannot be used in ACTL.

De Nicola et al. [27] have constructed a model checker for ACTL which translates ACTL formulas and models to CTL and relies on a CTL model checker to do the rest. So, we may choose to translate ATCTL to ACTL and use this model checker. In the following section, we will show how to do this for a large subset of ATCTL. Although there are ATCTL formulas that Kronos cannot handle, model checking is less efficient this way because the state space is blown up heavily in the reduction we define.

We will define a reduction from ATCTL to ACTL that is similar to the reduction from TCTL to CTL defined by Alur at al. [2]. It consists of constructing a so-called region automaton, whose states are equivalence classes of (timed) states of the original automaton. Because ATCTL formulas cannot distinguish between states in the same equivalence

\[
- (M^T, l, \chi_i) \models \psi_T \iff (M^T, l, \chi_i) \models \psi_T^T; \text{ and}

- (M^T, l, \chi_j^T + \varepsilon) \models \psi_T^T \iff \psi_T^T.
\]

These two conditions are easily verified by induction. Note that \( (\text{ACT} \land T^T) \) is satisfied by the action locations inserted into the translation.

The only if part holds because the mapping of runs is surjective.

- \( (M, l, \chi) \models \forall \psi \iff (M^T, l, \chi^T) \models \forall \psi \): Analogous.

- \( (M, l, \chi) \models \exists \psi \iff (M^T, l, \chi^T) \models \exists \psi \): The left-hand side holds iff there is a run \( (l_1, \chi_1) \xrightarrow{\delta_1} (l_2, \chi_2) \xrightarrow{\delta_2} (l_3, \chi_3) \rightarrow \cdots \) which satisfies the double until condition. As with the simple until, translate this run to a run of \( M^T \); the translation will satisfy the translated formula. The only difference is that we have to reach a state where \( \exists \psi \models \text{ACT} \land \text{ACT} \land T^T \models \psi_T \) holds. This is the action state immediately before the position \( (i, 0) \) of the original run where \( \psi \) becomes true.

- \( (M, l, \chi) \models \forall \psi \iff (M^T, l, \chi^T) \models \forall \psi \): Analogous.

There is some asymmetry in our definition: In most process algebras, \( \tau \) steps may change the state of the system thoroughly. In our setting, only transitions may become enabled or disabled (because of their clock constraints) after an internal step; no propositional formula changes its truth value.
class, the region automaton approximates the original behaviour finely enough for model checking.

**ACTL syntax.** Recall that the set $AP = \{p,q,\ldots\}$ of proposition symbols and the set $A = \{a,b,\ldots\}$ of action symbols are given.\(^{10}\) We repeat the constructs of ATCTL which make up ACTL:

$$\varphi,\psi ::= \bot \mid p \mid \neg \varphi \mid \varphi \wedge \psi \mid$$

$$\exists X_a \varphi \mid \forall X_a \varphi \mid \exists (\varphi \, a \uparrow \downarrow \psi) \mid \forall (\varphi \, a \uparrow \downarrow \psi) \mid \exists (\varphi \, a \uwave \psi) \mid \forall (\varphi \, a \uwave \psi)$$

The operators have semantics similar to the corresponding ATCTL operators. Abbreviations like $\exists \diamond_a \varphi$ are defined similarly to the CTL abbreviations.

**ACTL models.** Output automata are ACTL models. (The original ACTL models were called labelled transition systems.) They are similar to ATCTL models without clocks.

There are no time steps in output automata; however, in the translation of ATCTL models, we will distinguish “normal” action steps from $T$ steps; the latter will serve to translate time steps. Formally, we assume that the set $A$ of action symbols contains the normal labels (i.e., subsets of $A$) and the special action symbol $T$. So, a transition in an ACTL model has the form:

$$s_1 \xrightarrow{A} s_2 \quad \text{where} \quad A \subseteq A, \quad \text{or} \quad$$

$$s_1 \xrightarrow{T} s_2$$

If we speak about an ACTL transition in general, we sometimes write $s_1 \xrightarrow{A^T} s_2$, where either $A^T \subseteq A$ or $A^T = T$.

**ACTL semantics.** Assume given an ACTL-model $L = (S,A,D,AP,\mathcal{I},s_0)$. We interpret a formula in a state $s$ of the model. The satisfaction relation $(L,s) \models \varphi$ is similar to the satisfaction relation of ATCTL, but doesn’t include time stamps:

- $(L,s) \models \bot$ never.
- $(L,s) \models p$ iff $p \in \mathcal{I}(s)$.
- $(L,s) \models \neg \varphi$ iff $(L,s) \not\models \varphi$.
- $(L,s) \models \varphi \land \psi$ iff $(L,s) \models \varphi$ and $(L,s) \models \psi$.
- $(L,s) \models \exists X_a \varphi$ iff there is a run $s \xrightarrow{A} s' \to \ldots$ with $A \models a$ and $(L,s') \models \varphi$.
- $(L,s) \models \forall X_a \varphi$ iff every complete run starting in $s$ has the form $s \xrightarrow{A} s' \to \ldots$ and satisfies $A \models a$ and $(L,s') \models \varphi$.
- $(L,s) \models \exists (\varphi \, a \uparrow \downarrow \psi)$ iff there is a run $s_1 \xrightarrow{A_1} s_2 \xrightarrow{A_2} s_3 \cdots$ starting in $s = s_1$ which satisfies the simple until condition: there is a position $i \geq 1$ such that

---

\(^{10}\) Note that the “internal” action symbol $T$ or $\tau$ is not an element of $A$ and cannot be used in action terms.
3.10 Reduction of ATCTL to ACTL

The reduction to ACTL, eliminating real time, is a bit more complicated than the reduction to TCTL. It is similar to the reduction from TCTL to CTL given by Alur et al. [2].

We construct a region automaton. This is a finite automaton where a state is an equivalence class of states in the ATCTL model: we abstract as much as possible from the exact values of clocks. (The original ATCTL model contains infinitely many states, as there are infinitely many possible time stamps.) Still, the region automaton may be a very large automaton, too large for current model checkers. The construction is similar to the reduction from TCTL to CTL defined by Alur, Courcoubetis, and Dill [2].

The correspondence between an ATCTL model and its region automaton is a bit looser than the correspondence between an ATCTL model and its translated TCTL model: As the clock information in the translated model is lost, the mapping is no more injective. We propose two different translations of formulas, both incomplete: the straightforward translation, which is exposed completely, lacks a mapping of the reset \( \cdot \) in \( \varphi \) operator. It is difficult to translate because the reset \( \cdot \) in \( \varphi \) operator constructs a relation between states which are completely unrelated in the translated model. An alternative translation, which is explained briefly in section 3.11, introduces special transitions to create this relation, but cannot map the \( \forall \) operators.

Equivalent states. Assume given a TOA \( \mathcal{M} = (L, C, \mathcal{A}, \Delta, AP, J, l_0) \). We first will define an equivalence relation of time stamps \( \chi \) of the TOA; that leads us to an equivalence relation of states.

Note that all clock constraints \( t \leq n \) and \( t \geq n \) contain only natural numbers \( n \). In an ATCTL formula, one can express which clock is the first to change its integer part, but one cannot constrain the exact value of its fractional part. So, time stamps with the same integer parts and the same ordering of the fractional parts cannot be distinguished by
ATCTL formulas; we declare them equivalent. Further, assume given an $N$ such that that in every interesting formula $t \leq n$ or $t \geq n$, we have $n \leq N$.\footnote{In a practical situation, one only deals with a finite set of interesting formulas (or, properties that must be checked); in this set, clock constraints are always bound by some $N \in \mathbb{N}$.} Time stamps whose clocks exceed $N$ are declared equivalent. The equivalence relation is called $N$-similarity.

So, given two time stamps $\chi$ and $\chi'$, let $R_\chi = \{ c \in \mathcal{C} \mid \chi(c) \leq N \}$, and define $R_\chi'$ similarly. The two time stamps are $N$-similar, denoted $\chi \sim_N \chi'$, if:

- The same clocks are in the relevant area: $R_\chi = R_\chi'$; and
- For every $c \in R_\chi$, the integer parts are the same: $\lfloor \chi(c) \rfloor = \lfloor \chi'(c) \rfloor$; and
- For every pair of $c, c' \in R_\chi$, the orderings of the fractional parts are the same: $|\chi(c) - \chi(c')| = |\chi'(c) - \chi'(c')|$;\footnote{This clause is equivalent to:}
- For every $c \in R_\chi$, the fractional parts are both zero or both non-zero: $\chi(c) = \lfloor \chi(c) \rfloor$ iff $\chi'(c) = \lfloor \chi'(c) \rfloor$.

In this definition, $[r]$ denotes the integer part of the real number $r$, such that $r - 1 < [r] \leq r \wedge [r] \in \mathbb{Z}$.

The equivalence relation is then extended to states by: two states $(l, \chi)$ and $(l', \chi')$ are $N$-similar if $l = l'$ and $\chi \sim_N \chi'$. The equivalence classes of this extended relation are regions, written as $[(l, \chi)]_\sim$.

### 3.10.1 Translation of Models

The region automaton is the translation of an ATCTL model. Its locations are the regions; its transitions correspond to the transitions in the ATCTL model. Define an ACTL model $\mathcal{M}^A = (S^A, \mathcal{A}^A, \Delta^A, AP^A, I^A, s_0^A)$ (this is the region automaton) by:

- The set of states is $S^A = \{ [(l, \chi)]_\sim \mid l \in L, \chi : \mathcal{C} \to \mathbb{R}_0^+ \}$: states are the equivalence classes defined above.
- The set of labels is $\mathcal{A}^A = \mathcal{A} \cup \{ \top \}$
- The step relation is defined by:
  - For every action step $(l, \chi) \xrightarrow{A} (l', \chi')$ of $\mathcal{M}$, there is a step $[(l, \chi)]_\sim \xrightarrow{A} [(l', \chi')]_\sim$.
  - For every time step $(l, \chi) \xrightarrow{\delta} (l, \chi')$ of $\mathcal{M}$, there is a step $[(l, \chi)]_\sim \xrightarrow{\top} [(l, \chi')]_\sim$.
- The set of atomic propositions is $AP^A = AP \cup \{ c^{\leq n}, c^{\geq n} \mid c \in \mathcal{C} \land 0 \leq n \leq N \}$.
- The interpretation of the atomic propositions is defined by the rules:

\[
\begin{align*}
\text{If } p \in \mathcal{I}(l) & \Rightarrow p \in I^A([(l, \chi)]_\sim) \\
\chi(c) \leq n & \Rightarrow c^{\leq n} \in I^A([(l, \chi)]_\sim) \\
\chi(c) \geq n & \Rightarrow c^{\geq n} \in I^A([(l, \chi)]_\sim)
\end{align*}
\]

For every pair of $c, c' \in R_\chi$, we have $\text{fract}(\chi(c)) < \text{fract}(\chi(c'))$ iff $\text{fract}(\chi'(c)) < \text{fract}(\chi'(c'))$ (where $\text{fract}(r) = r - [r]$).
3.10. Reduction of ATCTL to ACTL

- The initial state is \( s_0^A = ([l_0, \chi_0])_\sim \), where \( \chi_0 \) is the timestamp that evaluates to 0 for every clock.

3.10.2 Translation of States, Steps and Runs

States are mapped straightforward: any state \((l, \chi)\) is mapped to its corresponding equivalence class \([([l, \chi])_\sim \). 

We map a step \((l, \chi) \xrightarrow{A^T_i \delta_i} (l', \chi')\) to \([([l, \chi])_\sim \xrightarrow{A^T_i \delta_i} ([l', \chi'])_\sim \). 

Runs of \( \mathcal{M} \), then, are mapped to runs of \( \mathcal{M}^A \) state- and stepwise, by composing the mappings of single steps. Let \((l_1, \chi_1) \xrightarrow{A^T_1 \delta_1} (l_2, \chi_2) \xrightarrow{A^T_2 \delta_2} (l_3, \chi_3) \rightarrow \cdots \) be a run of \( \mathcal{M} \). This run is mapped to \([([l_1, \chi_1])_\sim \xrightarrow{A^T_2 \delta_2} ([l_2, \chi_2])_\sim \xrightarrow{A^T_3 \delta_3} ([l_3, \chi_3])_\sim \rightarrow \cdots \). Recall that we allow Zeno runs, so there is no need to impose constraints on the time stamps.\(^\text{13}\)

**Lemma 3.10.1** The construction above maps runs of \( \mathcal{M} \) onto runs of \( \mathcal{M}^A \).

**Proof:** Let \((l_1, \chi_1) \xrightarrow{A^T_1 \delta_1} (l_2, \chi_2) \xrightarrow{A^T_2 \delta_2} (l_3, \chi_3) \rightarrow \cdots \) be a run of \( \mathcal{M} \). First, we have to show that the result of the construction is a run. This is clear because for every step of \( \mathcal{M} \), there is a step in \( \Delta^A \). So, in particular, for the ATCTL step \((l_i, \chi_i) \xrightarrow{A^T_i \delta_i} (l_{i+1}, \chi_{i+1}), \) there is a ACTL step \([([l_i, \chi_i])_\sim \xrightarrow{A^T_i \delta_i} ([l_{i+1}, \chi_{i+1}])_\sim \). 

Second, we have to show that every run of \( \mathcal{M}^A \) is hit. Let \([([l_1, \chi_1])_\sim \xrightarrow{A^T_1 \delta_1} ([l_2, \chi_2])_\sim \xrightarrow{A^T_2 \delta_2} ([l_3, \chi_3])_\sim \rightarrow \cdots \) be a run of \( \mathcal{M}^A \). We construct a run of \( \mathcal{M} \) which maps to it inductively. To begin, choose some arbitrary \((l_1, \chi_1') \in [([l_1, \chi_1])_\sim \). Then assume that the run has been constructed up to \( \cdots \xrightarrow{A^T_{i-1} \delta_{i-1}} (l_i, \chi_i') \). Look at the type of \( A^T_i \).

- Translating an action step back is easy because the locations of \( \mathcal{M} \) and the action set \( A_i \) are preserved by the translation. It is possible to choose \( C \subseteq C \) such that \( \mathcal{M} \) has a transition \((l_i, A^T_i C, l_{i+1}) \) or \( l_i \xrightarrow{A^T_i \delta_i C} l_{i+1} \). The transition invariant \( \delta \) is satisfied because an action step of \( \mathcal{M}^A \) is based on an action step of \( \mathcal{M} \), which satisfies \( \delta \) by definition. Then, \((l_i, \chi_i') \xrightarrow{A^T_i \delta_i} (l_{i+1}, \chi_{i+1}'[C_i := 0]) \) is the step to be added next.

- When translating a time step back \((l_i, \chi_i') \xrightarrow{T} (l_{i+1}, \chi_{i+1}') \), we have to look for a valid value for \( \delta_i \) such that \((l_i, \chi_i + \delta_i) \in [([l_{i+1}, \chi_{i+1}])_\sim \) (Note that \( l_i = l_{i+1} \)). Clearly, there are \((l, \chi) \in [([l, \chi])_\sim \) (where \( \chi \) may differ from \( \chi_i \)) and \((l, \chi') \in [([l_{i+1}, \chi_{i+1}])_\sim \) and \( \delta \in \mathbb{R}_0^+ \) such that \((l, \chi) \xrightarrow{T \delta} (l, \chi') \). Without loss of generality, 

\(^{13}\)One could describe non-Zenolessness, using the property that a clock that isn’t reset grows above any bound. So, either a clock is reset in infinitely many steps, or it exceeds \( N \) in infinitely many steps. This could be translated to a family of “fairness sets:” for every clock \( c \), a run has to visit the set \( \{([l, \chi])_\sim \ | \ \chi(c) = 0 \vee \chi(c) > N \} \) infinitely often. In addition, we have to require that there are infinitely many \( T \) steps in an infinite run. A run satisfying both conditions is the translation of some non-Zeno run of the original model.
3.10.3 Translation of Formulas

The translation

Theorem 3.10.1

We can then formulate an equivalence theorem similar to theorem 3.8.1:

\[
\frac{\text{reset } c}{\text{is an immediate successor of } ((l, \chi) \rhd c)}
\]

A boundary class is an equivalence class \([(l, \chi)]\sim\) which satisfies the condition that by waiting an arbitrarily small time \(\varepsilon > 0\), one reaches a different class: \((l, \chi + \varepsilon) \notin [(l, \chi)]\sim\). The reader will easily convince himself that of two immediately succeeding classes, one is a boundary class. (Note that a time stamp where some fraction part is 0 is not \(N\)-similar to one where this is not the case.)

So, either \([(l, \chi_i)]\sim\) is a boundary class; then choose \(0 < \delta_i < \min\{\{1 - \text{fract}(\chi'_i(c)) \mid c \in C}\}\). Or, \([(l_i+1, \chi_{i+1})]_\sim\) is a boundary class; then let \(\delta_i = \min\{\{1 - \text{fract}(\chi'_i(c)) \mid c \in C}\}\).

The reader will easily see that \(\delta_i\), chosen this way, has the required properties.

3.10.3 Translation of Formulas

\[
\begin{align*}
\bot^A &= \bot \\
p^A &= p \\
(c \leq n)^A &= c^\leq_n \\
(c \geq n)^A &= c^\geq_n \\
(\neg \varphi)^A &= \neg(\varphi^A) \\
(\varphi \land \psi)^A &= \varphi^A \land \psi^A \\
(\exists X \varphi)^A &= \exists X \varphi^A \\
(\forall X \varphi)^A &= \forall X \varphi^A \\
(\varphi_\text{a} \land \psi)^A &= \exists(\varphi_\text{a} \land \psi^A) \\
(\forall \varphi_\text{a} \land \psi^A) &= \forall(\varphi_\text{a} \land \psi^A) \\
(\varphi_\text{a} \land \psi^A) &= \exists(\varphi_\text{a} \land \psi^A) \\
(\forall \varphi_\text{a} \land \psi^A) &= \forall(\varphi_\text{a} \land \psi^A)
\end{align*}
\]

The translation (reset \(c\) in \(\varphi^A\)) is not defined.

3.10.4 Equivalence theorem

We can then formulate an equivalence theorem similar to theorem 3.8.1:

**Theorem 3.10.1** Let \(M = (L, \mathcal{C}, \mathcal{A}, \Delta, AP, J, I_0)\) be a TOA, \((l, \chi)\) one of its states and \(\varphi\) an ATCTL formula without \(\cdot\) in \(\cdot\) operators. Then,

\[
\models (M, l, \chi) \quad \text{iff} \quad (M^A, [(l, \chi)]_\sim) \models \varphi^A
\]

**Proof:** By induction over the formulas.

- \((M, l, \chi) \models \bot\) iff \((M^A, [(l, \chi)]_\sim) \models \bot^A\): trivial.
- \((M, l, \chi) \models p\) iff \((M^A, [(l, \chi)]_\sim) \models p^A\): The left-hand side holds iff \(p \in J(l)\), and the right-hand side holds iff \(p \in I^A([(l, \chi)]_\sim)\). Trivial.
- \((M, l, \chi) \models \neg \varphi\) iff \((M^A, [(l, \chi)]_\sim) \models (\neg \varphi)^A\): By induction.
- \((M, l, \chi) \models \varphi \land \psi\) iff \((M^A, [(l, \chi)]_\sim) \models (\varphi \land \psi)^A\): By induction.
- \((M, l, \chi) \models c \leq n\) iff \((M^A, [(l, \chi)]_\sim) \models (c \leq n)^A\): The left-hand side holds iff \(\chi(c) \leq n\), and the right-hand side holds iff \(c^\leq_n \in I^A([(l, \chi)]_\sim)\). By definition of \(I^A\).
3.11 Alternative Translation

- $(M, l, \chi) \models c \geq n$ iff $(M^\mathcal{A}, [(l, \chi)]_\sim) \models (c \geq n)^\mathcal{A}$: Analogous.

- $(M, l, \chi) \models \exists \chi \mathcal{A} \phi$ iff $(M^\mathcal{A}, [(l, \chi)]_\sim) \models (\exists \chi \mathcal{A} \phi)^\mathcal{A}$: The left-hand side holds iff there is a run $(l, \chi) \xrightarrow{A} (l', \chi') \rightarrow \cdots$ such that $A \models \chi$ and $(M, l', \chi') \models \phi$. Then, there is a run of $M^\mathcal{A}$, namely $[(l, \chi)]_\sim \xrightarrow{A} [(l', \chi')_\sim] \rightarrow \cdots$. It satisfies $\exists \chi \mathcal{A} \phi^\mathcal{A}$, which is the right-hand side, by induction.

On the other hand, if the right-hand side holds, there is a run $[(l, \chi)]_\sim \xrightarrow{A} [(l', \chi')]_\sim \rightarrow \cdots$, which corresponds to a run of $M$. This run is a witness for the left-hand side.

- $(M, l, \chi) \models \forall \chi \mathcal{A} \phi$ iff $(M^\mathcal{A}, [(l, \chi)]_\sim) \models (\forall \chi \mathcal{A} \phi)^\mathcal{A}$: Analogous.

- $(M, l, \chi) \models (\exists \phi \mathcal{U} \psi) \phi$ holds iff there is a run $(l_1, \chi_1) \xrightarrow{A_{l_1}} (l_2, \chi_2) \xrightarrow{A_{l_2}} (l_3, \chi_3) \rightarrow \cdots$ which satisfies the simple until condition. Let $(i, \delta)$ be the position used in the definition of the until simple condition. This translates to the ACTL simple until condition:

\[-(M, [(l, \chi_i + \delta)]_\sim) \models \phi^\mathcal{A}; \text{ and}\]
\[-(M, [(l, \chi_j + \varepsilon)]_\sim) \models \phi^\mathcal{A} \lor \psi^\mathcal{A}, \text{ for every position } (j, \varepsilon) \text{ before } (i, \delta); \text{ and}\]
\[-A_{l_j}^i \models T \text{ or } A_{l_j}^i \models \phi^\mathcal{A}, \text{ for every } 1 \leq j < i.\]

These three conditions are easily verified by induction, so $(M^\mathcal{A}, [(l, \chi)]_\sim) \models (\exists \phi \mathcal{U} \psi)^\mathcal{A}$ holds.

On the other hand, if the right-hand side holds, there is a run that witnesses it and we can find the corresponding run of $M$ that is a witness for the left-hand side.

- $(M, l, \chi) \models \forall \phi \mathcal{U} \psi$ iff $(M^\mathcal{A}, [(l, \chi)]_\sim) \models (\forall \phi \mathcal{U} \psi)^\mathcal{A}$: Analogous.

- $(M, l, \chi) \models (\exists \phi \mathcal{U} \beta \psi)$ iff $(M^\mathcal{A}, [(l, \chi)]_\sim) \models (\exists \phi \mathcal{U} \beta \psi)^\mathcal{A}$: The proof is similar to the simple until.

- $(M, l, \chi) \models \forall (\phi \mathcal{U} \beta \psi)$ iff $(M^\mathcal{A}, [(l, \chi)]_\sim) \models (\forall (\phi \mathcal{U} \beta \psi)^\mathcal{A}$: Analogous.

3.11 Alternative Translation

The operator reset $c$ is difficult to translate because it relates two states (the current state and a similar state where $c$ is zero) which are completely unrelated in the translated ACTL model. However, we can define an alternative translation which contains such a relation, by adding steps we call clock reset steps. Translations of modal operators are defined in a way that they ignore the clock reset steps. reset $c$ in $\phi$ is translated (speaking informally) to “after taking the $c$-clock reset step, (the translation of) $\phi$ holds.”
Translation of models. Assume given a TOA $M = (L, C, A, \Delta, AP, J, i_0)$. We want to define a (labelled) OA $M^A = (S^A, A^A, \Delta^A, AP^A, I^A, s_0^A)$.

- The set of action symbols is $A^A = A \cup \{T\} \cup \{CR_c|c \in C\}$.
- The step relation is defined by the two rules given above and the additional rule:

$$
\begin{align*}
[(l, \chi)]_\sim & \in S^A \\
(l, \chi) & \xrightarrow{CR_c} [(l, \chi[\{c\} := 0])_\sim
\end{align*}
$$

- The other parts are translated as above.

This construction maps runs of $M$ onto runs without clock reset steps of $M^A$.

Translation of formulas. The translation of action modalities has to be chosen such that clock reset steps are only taken by a reset $t$ in $\phi$ operator. The set of clocks $C$ is finite, so we can write $C = \{c_1, c_2, \ldots, c_m\}$ for some $m \in \mathbb{N}$. Define the abbreviation $CR := CR_{c_1} + CR_{c_2} + \cdots + CR_{c_m}$.

$$
\begin{align*}
(\exists X_{\exists} \phi)^A &= \exists X_{\exists & CR} \phi^A \\
(\forall X_{\forall} \phi)^A &= (\forall X_{\forall & CR} \top) \land \exists X_{\forall - CR} \phi^A \\
(\exists X_{\exists & U} \psi)^A &= \exists X_{\exists & CR & CR} \psi^A \\
(\exists X_{\exists & U} \psi)^A &= \exists X_{\exists & CR & CR} \psi^A \\
(reset c \text{ in } \phi)^A &= \exists X_{CR_c} \phi^A
\end{align*}
$$

The translation of the operators $\forall_{\exists U}$ and $\forall_{\exists U \beta}$ is not defined. So, while we now could define a translation for reset $c$ in $\phi$, some other formulas cannot be translated any more.

Discussion. The additional steps have a clear disadvantage: some $\forall$ operators cannot be translated any more, as there are additional runs in the ACTL model that do not correspond to some original run. Therefore, lemma 3.10.1 does not hold for the alternative translation.

As both translations are incomplete, an engineer will have to resort to other means to specify some properties, for example, he may add an extra clock to the model to measure the desired time. Alur, Courcoubetis, and Dill [2] also needed to do extra work to check timed until properties in TCTL: they defined augmented regions, where an extra clock is automatically added in the model checking algorithm.

3.12 Comparison of the Reductions

We have defined the logics ATCTL and (our variants of) ACTL and TCTL. In our framework, we can give a uniform presentation of ATCTL and the other languages: TCTL is “ATCTL without actions” and ACTL is “ATCTL without clocks.”
Complexity. The translation of formulas (as far as it is defined) is, for both reductions, simple; for TCTL, it is even linear. However, when we look at the translation of models, the differences are evident: The reduction to TCTL $\mathcal{T}$ (just add action locations) is linear in the number of transitions, but the reduction to ACTL (construct the region automaton) is exponential in the number of clocks and the largest clock constraints. (For exact numbers, see e.g. [2].) This is why we decided to implement only the translation to TCTL and let an optimised tool handle the more complex parts.

Problems. We have reduced full ATCTL to TCTL. Some operators without direct correspondence ($\langle a \rangle$ and $\langle a \rangle U \langle b \rangle$) were translated as a combination of TCTL operators. ACTL, on the other hand, has no direct correspondence for the real-time-related operators of ATCTL. We have looked for a translation of reset $c$ in $\mathcal{G}$, but didn’t find a satisfactory solution: also in the alternative translation, some formulas cannot be translated.

We finally have abandoned the research to complete the translation to ACTL and the plans to implement it, because it creates a large region automaton which needs heavy optimisations. The TCTL model checker Kronos already contains some of these optimisations.

3.13 Example: Internal Travel Office

Case description. As an example to show how to use ATCTL in property specification, we model an internal travel office of a large firm. The case is adapted from Verbeek et al. [101].

Imagine an internal travel office in a large firm, e.g. a university. Employees of the firm book business trips via the office. To do this, the office needs a travel permit and a payment allowance.

As an extra service, also private trips can be booked. The cost of a private trip is deducted from the employee’s salary, and no payment allowance is needed.

The office proposes a trip schedule to the employee; as soon as she or he accepts it, the office tries to book a trip and a hotel for the appropriate period. If this succeeds, the employee is invited to come along and to pick up the necessary documents. If the booking fails, the employee is informed and the trip is cancelled; however, the employee is allowed to restart the procedure. (Of course, in a more advanced system, the office would try and look automatically for a similar trip and place to stay.) The financial department handles the payments and, if applicable, the deduction from the salary.

We plan to introduce a workflow system that controls this process. We would like to ensure that no payment is made without allowance, and we use model checking to verify this requirement.

We have modelled the system using TCM. By now, we only handle the parallel composition of (finitely many) statecharts. So, the behaviour of finite object-oriented systems can be modelled.

The system consists of two objects, corresponding to the two departments, and three external entities, namely the employee, an airline and a hotel. It is described by the collaboration diagram in figure 3.3. The two classes’ behaviour is shown in the statecharts of figures 3.4 and 3.5.
Figure 3.3: Typical collaboration of the ITO system.

Figure 3.4: The ITO’s process for a trip.
3.13. Example: Internal Travel Office

The prototype translator (implemented as an extension of TCM) translated these diagrams to a TOA, according to the semantics defined by Eshuis and Wieringa [33]: state names are translated to proposition symbols; actions become labels on the transitions. Then, the translator saved the TOA as a Kronos input file. The TOA has 512 locations.

We have checked the following properties using Kronos:

- “The system is non-Zeno” or, more exactly, “in every state, time may pass at least one second.” This property needs to be checked because the semantics for TOA does not exclude Zeno systems. It can be formalised in ATCTL as:

\[
\text{init} \rightarrow \forall \square [\exists T \text{ any } U T] 
\]

- “No payment without allowance.” A first formalisation of this property is:

\[
\neg \exists (T \text{ allowed} U \text{ pay} T) 
\]

However, Kronos reported that this property does not hold, and provides a counterexample: a private trip can be booked without allowance.

We should state this more precise, saying “without allowance, all payments are deducted from the salary.” This was formalised in ATCTL as:

\[
\neg \exists (T \text{ allowed} U \text{ pay} \& \neg \text{ deduct} T) 
\]

This property was proved by Kronos.

- “No payment without statement of expenses.”

\[
\neg \exists (T \text{ got claim} U \text{ pay} T) 
\]

Note that here, we need not include the clause \& – \text{ deduct}.

Kronos reported that the property holds in all three cases. All execution times were less than one second, measured on a Sun UltraSparc 10 with 256 MB RAM under SunOS 5.8.
3.14 Related Work

The logic CTL was first proposed by Emerson and Clarke [30] to describe requirements on synchronisation between modules in a specification. Satisfiability of CTL is decidable (in exponential time) [31]. CTL models can be checked against a formula in time linear in the size of the model and the formula [19]. So, the logic is suitable for efficient automatic verification.

There are many variants and extensions of CTL. ACTL, defined by De Nicola and Vaandrager [28], adapts CTL with constructs to describe actions. ACTL was mainly designed to bridge the gap between state- and action-oriented specification techniques (for example, Kripke structures are state-oriented; process algebra is action-oriented). TCTL, defined by Alur et al. [2], extends CTL with constructs to specify real-time properties.

ATCTL is a adaptation and combination of these two variants. ATCTL extends both ACTL and TCTL further: First, TCTL does not contain a “next state” operator, but ATCTL does. Second, we have extended ACTL with proposition symbols, so that we can give a more uniform presentation of the logics. ACTL and TCTL, as we present them, are just subsets of ATCTL and supersets of CTL. Third, Alur et al. [2] defined TCTL with a slightly different syntax; we use a syntax similar to Henzinger et al. [48], which separates more clearly the action- and the real-time-oriented parts.

Model checkers. There are several model checkers for CTL and its extensions. A TCTL model checking algorithm is described by Henzinger, Nicollin, Sifakis and Yovine [48]. However, it seems that this algorithm has never been fully implemented. Kronos, a model checker by Yovine [108], only knows the combined operators $U^{\geq n}$ etc. similar to the abbreviations we have defined for ATCTL. As mentioned above, this is not a severe restriction, because we always could make do with the combined operators. However, some formulas cannot be abbreviated, for example “The time between the first and the third happening of action $a$ is at most 5 seconds” is expressed by $\forall (\neg a U a \Rightarrow (\forall (\neg a U a \forall (\neg a U a c \leq 5))))$.

Several groups have worked on ACTL model checking. De Nicola and Vaandrager [28] describe how the ACTL model checking problem can be reduced to the CTL model checking problem, namely by a translation similar to the ones we have defined above. De Nicola, Fantechi, Gnesi and Ristori [27, 35] described an implementation of this procedure using the model checker EMC [19]. Fantechi, Gnesi et al. [34] later presented a symbolic model checker for ACTL. Mateescu and Sighireanu [76] described a model checker for a subset of $\mu$-calculus. ACTL and CTL can be encoded in this subset. Another model checker for the $\mu$-calculus is TRUTH by Lange, Leucker et al. [67].

As shown above, all these model checkers can be used to verify ATCTL properties against ATCTL models via one of the reductions.

3.15 Conclusion

The goal of this chapter was to define a logic that simplifies the formal specification of desired properties of statecharts. Statecharts are a rich language; a suitable logic should
include means to express properties of states and of actions. For real-time statecharts, it should also be able to express real-time requirements.

We have defined an extension ATCTL of CTL with actions and real time, and we have shown that the resulting logic is reducible to TCTL, so that ATCTL formulas can be model checked by, e.g., Kronos. An alternative reduction to ACTL allows us to model check ATCTL formulas not including a clock reset action. However, this second reduction is much less efficient, because the generated ACTL automaton has a size exponential in the number of clocks and the largest clock constraints. By means of an example, we have shown that model checking works at least for small cases. All reductions are essentially surjective, so that it is possible to translate counterexamples generated by a model checker back to ATCTL models.

ATCTL is a logic that fits (real-time) statecharts well, because it allows the analyst to speak about states, actions and real time. Thanks to the reductions, we can check properties of simple UML models by reusing existing model checkers.

Unfortunately, there is no complete reduction to ACTL; we have decided to abandon this work because the reduction to TCTL allowed us already to use model checkers efficiently.

The logic ATCTL is mainly intended to simplify the task of writing down a requirement formally, by providing the engineer with all necessary building blocks. The task of formalising, however, could be simplified substantially by a tool that provides property patterns. A property pattern is a scheme for a standard property like "error state ... is avoided", that are found as requirements of many systems. Dwyer, Avrunin, and Corbett [29] have described a set of often used standard properties; TCM could be extended to offer these patterns and translate them automatically to the corresponding ATCTL property.
Chapter 4

Probabilistic Extension: P-Statecharts

This chapter is an updated version of [56, 57].

4.1 Introduction

Statecharts provide means to structure the behaviour (hierarchy and parallelism) and therefore allow to keep an overview of a nontrivial system. In addition, they are already known to many software engineers. However, statecharts provide no means to specify probabilistic choice.

As we have seen in chapter 1, probabilistic choice can be used to model randomised algorithms, abstraction, or unreliable systems. There exist several languages to describe these systems, for example Markov Decision Processes (MDP), but these languages are not adapted to describe larger systems.

This chapter extends statecharts with probabilistic choice as a means to support the design and verification of probabilistic systems. A transition is allowed to lead to one of several states depending on a probability distribution; each probability distribution is guarded by a trigger, similar to simple probabilistic automata in Segala and Lynch [93]. The extension addresses system randomness, for example a system that embodies a randomised algorithm.

The interference of probabilities, priorities and nondeterminism raises some subtle semantic issues. We attack these issues in a way that allows one still to employ an arbitrary priority scheme to resolve or reduce nondeterminism. The semantics is formally defined as a mapping to (strictly) alternating probabilistic transition systems [40], a subset of Markov decision processes (MDP) [86].

To allow formal verification of probabilistic temporal properties over probabilistic statecharts, properties are expressed in the probabilistic branching-time temporal logic PCTL [6], the prime logic for property specification and verification of models that exhibit both probabilities and nondeterminism. These properties can be checked using the model checker PRISM [66]. To facilitate model-checking of the underlying model, we confine ourselves to bounded integer variables and assume a closed system.

We base our probabilistic semantics as in chapter 3 on the statechart semantics of Eshuis and Wieringa. The setup of our probabilistic extension, however, is independent of the basis we take. This means that other formal statechart semantics can equally well
Chapter 4. Probabilistic Extension: P-Statecharts

be enhanced with a similar probabilistic extension, as long as certain principal conditions are respected. We give an account of these conditions in section 4.9.

We use probabilistic model checking of properties specified in the existing language PCTL. We have chosen not to extend it to something like PATCTL, because probabilistic model checkers are not yet developed as far as the model checkers used in chapter 3, so it is more important to keep the models and properties simple and efficient. In addition, a language like PATCTL would include so many modal operators that it would become unmanageable.

**Closed-system assumption.** Opposed to the statecharts in section 2.2 we assume a closed system model in the sense that we do not consider the system to be subject to inputs from the environment. Therefore, P-statecharts are not suitable to express environmental randomness straightforward. (Environmental randomness can be simulated by modelling the compound system, consisting of the system under development and its [probabilistic] environment.) Chapter 5 will explicitly include environmental randomness. This “closed-system assumption” simplifies model checking.

### 4.2 Probabilistic UML Statecharts

This section introduces probabilistic UML-statecharts (P-statecharts, for short), together with some drawing conventions. We first fix some notations. We assume familiarity with basic measure and probability theory [94]; see also appendix A.

**Collection of statecharts.** Remember that a system is described by a finite collection of communicating statecharts. In the following, we assume a fixed finite collection of P-statecharts, denoted by \( \{PSC_1, \ldots, PSC_n\} \).

**Syntax.** A single P-statechart \( PSC_i \) consists of

- A finite set \( \text{Nodes}_i \) of nodes with a tree structure, as for statecharts (see the definition in section 2.2).
- A finite set \( \text{Events}_i \) of events, as for statecharts.
- A finite set \( \text{Vars}_i \) of variables, as for the extension of statecharts described in section 2.4.1. The initial valuation \( V_{0,i} : \text{Vars}_i \rightarrow \mathbb{Z} \) defines initial values of the variables. (We will only allow bounded integer variables for model checking.)
- A set \( \text{Guards}_i \) of guard expressions, as for statecharts. In addition to guards of the form \( j.\text{isin}(x) \) (for \( j \in \{1, \ldots, n\} \) and \( x \in \text{Nodes}_j \)), we allow comparisons like \( \text{expr} \leq \text{expr} \) and \( \text{expr} \geq \text{expr} \), for arithmetic expressions made up from the variables and integer constants.
- A set \( \text{Actions}_i \) of actions, as for statecharts. In addition to actions of the form \( \text{send} \ j.e \) (for \( j \in \{1, \ldots, n\} \) and \( e \in \text{Events}_j \)), we allow assignments like \( v := \text{expr} \), where \( v \in \text{Vars}_i \). (As we assume a closed system, we forbid the action “to send an event to an external component”.)
A finite set $PEdges_i$ of P-edges, replacing the set of edges of a statechart. A P-edge is a quadruple $(X, e, g, P)$ where $X \subseteq Nodes_i$ is a non-empty set of source state nodes, $e \in Events_i \cup \{\bot\}$ is the triggering event, $g \in Guards_i$ is a guard, and $P$ indicates the possible actions and target nodes. Formally, it is a probability measure in the discrete probability space $(P(Actions_i) \times (P(Nodes_i) \setminus \{\emptyset\}), P)$.

We assume that there is a bijective index function $i: \{1, \ldots, |PEdges_i|\} \rightarrow PEdges_i$ to simplify the identification of P-edges. If $i(j) = (X_0, e_0, g_0, P_0)$, we sometimes write $i(j).X$ for $X_0$ etc.

We will denote a P-statechart by $PSC_i = (Nodes_i, Events_i, Vars_i, PEdges_i)$.

The above definition extends our former definition of statecharts by local variables (found in many statechart variants, omitted in section 2.2 for simplicity) and P-edges (new for P-statecharts). A P-edge $(X, e, g, P)$ can be considered as a hyperedge with source node set $X$, and (possibly) multiple targets $(A, Y)$, each target having a certain probability $P(A, Y)$. Note that a target is an action set $A$ together with a non-empty set $Y$ of successor nodes. Once the P-edge is triggered by event $e$ and guard $g$ holds in node $X$, a target $(A, Y)$ is selected with probability $P(A, Y)$.

**Drawing a P-Statechart.** We extend the drawing of a statechart as follows. A trivial P-edge (where probability 1 is assigned to a unique action set/node set) is drawn like an edge in a statechart: an arrow $e[g] \rightarrow A$. A P-edge possessing a non-trivial probability space consists of two parts: first an arrow with event and guard $e[g]$, that points to a symbol $\oplus$ (a so-called P-pseudonode), then several arrows emanating from the P-pseudonode, each with a probability and an action set $\frac{p}{A} \rightarrow$. This notation is inspired by C-pseudonodes $\odot$, used for case selection purposes e.g., by Harel and Gery [44].

**Example 4.2.1** Figure 4.1 depicts a P-statechart which shows the behaviour when playing with an unreliable, but fair coin: the event “toss” may or may not be ignored. If the system reacts, it generates “heads” or “tails”, each with 50% chance. If the output is heads, the system stops playing. It is unspecified how (un)reliable the system is.

In figure 4.2, a nonprobabilistic version of the unreliable coin is given. We cannot express that the coin is fair in a traditional statechart, as every choice is nondeterministic. Probabilistic statecharts are flexible enough to allow for the specification of probabilities only when they are known – nondeterminism still can be used where there is a choice and we don’t know the probability distribution.

### 4.3 P-Statechart Semantics

This section discusses the semantics of P-Statechart, which is an adaptation of the nonprobabilistic semantics in section 2.2. The semantics is defined in two phases. First, it is defined what will be a step. This encompasses the resolution of nondeterminism, probabilistic choice and priorities within a single P-statechart. Subsequently, these steps are used as the building blocks in a mapping of a collection of P-statecharts to a Markov decision process.
Intuitive semantics for a single P-statechart. The intuitive behaviour of a P-statechart can be described as follows. The statechart is always in some state (which consists of one or several nodes). A P-edge is enabled if the P-statechart is in the source node(s), the event of the edge happens and its guard holds. Then, the system chooses one of the possible targets (probabilistically and nondeterministically). The system executes as many enabled P-edges at once as possible without conflict: it leaves the source nodes, executes the chosen actions and enters the chosen target nodes of the P-edge(s).

4.3.1 Step Construction

A step is, basically, the set of edges taken simultaneously in one transition of a single P-statechart $PSC_i$. This section describes how a step is constructed.

Configurations are defined as for statecharts in section 2.2. The set of all configurations of $PSC_i$ is denoted $Conf_i$.

States are similar to plain statechart states, but here, we include variables: A state of $PSC_i$ is a triple $(C_i, I_i, V_i)$ where $C_i$ is a configuration, $I_i \subseteq Events_i$ is a set of events (to which the P-statechart still has to react), and $V_i : Vars_i \rightarrow \mathbb{Z}$ is a valuation of the variables. The set of all valuations of $PSC_i$ is denoted $Val_i$. The validity of guard $g$ in a state depends on the configurations $C_1, \ldots, C_n$ and the valuations $V_1, \ldots, V_n$. We write $(C_1, \ldots, V_1) \models g$ iff $g$ holds in the state of the collection of P-statecharts.

In a plain statechart, edges are a basic notion. We derive the edges of a P-statechart from its P-edges. An edge is a triple $(j, A, Y)$, where $j$ identifies a P-edge, $A \subseteq Actions_i$ is a set of actions and $Y \subseteq Nodes_i$ is a set of target nodes. The set $Edges_i$ is defined as:

$$\{ (j, A, Y) \mid \exists X, e, g, P : i_1(j) = (X, e, g, P) \in PEdges_i \land P(\{(A, Y)\}) > 0 \}$$

A P-edge $(X, e, g, P)$ is enabled if the current configuration $C_i$ contains its source state nodes $X$, the event $e$ is in the current input set $I_i$ and the guard $g$ holds: $(C_1, \ldots, V_1) \models g$. 
4.3. P-Statechart Semantics

We denote the set of enabled P-edges by $EnP(C_i, I_i, V_i)$. An edge is enabled if its corresponding P-edge is enabled.

The scope of an edge $(j, A, Y)$ is the smallest (in the parent–child hierarchy) OR-node that contains both the source nodes $i(j).X$ and the target nodes $Y$.

A step is a set of edges that are taken together as a reaction to events. The edges in a step for P-statechart $PSC_i$ depend on its current state $(C_i, I_i, V_i)$ (and, for the guards, on the configurations and valuations of the other statecharts in the collection). A step has to obey the same principal constraints as a step in a plain statechart:

**Enabledness.** All edges in the step must be enabled.\(^1\)

**Consistency.** All edges in the step must be pairwise consistent.

**Priority.** We assume a given priority scheme (a partial order on the edges) that resolves some of the inconsistencies: If an enabled edge $e$ is not in the step, then there must be an edge in the step that is inconsistent with $e$ and does not have lower priority than $e$.

**Maximality.** A step must be maximal. This means that adding any edge leads to a violation of the above conditions.

We now give an algorithm to construct a step of a single P-statechart which – by construction – satisfies the conditions above. The algorithm employs a specific order with respect to the resolution of nondeterminism and probabilities. Assume that the current state is $(C_i, I_i, V_i)$.

**Algorithm 4.3.1 Step Construction Algorithm.**

1. Calculate the set of enabled P-edges: for $j \in \{1, \ldots, |PEdges_i|\}$,
   $$j \in EnP(C_i, I_i, V_i) \text{ iff } i(j).X \subseteq C_i \land (i(j).e \in I_i \cup \{\perp\}) \land (C_{1\ldots n}, V_{1\ldots n}) \vDash i(j)g$$

2. Draw samples from the probability spaces of the enabled P-edges, reducing the set $EnP(C_i, I_i, V_i)$ to a set $En(C_i, I_i, V_i)$ of enabled edges.

3. Calculate $Steps(En(C_i, I_i, V_i))$, where $Steps(E)$ (for $E \subseteq Edges_i$) contains all maximal, prioritized, consistent sets of edges $\subseteq E$.

4. Choose nondeterministically an element of $Steps(En(C_i, I_i, V_i))$.

Task 2 can be formalised as follows: For every state $(C_i, I_i, V_i)$, we define a discrete probability space $\mathcal{P}R_{(C_i, I_i, V_i)} = (\mathbb{P}(Edges_i), \mu)$. The probability measure $\mu : \mathbb{P}(\mathbb{P}(Edges_i)) \rightarrow [0, 1]$ of the above probability space is defined as follows. We assume that for every P-edge $j \in EnP(C_i, I_i, V_i)$, an independent choice of a possible target $(A_j, Y_j)$ is made. Because the choices are made independently, the total probability of a combination of choices is the product of the probabilities of the single choices. Therefore,

$$\mu(\{(j, A_j, Y_j) \mid j \in EnP(C_i, I_i, V_i)\}) = \prod_{j \in EnP(C_i, I_i, V_i)} (i(j).P)(\{(A_j, Y_j)\})$$

\(^1\)It may happen that no P-edge, and consequently no edge, is enabled. Because of the closed world assumption, this is a deadlock.
Note that if $EnP(C_i, I_i, V_i) = \varnothing$, then $\mu$ is the trivial probability measure such that $\mu(\{\varnothing\}) = 1$.

Tasks 3 and 4 can be achieved by applying the original algorithm for nextstep (algorithm 2.3.1) to the calculated set $En(C_i, I_i, V_i)$. As nextstep is a nondeterministic algorithm that, given a set of enabled edges, calculates a maximal, prioritized, consistent step. The above algorithm (consisting of Tasks 1–4) leads to a step that is enabled, consistent, prioritized and maximal.

4.3.2 Order of Probabilistic and Nondeterministic Choices in Algorithm 4.3.1

It is worth to highlight that (after calculating the enabled possibilities in Task 1), we first choose probabilistically (in Task 2) according to the probabilities given by the P-edges. Only after that, in Tasks 3 and 4, we resolve the nondeterminism between the remaining possibilities. This order – first resolve probabilism, then nondeterminism – is essential, as shown by the following two examples, and can only be reversed by restricting the expressivity of P-statecharts.

Priority depends on probabilistic choices. In several priority schemes for statecharts, priority depends on the scope [44]. For example, in the STATEMATE priority scheme, smaller (in the parent–child-hierarchy) scopes have lower priority [46]. The P-statechart in figure 4.3 describes a fragment of a system with two printers, of which the preferred printer (printer 1) is available only with probability $\frac{1}{2}$, and the other (printer 2) is available in $\frac{3}{4}$ of the cases. The probabilities are independent. If a print request is started, the system directs it to the best available printer. The edge leading from $A$: Ready to print to $C$: Printing on printer 2 (denoted $A \rightarrow C$) with scope $G$: Not preferred has higher priority than $A \rightarrow D$ and $A \rightarrow E$ (with scopes $F$: Not printing, but $A \rightarrow C$ has lower priority than the edge $A \rightarrow B$ (with scope root). So, if in configuration $\{A, F, G, root\}$ event start happens, the step produced by the above algorithm is:

- $\{A \rightarrow B\}$ with probability $\mu(\{A \rightarrow B, A \rightarrow C\}) + \mu(\{A \rightarrow B, A \rightarrow D\}) = \frac{1}{2} \cdot \frac{3}{4} + \frac{1}{2} \cdot \frac{1}{4} = \frac{1}{2}$, as edge $A \rightarrow B$ has priority over all other edges.
- $\{A \rightarrow C\}$ with probability $\mu(\{A \rightarrow E, A \rightarrow C\}) = \frac{1}{2} \cdot \frac{3}{4} = \frac{3}{8}$, as $A \rightarrow C$ has priority over $A \rightarrow E$. 

Figure 4.3: Example of priority depending on target state
4.3. P-Statechart Semantics

As the edges that belong to one P-edge have different scopes, it is impossible to resolve the priorities prior to resolving the probabilistic choice. Although this is exemplified using the priority scheme of Statemate, a similar P-statechart can be drawn for other priority schemes which also depend on the scope, for example the scheme in Harel and Gery [44]. The UML priority scheme [83] does not depend on the scope, but only on the source nodes. For such a priority scheme, the above phenomenon is not a problem; but the next example is independent of priorities.

Consistency depends on probabilistic choices. The P-statechart in figure 4.4 shows a system which reacts to an event $e$ in two independent components, of which one causes an error with probability $\frac{1}{4}$. The edge $A \rightarrow Error$ is inconsistent with $B \rightarrow D$, as the scopes root and $G$ are not orthogonal, i.e., they are not (descendants of) different children of an AND-node; but $A \rightarrow C$ is consistent with $B \rightarrow D$. So, if in configuration $\{A, B, F, G, H, root\}$ event $e$ happens, the step produced by the above algorithm is:

- $\{A \rightarrow C, B \rightarrow D\}$ with probability $\mu(\{A \rightarrow C\}) = \frac{3}{4}$.
- Either $\{B \rightarrow D\}$ or $\{A \rightarrow Error\}$ with probability $\mu(\{A \rightarrow Error\}) = \frac{1}{4}$; scope-dependent priority schemes only allow one of the two cases.

Thus, the probability of taking edge $B \rightarrow D$ as a reaction to event $e$ depends on the resolution of the probabilistic choice in the parallel node $F$. It is impossible to resolve the nondeterminism first, as there may or may not be inconsistent edges.

In summary, the influence of the scope in the construction of a step, as present in both the consistency definition and the priority scheme, forces us to resolve probabilism prior to establishing consistency and priority.

Changing the order anyway. If we restrict P-statecharts in two points, we may change the order of resolution:

1. All edges (with positive probability) belonging to one P-edge have the same scope.
2. All edges (with positive probability) belonging to one P-edge have the same priority. This can be achieved, for example, by choosing a priority scheme that only depends on the scope [44], together with condition 1, or on the source state [83].
Consistency and many priority schemes only depend on the scope. Therefore, with these restrictions, we could change algorithm 4.3.1 such that it first resolves nondeterminism and then probabilism. This is exactly what we will do in chapter 5.

### 4.3.3 Step Execution

The execution of a step is similar to the execution of a plain statechart step, as all probabilistic aspects have been resolved during step construction. However, we have included variables here, so we extend some of the definitions given in section 2.2.

On the level of a single statechart, executing a step consists of two parts: updating the variables and events occurring in the actions and determining the new state.

The definition of default completion is the same as for plain statecharts.

**Executing a step.** Given configurations \((C_1, \ldots, C_n)\), steps \((T_1, \ldots, T_n)\) and valuations \((V_1, \ldots, V_n)\), we define the new state \((C'_1, I'_1, V'_1)\) for P-statechart \(PSC_i\) by:

- \(C'_i\) is the default completion of the union of \(\bigcup_{(j,A,Y) \in T_i} Y\) (all target nodes entered) and \(\{x \in C_i \mid \forall (j,A,Y) \in T_i : x \text{ is not a descendant of } \text{scope}(i, j, X, Y)\}\)

- \(I'_i = \bigcup_{k=1}^n \{ e \mid \exists (j, A, Y) \in T_k : \text{send } i.e \in A \}\)

- \(V'_i = V_i[\forall v := \text{expr} \mid \exists (j, A, Y) \in T_i : v := \text{expr} \in A]\), the same valuation as before except for the assignments in any action of the step. If these assignments are inconsistent, pick (nondeterministically) any of them.

We denote this as: \(\text{Execute}(C_1, \ldots, C_n, T_1, \ldots, T_n, V_1, \ldots, V_n) = ((C'_1, I'_1, V'_1), \ldots, (C'_n, I'_n, V'_n))\).

### 4.3.4 BPTS Semantics

Recall that in the step construction algorithm we resolve probabilistic choices prior to resolving non-determinism. A semantic model – that contains both non-determinism and discrete probabilities – preferably obeys the same order. Bundle probabilistic transition systems (BPTS) [23] is one of the rare models for which this is indeed the case. In a BPTS transition, first a “bundle” or set of possible next states is chosen probabilistically, and then one element is chosen nondeterministically.

**Bundle probabilistic transition systems.** A BTPS is a quintuple \((\Sigma, \mathcal{T}, AP, \mathcal{I}, \sigma_0)\) where:

- \(\Sigma\) is a finite, non-empty set of states.

- The transition relation \(\mathcal{T} : \Sigma \rightarrow \text{Prob}\ (\mathcal{P}(\Sigma))\) assigns to each state a probability space over \(\mathcal{P}(\Sigma)\).

- \(AP\) is a set of atomic propositions.

- \(\mathcal{I} : \Sigma \rightarrow \mathcal{P}(AP)\) is the interpretation of the atomic propositions.
4.3. P-Statechart Semantics

A

C

B

D

Figure 4.5: Example of non-disjoint transitions

- $\sigma_0 \in \Sigma$ is the initial state.

Our definition differs from the original one as follows: D’Argenio et al. [23] allow $\Sigma$ to be infinite; they include indices to distinguish individual transitions and a set of actions; and they omit the state-labelling $I$ and the initial state $\sigma_0$. In addition, they require that the transitions are disjoint. We do not constrain the transitions because we see no reason to forbid P-statecharts like the one in figure 4.5, where state B can be reached by multiple transitions.

**Informal BPTS behaviour.** A BPTS is always in some state $\sigma$. When it takes a transition, it first chooses probabilistically one element of $T(\sigma)$. This is the set of possible next states. Then, the next state is chosen nondeterministically out of this set.

**Example 4.3.1** Figure 4.6 shows half of the board for an almost finished backgammon game [62]. Assume it is white’s turn. The player first throws two dice and then moves two checkers (symbolised by white circles) to a point with a lower number. The cast of the dice indicates how many points the checker can be moved. If both dice show the same number, the player plays twice (so, he may make four moves in total). A player wins when he bears all his checkers off the board, i.e., he moves them to the imaginary point 0 to the right of the board. If the player cannot move nor bear off a checker normally, he is allowed to bear off one checker from the highest point to a “negative” point.

We can model this by a BPTS: first, there is a probabilistic choice (between the possible outcomes of the dice), and then, the player chooses nondeterministically which checkers to move. Figure 4.7 shows the BPTS that models the legal moves in this situation.

**BPTS semantics of a collection of P-statecharts.** For a finite collection of P-statecharts (indexed from 1 to $n$), the BPTS $(\Sigma, T, AP, I, \sigma_0)$ is defined by:

- $\Sigma = \times_{i=1}^{n} (Conf_i \times P (Events_i) \times Val_i)$.

- Given a state $\sigma = ((C_1, I_1, V_1), \ldots, (C_n, I_n, V_n))$, let $T(\sigma) = (P(\Sigma), \mu)$ where $\mu$ is the (discrete) probability measure defined as follows. We assume that for every P-statechart $PSC_i$ in the collection, a set of edges $E_i \subseteq Edges_i$ is chosen independently. The total probability, then, is the product of the probabilities of the single choices.
Chapter 4. Probabilistic Extension: P-Statecharts

Figure 4.6: An almost finished backgammon game

Figure 4.7: A BPTS showing the legal moves for white in the situation of figure 4.6. States are drawn as circles; in most states, inscribed numbers indicate where a white checker is placed. The probability that in the initial state 6, 3 a single line is chosen is \( \frac{1}{36} \); the probability that a double line is chosen is \( \frac{2}{36} \). Arrows indicate nondeterministic choices.
Therefore,

\[
\mu(\{\{\rho \in \Sigma \mid \exists T_1 \in \text{Steps}(E_1), \ldots, T_n \in \text{Steps}(E_n) : \rho = \text{Execute}(C_{1 \ldots n}, T_{1 \ldots n}, V_{1 \ldots n})\}\}) = \prod_{i=1}^{n} \mu_i(\{E_i\})
\]

where \(\mu_i\) is the probability measure of \(\mathcal{PR}(C_i, I_i, V_i) = (P(\text{Edges}_i), \mu_i)\) mentioned in the explanation of algorithm 4.3.1. Of course, if an \(E_i\) is not a valid step, then \(\mu_i(\{E_i\}) = 0\). For other sets \(S \subseteq \Sigma\), let \(\mu(\{S\}) = 0\).

- The set of atomic propositions is \(AP = \bigcup_{i=1}^{n}\{i.\text{isin}(x) \mid x \in \text{Nodes}_i\}\).
- The interpretation of the atomic propositions is defined by the rule:

\[
x \in C_i \text{ for some } i \in \{1, \ldots, n\} \implies i.\text{isin}(x) \in I((C_1, I_1, V_1), \ldots, (C_n, I_n, V_n))
\]

- \(\sigma_0 = ((C_{0,1}, \emptyset, V_{0,1}), \ldots, (C_{0,n}, \emptyset, V_{0,n}))\), where \(C_{0,i}\) is the default completion of \(\{\text{root}_i\}\) and \(V_{0,i}\) is the initial valuation in the \(i\)th P-statechart.

### 4.3.5 Markov Decision Process Semantics

Although a BPTS model would be appropriate as semantical model, we also give a semantics in terms of the (more standard) model of Markov decision processes (MDP) [86]. This slightly complicates the semantics, but has the nice property that it allows model checking of probabilistic properties.

To allow the interpretation of temporal logic formulas later on, we equip an MDP with a state-labelling that assigns a set of atomic propositions to states.

**Markov decision processes.** An MDP is a quintuple \((S, Distr, AP, I, s_0)\) where:

- \(S\) is a finite, non-empty set of states.
- \(Distr\) assigns to each state a finite, non-empty set of distributions\(^2\) on \(S\).
- \(AP\) is a set of atomic propositions.
- \(I : S \rightarrow \mathbb{P}(AP)\) is the interpretation of the atomic propositions.
- \(\sigma_0 \in S\) is the initial state.

Informally speaking, an MDP exhibits the following behaviour: Whenever the system is in state \(\sigma\), a probability distribution \(\mu \in Distr(\sigma)\) is chosen nondeterministically. Then, the system chooses probabilistically the next state according to the selected distribution \(\mu\).

\(^2\)Probability textbooks [94] call this a probability weight, but “distribution” is found in the MDP literature.
Chapter 4. Probabilistic Extension: P-Statecharts

Example 4.3.2 The MDP in figure 4.8 models a simple gambling machine with two reels. In most states, the figure shows the state of the first reel atop the state of the second reel. The reels, once stopped, show apples (with probability $\frac{3}{4}$) or cherries (with probability $\frac{1}{4}$). They do not stop in some specific order; so, there is a nondeterministic choice in state $\text{turning}/\text{turning}$. When both reels show cherries, the jackpot state is reached; in the combination apple/cherry, there is some reward.

Steps and runs in an MDP. A step is an expression of the form $\sigma \xrightarrow{\mu} \sigma'$ where $\mu \in \text{Distr}(\sigma)$ and $\mu(\sigma') > 0$.

A run is a sequence of steps, where the target of each step is the source of the next one. We write this as: $\sigma_1 \xrightarrow{\mu_1} \sigma_2 \xrightarrow{\mu_2} \sigma_3 \rightarrow \cdots$. A complete run is a run that cannot be extended, either because it is infinite or there is no step starting in its final state.

An infinite run $\sigma_1 \xrightarrow{\mu_1} \sigma_2 \rightarrow \cdots$ is fair if for every state $\sigma$ that is visited infinitely often, each element of $\text{Distr}(\sigma)$ is also chosen infinitely often. (We will need fairness in section 4.5.)

MDP semantics of a collection of P-statecharts. In an MDP, first a non-deterministic choice is made (among the available distributions) after which a next state is selected probabilistically. This order is reversed for the construction of a step of a P-statechart. To overcome this difference, we add auxiliary states to the MDP. Recall that (original) states consist of, per P-statechart, a configuration, a set of events, and a valuation, written $(C, I, V)$. Auxiliary states will correspond to the outcome of Task 2 of the step construction algorithm and consist of, per P-statechart, a configuration, a set of enabled edges and a valuation, written $(C, E, V)$. Each auxiliary state will be labelled with the distinguished atomic proposition $\Delta$. It offers a non-deterministic choice of trivial distributions (assign-
4.3. P-Statechart Semantics

ing probability 1 to a single state) only, such that each successor state is an original state (not labelled $\triangle$). Original states, in turn, do only possess singleton sets of probability distributions (hence there is no non-determinism), and all states with positive probability will be states labelled $\triangle$. This type of MDPs is also known as (strictly) alternating probabilistic transition systems [40]. In an alternating probabilistic transition system, each state either allows only probabilistic choice or only nondeterministic choice; therefore, it could be regarded as a (special case of) MDP as well as a (special case of) BPTS.

For a finite collection of P-statecharts (indexed from 1 to $n$), the MDP $(S, Distr, AP, I, s_0)$ is defined by:

- The set of states is the disjoint union of the original and auxiliary states, $S = O \cup A$, where

$$O = \times_{i=1}^{n} (Conf_i \times P(Events_i) \times Val_i)$$

$$A = \times_{i=1}^{n} (Conf_i \times P(Edges_i) \times Val_i).$$

- The set of distributions assigned to state $(s_1, \ldots, s_n)$ is defined by the rules:

$$\mu((C_1, E'_1, V_1), \ldots, (C_n, E'_n, V_n)) = \prod_{i=1}^{n} P_i(\{E'_i\})$$

$$\text{and } \mu(s) = 0 \text{ otherwise.}$$

- The interpretation of the atomic propositions is:

$$I((s_1, \ldots, s_n)) = \begin{cases} \bigcup_{i=1}^{n} \{i.isin(x) \mid x \in Nodes_i\} & \text{if } (s_1, \ldots, s_n) \in O \\ \{\triangle\} & \text{otherwise} \end{cases}$$

- $s_0 = ((C_{0,1}, \emptyset, V_{0,1}), \ldots, (C_{0,n}, \emptyset, V_{0,n})) \in O$, where $C_{0,i}$ is the default completion of $\{\text{root}_i\}$ and $V_{0,i}$ is the initial valuation in the $i$th P-statechart.

**Example 4.3.3** To illustrate how P-statecharts are mapped to MDPs, we consider the “unreliable, but fair coin” P-statechart from figure 4.1. We compose this P-statechart (numbered 1) with an event generator (numbered 2), which generates toss events nondeterministically; see figure 4.9. (If we don’t add a component which generates toss events, the only reachable state of the system would be the initial state, due to the closed system...
Chapter 4. Probabilistic Extension: P-Statecharts

Figure 4.9: P-statechart that generates toss events nondeterministically

Figure 4.10: MDP semantics of the P-statecharts in figures 4.1 and 4.9. The inscriptions are the configuration and the set of input events for figure 4.1; the data for figure 4.9 is understood.

4.4 Comparison to Non-Probabilistic Semantics

In this section, we compare our P-statechart semantics to the semantics of the corresponding non-probabilistic statechart, where probabilistic choices are replaced by nondeterminism. For a definition of statecharts as we use them, see section 2.2.

Projections. To facilitate the comparison, we define two projections: one that abstracts from the probabilistic choices in a P-statechart (called $a_1$), and one that abstracts from the probabilistic choices in an MDP (called $a_2$). The projections replace probabilities by nondeterminism. Later on, we will define a third projection $\pi$ to remove auxiliary variables introduced by $a_1$. 
4.4. Comparison to Non-Probabilistic Semantics

Further, we define the abbreviation

\[ \alpha_1(\{PSC_1, \ldots, PSC_n\}) = \{\alpha_1(PSC_1), \ldots, \alpha_1(PSC_n)\} \]

**Definition of \( \alpha_1 \).** A naïve definition of \( \alpha_1 \) would just be \( \alpha_1(PSC_i) = (\text{Nodes}_i, \text{Events}_i, \text{Vars}_i, \text{Edges}_i) \). However, in the case that some edge has a higher priority than another edge which belongs to the same P-edge, the latter is never taken. For example, the P-statechart of figure 4.11 contains edges of this kind, if we choose the Statemate priority scheme \cite{46}. The edge \( A \to C \) has higher priority than \( A \to B \). So, in the naïve translation, \( A \to B \) would never be taken.

To solve this problem, we refine the translation as follows: For every nontrivial P-edge where edges have different priorities, we add one variable to make a nondeterministic choice between the possible continuations of the P-edge. The actual choice is then moved to a place where it does not interfere with priorities: in every transition, the variables are assigned a value that is chosen nondeterministically. When (the translation of) a nontrivial P-edge is taken, the statechart chooses according to the last assigned value. (This leads to some unnecessary assignments: a variable is also assigned if the edge is not going to be enabled, but this translation is easier to define than the “minimal” one.)

So, \( \text{Edges}_i \) has the form

\[ \{ (X, e, g \land \ldots, A \lor \ldots, Y) \mid \exists d : \exists P : t_i(d) = (X, e, g, P) \land (d, A, Y) \in \text{Edges}_i \} \]

where the \( \ldots \) stand for checks of and assignments to new variables, where necessary. We also add an extra state \( \text{Init} \) to initialize the variables. As a consequence, at most one of the edges corresponding to a P-edge is enabled.

1. For each P-edge \( t_i(d) = (X, e, g, P) \in \text{Edges}_i \), define \( T_{i,d} := \{(d, A, Y) \mid P(A, Y) > 0\} \subseteq \text{Edges}_i \). This set is finite.

2. If \( T_{i,d} \) contains two edges that have different priority, choose a bijective index function \( t_{i,d} : \{1, \ldots, |T_{i,d}|\} \rightarrow T_{i,d} \).

   Without loss of generality, we assume that the P-edges \( t_i(1), \ldots, t_i(\delta_i) \) are the ones with edges of different priority (for a suitable \( \delta_i \in \{0, \ldots, |\text{Edges}_i|\} \)).

3. Now, translate the P-statechart \( PSC_i \) as follows to a statechart \( \alpha_1(PSC_i) = (\text{Nodes}_i, \text{Events}_i, \text{Vars}_i, \text{Edges}_i) \):
Chapter 4. Probabilistic Extension: P-Statecharts

Definition of a statechart.

Example 4.4.1

KS automaton

We overload the projection operator

where

is defined by

The set \( \text{Events}_i \) is unchanged.

\( \text{Vars}_i = \text{Vars}_i \cup \{ v_1, v_2, \ldots, v_k \} \), where the \( v_i \) are new variables, i.e., \( v_i \notin \text{Vars}_i \).

The sets \( \text{ Guards}_i \) and \( \text{Actions}_i \) of the translated statecharts are defined as normal and therefore include checks of and assignments to the new variables.

Edges are defined by the following rules.

\[ t_i(d) = (X, e, g, P) \in \text{PEdges}_i \quad d > \delta_i \]

\[ (d, A, Y) \in \text{T}_i,d \quad k_j \in \text{dom} \ t_{i,d}, \text{ for all } j \]

\[ (X, e, g, A \cup \{ v_j := k_j \mid j = 1, \ldots, \delta \}, Y) \in \text{Edges}_i \]

\[ t_i(d) = (X, e, g, P) \in \text{PEdges}_i \quad d \leq \delta_i \]

\[ t_{i,d}(k) = (d, A, Y) \in \text{T}_i,d \quad k_j \in \text{dom} \ t_{i,d}, \text{ for all } j \]

\[ (X, e, g \land v_d = k, A \cup \{ v_j := k_j \mid j = 1, \ldots, \delta \}, Y) \in \text{Edges}_i \]

\[ k_j \in \text{dom} \ t_{i,d}, \text{ for all } j \]

\[ (\{ \text{Init} \}, \bot, true, \{ v_j := k_j \mid j = 1, \ldots, \delta \}, \{ \text{default}_i(\text{root}_i) \}) \in \text{Edges}_i \]

The second rule accounts for edges of different priority belonging to the same P-edge. Therefore, it includes an extra guard \( v_d = k \). The third rule defines edges from the new initial state to the original one and initialises the variables.

Example 4.4.1

Figure 4.12 illustrates \( a_1 \): it contains a possible translation of figure 4.11 to a statechart.

Definition of \( \pi \). Given a finite collection of P-statecharts \( \{ \text{PSC}_1, \ldots, \text{PSC}_n \} \), we define the projection \( \pi \) from \( \text{KS}'_2 = \text{sem}_2(a_1(\text{PSC}_1, \ldots, \text{PSC}_n)) = (S', \Delta', \text{AP}', \text{I}', s_0) \) to the output automaton \( \text{KS}_2 = (S', \Delta', \text{AP}', \text{I}', s_0) \):

- We overload the projection operator \( \pi \) with a projection of states. \( \pi : S \setminus \{ s_0 \} \to S' \) is defined by

\[ s = ((C_1, I_1, V_1), \ldots, (C_n, I_n, V_n)) \in S \]

\[ \pi(s) = ((C'_1, I'_1, V'_1), \ldots, (C'_n, I'_n, V'_n)) \in S' \]

where \( V'_i = V_i \mid \text{Vars}_i \setminus \{ v_1, \ldots, v_k \} \). Note that \( \pi(s_0) \) is undefined.
4.4. Comparison to Non-Probabilistic Semantics

\( (s_1, s_2) \in \Delta \quad s_1 \neq s_0 \quad (\pi(s_1), \pi(s_2)) \in \Delta' \)

\( AP' = AP \).

\( \mathcal{I}'(\pi(s)) = \mathcal{I}(s) \), for all \( s \). Note that \( \mathcal{I}' \) is well-defined because the truth values of atomic propositions do not depend on the variables.

\( s'_0 = \pi(s) \), for any \( s \) such that \( (s_0, s) \in T \). Note that \( s'_0 \) is well-defined because \( s_0 \) is (the image of) the pre-initial state \( \text{Init} \) added by \( \alpha_1 \); all transitions from \( s_0 \) lead to states that only differ in the values of \( v_1, \ldots, v_\delta \).

Note that although \( \pi \) identifies some states, it does not add any new behaviour; for every path in \( \pi(\text{sem}_2(\alpha_1(S))) \), there is a corresponding path in \( \text{sem}_2(\alpha_1(S)) \).

**Definition of \( \alpha_2 \).** The projection \( \alpha_2 \) maps an MDP to an output automaton and is defined by: \( \alpha_2(\text{S}, \text{Distr}, AP, \mathcal{I}, s_0) = (\text{S}', \Delta, AP, \mathcal{I} | \mathcal{S}', s_0) \), where:

- \( \text{S}' = \{(C_1, I_1, V_1), \ldots, (C_n, I_n, V_n)\} \in \text{S} \mid \forall i : I_i \subseteq \text{Events}_i \}
- \( \Delta = \{(s, s') \in \text{S} \times \text{S} \mid \exists \mu \in \text{Distr}(s) : (\exists \mathcal{S} \in \text{S} : \mu(\mathcal{S}) > 0 \land \exists \mu' \in \text{Distr}(\mathcal{S}) : \mu'(s') > 0)\} \)

State set \( \text{S}' \) contains all non-auxiliary states, and \( (s, s') \in \Delta \) whenever \( s \) can move to \( s' \) via some auxiliary state \( \mathcal{S} \) in the original MDP with positive probability.

**Theorem 4.4.1** The following diagram commutes up to ismorphism (in the sense that the mentioned structures have isomorphic sets of complete runs):

\[
\begin{array}{ccc}
\{\text{PSC}_1, \ldots, \text{PSC}_n\} & \xrightarrow{\alpha_1} & \{\text{SC}_1, \ldots, \text{SC}_n\} \\
\text{sem}_1 \downarrow & & \pi \circ \text{sem}_2 \downarrow \\
\text{MDP} & \xrightarrow{\alpha_2} & \text{loa}
\end{array}
\]

where \( \text{sem}_1 \) denotes our MDP-semantics of P-statecharts and \( \text{sem}_2 \) denotes the input-output-automaton semantics of section 2.2.

**Proof:** Assume given a finite collection of P-statecharts \( \mathcal{C} = \{\text{PSC}_1, \ldots, \text{PSC}_n\} \) with \( \text{PSC}_i = (\text{Nodes}_i, \text{Events}_i, \text{Vars}_i, \text{PEdges}_i) \). We denote \( \text{loa}_1 = (\text{S}_1, \Delta_1, AP_1, \mathcal{I}_1, s_{0,1}) = \alpha_2(\text{sem}_1(\mathcal{C})) \) and \( \text{loa}_2 = (\text{S}_2, \Delta_2, AP_2, \mathcal{I}_2, s_{0,2}) = \pi(\text{sem}_2(\alpha_1(\mathcal{C}))) \). We have to prove: \( \text{loa}_1 \) is isomorphic to \( \text{loa}_2 \).

- The sets of states are the same. The set of MDP states according to \( \text{sem}_1(\mathcal{C}) \) is \( \times_{i=1}^{n} \text{Conf}_i \times \text{Events}_i \times \text{Val}_i \cup \times \ldots \). Consequently, \( S_1 \) is the first part of this union: \( S_1 = \times_{i=1}^{n} \text{Conf}_i \times \text{Events}_i \times \text{Val}_i \).

\( S_2 \) contains all non-intermediary states of the statecharts \( \alpha_1(\mathcal{C}) \) (without the extra variables). These are exactly the states in \( S_1 \).
In each state, the transitions are the same. The central point of this proof is: The sets of steps in both semantics correspond to each other.

Assume given a state \( s_i = (C_i, I_i, V_i) \) of one component of \( C \) (according to \( \text{sem}_1 \)) and a step \( T_i \) with positive probability. We have to prove that there is a corresponding step \( T'_i \) in a corresponding state \( s'_i = (C_i, I_i, V'_i) \) of \( \text{sem}_2(\alpha_1(C)) \).

\( V'_i \) is defined by: \( V'_i(v_d) = k \) if \( i_{i,d}(k) \in T_i \); \( V'_i(v) = V_i(v) \) if \( v \in \text{Vars}_i \); choose any value for \( V'_i(v_d) \) otherwise.

We have some freedom of choice in the target of \( T'_i \): the step \( T_i \) does not prescribe any assignments to the new variables \( v_1, \ldots, v_{\delta_i} \). If one considers a path, one would choose the target that enables the next step in the path. For this proof, we assume given an assignment \( a : v_j \mapsto a(v_j) \) that describes the desired new values.

- Given \( (d, A, Y) \in T_i \), where \( d > \delta_i \), \( T'_i \) contains the edge \((i_i(d).X, i_i(d).e, i_i(d).g, A', Y)\), where \( A' = A \cup \{ v_j := a(v_j) \} \).

- Given \( (d, A, Y) \in T_i \), where \( d \leq \delta_i \), \( T'_i \) contains the edge \((i_i(d).X, i_i(d).e, g', A', Y)\), where \( g' = i_i(d).g \wedge v_d = i_{i,d}^{-1}(d, A, Y) \) and \( A' \) is defined as above.

This set is a step: it contains only edges enabled in \( s'_i \) and it fulfills the other step properties because \( T_i \) is a step.

Conversely, when given a state \( s'_i = (C_i, I_i, V'_i) \) of one component of \( S \) (according to \( \text{sem}_2(\alpha_1(S)) \)) and a step \( T'_i \), we have to prove that there is a corresponding step \( T_i \) in a corresponding state \( s_i = (C_i, I_i, V_i) \) of \( \text{sem}_1(S) \). This is exactly the inverse operation of the above: restrict \( V'_i \) to \( V_i \) and define \( T_i \) by deleting references to the new variables \( v_j \) from \( T'_i \).

The complete proof that the transitions are the same can easily be derived from the above.

- The two sets of atomic propositions are equal. Trivial.
- In each state, the same propositions hold in both translations. Trivial.
- The two translations have the same initial state. Easy.

Note that we only prove trace equivalence, not bisimulation. A bisimulation would be a relation between the states of \( \text{loa}_1 \) and \( \text{loa}_2 \) such that in related states, there are similar nondeterministic choices. However, the translation from a P-statechart to a statechart in \( \text{loa}_1 \) just replaces probabilistic by nondeterministic choice, while in \( \text{loa}_2 \), part of the nondeterministic choices is moved to a place where it does not affect the priorities.

**Corollary 4.4.1** The P-statechart semantics is a conservative extension of the statechart semantics of [33].
4.5 Property Specification for P-Statecharts

As a property specification language for P-statecharts we propose to use the probabilistic branching time logic PCTL, which extends CTL with probabilistic features. PCTL was originally interpreted over fully probabilistic systems, i.e., systems that do not exhibit any non-determinism [41]. We use the interpretation of PCTL over MDPs defined by Baier and Kwiatkowska [6, 66],3 similar to pCTL and pCTL+. MDPs are defined in section 4.3.5 of this thesis. PCTL allows one to express properties such as

\((\forall) \quad \text{"The probability that a system crashes within 13 steps without ever visiting certain states is at most } 10^{-5}.\"

In order to decide these properties, the non-determinism is resolved by means of schedulers (also known as adversaries [6] or policies [86]). Temporal formulas are then interpreted with respect to all schedulers or some schedulers. Here, we restrict ourselves to the fragment of PCTL for which actual model-checking tool-support is available; i.e., we only consider path properties interpreted for all fair schedulers. Formulas of the form “There is a fair scheduler such that . . .” can be checked via duality (see later on). The model-checking algorithm thus returns “true” for property \((\forall)\), iff \((\forall)\) holds for all fair schedulers that resolve the non-determinism in the MDP. For simplicity, we use the PCTL variant without next operator defined by Hansson and Jonsson [41].4

Although it would be possible to extend ATCTL further to something like PATCTL, we have decided to use plain PCTL for two reasons: on one hand, probabilistic model checking is not yet developed as far as the model checkers used in chapter 3, so it is more important to keep the models small and efficient. On the other hand, a logic PATCTL would include so many additional operators that it would become unmanageable.

Syntax and informal semantics. The syntax of PCTL is given by the following grammar, where \(a\) denotes an atomic proposition, \(v\) denotes a variable symbol, \(k \in \mathbb{N}\) is a natural number, \(p \in [0, 1]\) denotes a probability, and \(\sqsubseteq\) is a placeholder for a comparison operator \(<, \leq, =, \geq, >:\n
\[\varphi, \psi ::= \text{true} \mid \text{false} \mid a \mid v \leq k \mid v \geq k \mid v < k \mid v = k \mid \neg \varphi \mid P_{\varphi}[\varphi \sqsubseteq k] \mid P_{\varphi}[\varphi \sqsubseteq k] \mid P_{\varphi}[\varphi \sqsubseteq k] \mid P_{\varphi}[\varphi \sqsubseteq k]\]

The meaning of \text{true}, comparisons, conjunction and negation is standard. Recall from section 4.3.5 that atomic propositions are \(\Delta\) and \(i.isin(x)\), which holds in states where P-statechart \(i\) is in node \(x\).

Formula \(P_{\varphi}[\varphi \sqsubseteq k \psi]\) holds in a state if the probability of the set of paths that reach a \(\psi\)-state in at most \(k\) steps while passing only through \(\varphi\)-states is \(\sqsubseteq p\). \(P_{\varphi}[\varphi \sqsubseteq k \psi]\) has the same meaning, but does not put a bound on the number of steps needed to reach the \(\psi\)-state. A formal interpretation on MDPs has been defined by Baier and Kwiatkowska [6] and is omitted here.

The abbreviation \(\Diamond\) can be defined e.g., as \(P_{\varphi}[\Diamond \leq k \varphi] = P_{\varphi}[^{\text{true}}_\varphi \sqsubseteq k \varphi]\).

---

3Baier and Kwiatkowska sometimes call the logic PBTL.
4This is done because in our MDP semantics, a single step is translated to two consecutive steps.
Example 4.5.1 Figure 4.13 shows an example of a simple MDP, adapted from Baier and Kwiatkowska [6]. In state start, there is a nondeterministic choice between two transitions; in \( \mu \), there is a nontrivial probabilistic choice afterwards.

Property \( \Psi \) can be expressed as

\[
\mathcal{P}_{\leq 10^{-5}} \left[ \neg t U^{\leq 13} \text{crash} \right]
\]

where \( t \) is the state that should be avoided.

Schedulers and fair schedulers. The above explanation is ambiguous if nondeterminism is present, because the probability will (in general) depend on the resolution of non-determinism. Non-determinism is resolved by schedulers.

A scheduler is a function from the incomplete runs of an MDP to a possible continuation: for each incomplete run \( \sigma_1 \xrightarrow{\mu_1} \sigma_2 \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_{n-1}} \sigma_n \), it selects a distribution from \( \text{Distr}(\sigma_n) \). So, the scheduler does not resolve probabilistic choices. Several types of schedulers do exist [6].

A scheduler is strictly fair, if all runs that it selects are fair. A scheduler is fair, if almost all runs that it selects are fair, i.e., it selects fair runs with probability 1.

We interpret a PCTL-formula over all fair schedulers. Thus, for instance, \( \mathcal{P}_{\leq 10^{-5}} \left[ \neg \varphi U^{\leq 13} \text{crash} \right] \) is valid if for every fair scheduler, the probability to reach a crash-state within 13 steps (without visiting a \( \varphi \)-state) is at most \( 10^{-5} \). Properties that should be interpreted over some fair scheduler can be stated via duality, e.g.: “there is a fair scheduler such that the probability to reach a blue state after only visiting red states is \( \leq p' \), \( \mathcal{P}_{\leq p}(\text{red } \exists U \text{ blue}) \), is equivalent to “it is not the case that for all fair schedulers, the probability to reach a blue state after only visiting red states is \( > p' \), \( \neg \mathcal{P}_{> p}(\text{red } U \text{ blue}) \).

Example 4.5.2 In figure 4.13, the complete run start \( \xrightarrow{\mu} t \rightarrow \text{start} \xrightarrow{\mu} t \rightarrow \text{start} \rightarrow \cdots \) is not fair, since start is visited infinitely often and the transition to \( v \) is ignored infinitely often. Every other complete run ends either in \( v \) or in crash after finitely many steps and, therefore, is fair. The scheduler that always chooses the transition start \( \rightarrow v \) is strictly fair, since the unfair run is not selected. For this scheduler, property \( \Psi \) from example 4.5.1 holds, as the probability to reach state crash is 0.

The scheduler that always chooses the transition start \( \xrightarrow{\mu} \cdots \) is not strictly fair, because the unfair run could be selected. Nevertheless, the scheduler is fair because the probability that the unfair run is selected is 0.

\[5\]Recall that an infinite run \( \sigma_1 \xrightarrow{\mu_1} \sigma_2 \rightarrow \cdots \) is fair if for every state \( \sigma \) that is visited infinitely often, each element of \( \text{Distr}(\sigma) \) is also chosen infinitely often.
PCTL interpreted over P-statecharts. We also define an interpretation of a PCTL-formula over a finite collection of P-statecharts \( \{PSC_1, \ldots, PSC_n\} \) and one of its states \( (s_1, \ldots, s_n) \) where \( s_i = (C_i, I_i, V_i) \). Formally, the semantics is defined via the MDP semantics, i.e., \( \{PSC_1, \ldots, PSC_n\} \models \varphi \) iff the corresponding MDP satisfies \( \overline{\varphi} \). Here \( \overline{\varphi} \) denotes a syntactic translation needed to “hop along” the auxiliary (\( \Delta \)-labelled) MDP states. It is defined by induction over the structure of formulas. For elementary PCTL-formulas, such as atomic propositions and variable constraints, this translation is simply the identity, e.g., \( \text{true} = \text{true} \). For the remaining operators we have:

\[
\begin{align*}
\overline{\varphi \land \psi} &= \overline{\varphi} \land \overline{\psi} \\
\overline{\neg \varphi} &= \neg \overline{\varphi} \\
\overline{\mathcal{P}_{\geq P} [\varphi \cup \Delta k \psi]} &= \mathcal{P}_{\geq P} [\overline{\varphi} \cup \Delta \overline{\Delta k \psi}] \\
\overline{\mathcal{P}_{< P} [\varphi \cup \psi]} &= \mathcal{P}_{< P} [\overline{\varphi} \cup \Delta \overline{\psi}]
\end{align*}
\]

Example 4.5.3 For the P-statechart in figure 4.1 and its MDP semantics in figure 4.10, we express the following properties:

- “The probability that eventually the game will be over is 1”:
  \[ P_{=1} [\Diamond \neg 1 . \text{isin(playing)}] \]
  This property holds for all fair schedulers: a fair scheduler must choose to toss the coin infinitely often (with probability = 1) and is not allowed to ignore the outcome forever (with probability > 0).

- “In less than 50% of the cases, the game will be won within at most 20 steps”:
  \[ P_{<0.5} [\Diamond \leq 20 1 . \text{isin(won)}] \]
  The truth of this property depends heavily on the scheduler’s decisions in the first 20 steps.

4.6 Example: Hawks and Doves

This section applies P-statecharts and PCTL to the specification and verification of the behaviour of a small example taken from theoretical biology. Conflicts between animals are often analysed using simulation techniques. We consider the following variant of the hawk–dove-game [20, 77], a classical simulation to explain why it is sometimes better to fight only with limited effort.

In a population of animals, individuals combat for some advantage (such as food, dominance, or mates), their success being measured in points. Individuals may fight using several strategies. In particular, we consider:

**Hawk strategy:** Hawk-like individuals will fight with great effort, until they win the contest (+5 points) or are severely injured (−3 points).

**Dove strategy:** Dove-like individuals will fight with limited effort, until they win the contest (+5 points) or give up after some fight (−1 point). When facing a hawk, they immediately give up (±0 points).
We consider a small scenario with three individuals and an arbiter. In every round, the arbiter chooses nondeterministically a pair of individuals; they will be opponents in the next contest. The two individuals select probabilistically the hawk or dove strategy. The arbiter decides who wins. If the opponents choose the same strategy, the arbiter gives them equal chances.

Figures 4.14 and 4.15 show the P-statechart for one individual and the arbiter, respectively. The players all start off with 17 points and the individual scores may float in the interval $[0, 55]$ (otherwise they stop). Applying the MDP semantics of section 4.3 together with some further optimisations (omitting trivial intermediary states, encoding the configuration efficiently) leads to a system of 3,147,947 reachable states. The size of the state space is mainly dominated by the integer variables storing the scores.

### 4.6.1 Scenarios checked

Different scenarios were checked with the model checker PRISM [66] where each scenario consisted of different types of animals. These types were generated by taking different values for $p$, the probability to behave like a dove. Formulas are checked for the initial state. The three considered scenarios are the following.
4.6. Example: Hawks and Doves

One daring and two careful players. This is a scenario with two individuals \((c_1\text{ and }c_2)\) for which \(p = 0.75\) and one individual \((d)\) with \(p = 0.5\). The probability that any individual dies (its points drop below zero: \(\text{dead}(i) := (\text{points}_i < 0)\)) turns out to be very small, with the daring individual running a higher risk of being killed, since

\[
P_{\leq 10^{-7}}[\neg \text{dead}(c_1) \land \neg \text{dead}(d)] \cup \text{dead}(c_2) \quad \text{holds (4 min 54")}
\]

(and likewise with \(c_1\) and \(c_2\) reversed), but

\[
P_{\leq 10^{-7}}[\neg \text{dead}(c_1) \land \neg \text{dead}(c_2)] \cup \text{dead}(d) \quad \text{is refuted (5 min 28")}.
\]

The times after a formula indicate how long PRISM needed to verify resp. falsify the property. Before checking any property, PRISM needed 20’ to build the model. All times were measured on a Sun UltraSparc 10 with 256 MB RAM under SunOS 5.8.

The actual probability of \(d\) dying first is (depending on the scheduler) at most \(7.206 \cdot 10^{-7}\), while the probability of the careful one dying first is at most \(5.923 \cdot 10^{-8}\) (each). We have found these and other values by trying several probabilities in a formula of the above form.

On the other hand, the daring individual is likely to outperform the others on earning a certain number of points, say 20. This follows from verifying:

\[
P_{< 0.5}[(\text{points}_{c_1} < 17 + 20 \land \text{points}_d < 17 + 20) \cup \text{points}_{c_2} \geq 17 + 20] \quad \text{holds (1 min 46")}, \text{ and}
\]

\[
P_{\leq 0.75}[(\text{points}_{c_1} < 17 + 20 \land \text{points}_{c_2} < 17 + 20) \cup \text{points}_d \geq 17 + 20] \quad \text{is refuted (1 min 53")}.
\]

Three aggressive players. In this scenario each animal \((d_1, d_2, d_3)\) plays hawk with probability 0.9 (i. e., \(p = 0.1\)). The probability that some of the individuals dies is relatively high, e. g.,

\[
P_{\leq 0.01}[(\neg \text{dead}(d_1) \land \neg \text{dead}(d_2)] \cup \text{dead}(d_3) \quad \text{is refuted (6 min 7")}
\]

(likelihood for the permutations of the \(d_i\)). So, there are schedulers which will lead to \(d_3\) dying first with more than 1 % chance. The probability that one of the individuals earns at least 20 points within 100 steps is always less than 0.75, as

\[
P_{< 0.75}[\diamondsuit \leq 100\text{ } (\text{points}_{d_1} \geq 17 + 20)] \quad \text{holds (3 min 28")}.
\]

Three careful players. In the opposite situation (the three individuals play dove with probability 0.9), the individuals \((c_1, c_2, c_3)\) are less likely to die and more likely to get a reward fast. The probability that any of the individuals dies is rather low as, e. g.,

\[
P_{\leq 10^{-10}}[(\neg \text{dead}(c_1) \land \neg \text{dead}(c_2)] \cup \text{dead}(c_3) \quad \text{holds (5 min 17")}.
\]

So, for any fair scheduler, the probability of \(c_3\) dying first never exceeds \(10^{-10}\). The probability that one of the individuals earns at least 20 points within 100 steps turns out to be greater than 0.8, since

\[
P_{\geq 0.8}[\diamondsuit \leq 100\text{ } (\text{points}_{c_1} \geq 17 + 20)] \quad \text{is refuted (3 min 36")}.
\]
Conclusion. As a general conclusion of the experiments we may state that it is good for a population as a whole if the animals are careful; but an individual may be at an advantage if it is more daring than the others.

The probabilistic choice in the hawk–dove game is used for the following purposes:

- We abstract from the reasons or data dependencies that lead an individual to choose a certain strategy.
- When we only model individuals that always choose the same strategy, we need a much larger number of individuals to produce a realistic model. So, the probabilistic choice between strategies actually allows us to reduce the size of the model, while we are still able to choose $p$ arbitrarily.
- Fairness of the arbiter is guaranteed by probabilistic choice: if both contestants choose the same strategy, they have equal chances to win.

Describing the model using P-statecharts allowed us to structure it: we described the individual and the arbiter separately, keeping the diagrams simple. We used hierarchy in the statechart for an individual to express common behaviour in several states.

Note that in the hawk–dove game, we did not verify desired properties. As in scientific experiments, we wanted to find the actual properties of the researched object without changing it.

Limitations of our experiments. Calculating the expected number of points shows that the maximum of expected points is not reached at $p = 1$ (where the animals behave like doves with probability 1), but at some smaller value $p \in (0, 1)$, depending on the exact distribution of points in the different contest situations. $E_{dove}$ and $E_{hawk}$, the expected number of points a dove resp. a hawk will get in a single contest, are:

$$E_{dove} = p(0.5 \cdot 5 + 0.5 \cdot -1) + (1 - p) \cdot 0 = 2p$$
$$E_{hawk} = p \cdot 5 + (1 - p) \cdot (0.5 \cdot 5 + 0.5 \cdot -3) = 4p + 1$$

So, the total expected value is:

$$E = p \cdot E_{dove} + (1 - p) \cdot E_{hawk} = -2p^2 + 3p + 1$$

The maximum for $E$ is, in our setting, reached at $p = \frac{3}{4}$. Unfortunately, we could not prove a similar result with PRISM. We expect that a model checker based on the work by de Alfaro [22] could express and prove this property.

4.7 Example: Gambling Machine

This section applies P-statecharts to another example, where parallelism helps in creating a compact representation.

A fruit machine is a kind of gambling machine that contains three reels that show fruit symbols. When the user starts a game, the reels spin until the user stops them. If the visible parts of the reels show some specific combination of symbols (e.g., three times cherry), the player gets a prize. For simplicity, we do not consider the display.
The fruit machine contains a combination of probabilistic and nondeterministic elements; for example, the outcome of a single reel is probabilistic, but which of the reels stops first is nondeterministic.

The system contains the following objects: the \textit{reels and prizes} object describes the reels’ outcome and which prize the user gets. The \textit{game controller} ensures a correct sequence of play. The \textit{cash box interface} handles contact with the cash box for payments and prize money.

A typical interaction of the objects is shown in the collaboration diagram in figure 4.16. It depicts the following scenario: a user enters a single coin and then presses the start button. The user presses the three “stop” buttons and then gets a prize.

The statechart for the reels and prizes object is shown in figure 4.17. Note that the state of the reels is only registered as long as it is needed. We use the priority scheme from Harel and Gery [44]: smaller scopes have higher priority. This implies, e.g., that P-edges to subnodes of triple winning have priority over P-edges to subnodes of double winning, and so, the highest prize is found and paid. Some P-edge labels have been omitted, namely the triggers of the edges to find the prize: each edge that leads to a prize has the label \texttt{prize / send cashbox\textunderscore pay(amount)} with the appropriate amount from table 4.1 filled in. We have shown one example.

The statecharts of the other objects are shown in figures 4.18 and 4.19.

We have composed the above statecharts with a simple user simulation and con-
Figure 4.17: The reels and prizes’ behaviour. To avoid cluttering, some P-edge labels have been omitted and some P-edges are drawn dashed or dotted.
4.7. Example: Gambling Machine

The game controller’s behaviour

![Diagram of the game controller's behaviour]

The cash box interface’s behaviour

![Diagram of the cash box interface's behaviour]

structed the MDP which is their semantics. The MDP has been fed into the model checker PRISM [66], a probabilistic model checker for MDPs and similar structures.

We had to put a bound on the integer variable which represents the stock of return money (as in a real automaton, there is only room for a finite amount of coins). The automaton starts with 200 coins. We have added a game counter to express properties for a series of games. After some simplifications (omitting trivial intermediary states etc.), the MDP had 1,789,057 reachable states.

4.7.1 Desired Properties Checked

We have checked the MDP with some properties which resemble legal requirements to the automaton [96]. Most of these requirements are conditions on the allowed minimal, maximal, or mean loss of the player. We have added a variable gamecount to the model to help in formulating requirements on a series of games. We have checked the following properties (on a Sun UltraSparc 10 with 256 MB RAM under SunOS 5.8):

The player doesn’t lose too much. In 30 games, the player loses at most 70% with probability at least 0.9 (a single game costs 1 coin, so the maximum loss is 30 coins):

\[
P_{\geq 0.9}\left[\text{gamecount} < 30 \cup \text{gamecount} = 30 \land \text{money} \leq 200 + 0.7 \cdot 30\right]
\]

Remember that money represents the amount of money in the automaton; a loss for the player makes money rise. PRISM reported in 4 min 20" that the property holds.
The player doesn’t win too much. In 30 games, the player wins less than 10 coins with probability at least 0.99:

\[ P_{\geq 0.99}[^{\text{gamecount < 30}} \cup (\text{gamecount = 30} \land \text{money > 200 - 10})] \]

Also here, PRISM reported in 5 min 47” that the property holds.

The median loss is less than 50 %. In 30 games, the probability to lose at least 15 coins is less than 50 %:

\[ P_{< 0.5}[^{\text{gamecount < 30}} \cup (\text{gamecount = 30} \land \text{money \geq 200 + 15})] \]

PRISM reported in 4 min 45” that the property holds.

The stock of coins is large enough. This is not a legal requirement, but a practical one: how large should the stock of return money be that the chance a prize cannot be paid is less than $10^{-20}$? Let’s check whether it is large enough:

\[ P_{\leq 10^{-20}}[\Diamond \text{money < 1}] \]

PRISM reported that the property is false. Experimentation with some other probabilities led to the conclusion that the probability that a prize cannot be paid some time is quite high. However, when we assume that the coins are refilled regularly (say, every 1000 steps), it is enough to prove

\[ P_{\leq 10^{-20}}[\Diamond^{\leq 1000} \text{money < 1}] \]

which is true. PRISM needed 16 min 31” to check this; we assume that the check is slow because the tool unfolds the system to 1000 steps.

Conclusion. We can conclude that the automaton model has the desired properties, provided that we check the coin stock in the fruit machine on a regular basis.

Simulation or experimentation with a physical gambling machine always has the risk that the measured loss over the test series of games is larger than allowed; to minimize this risk, the manufacturer includes a safety margin in the system, for example by making the median loss 40 % while the law required it to be not more than 50 %. On the other hand, there is also a small risk that a gambling machine passes the test while it actually does not satisfy the legal requirements. With probabilistic model checking, both risks are avoided, as we can calculate the probabilities exactly.

With P-statecharts, we were able to produce a simple description of the reels and prizes object. We have modelled the three reels independently; it was not necessary to draw all possible combinations explicitly. Without this or a similar feature, we were obliged to draw the $6 \cdot 6 \cdot 6$ states one for one, similar to figure 4.8. So, the parallelism defined in statecharts enables engineers to create models of complex systems. It is a relatively slight disadvantage that we have to draw hyperedges for every winning combination.

Hierarchy and a suitable priority scheme allowed us to find the highest prize very easily.
Requirements that could not be checked. Some of the legal requirements are real-time properties, e.g.: Dutch law [96] requires that a game must last at least 4 seconds on average. P-statecharts do not include real-time, so it is impossible to check this requirement. Chapter 5 is going to handle stochastic timing in statecharts; if we model the system according to that extension of statecharts, we can check real-time properties.

4.8 Related Work

Harel [43] already mentions, a possible probabilistic variant of statecharts and calls them “Markov-charts”: they could be seen as an extension of discrete time Markov chains. However, Harel never has defined a probabilistic variant himself.

King and Pooley [63] describe a system using UML collaboration diagrams and statecharts, very similar to our use of the UML. Probabilistic choice is modelled by special guards stating the probability. These guards can easily be translated to our P-edges. King and Pooley analyse a system model by translating it to a generalised stochastic Petri net (GSPN). A disadvantage of GSPN is that they cannot model nondeterminism.

Many authors concentrate on a stochastic extension of statecharts and treat the probabilistic extension only in the margin. For example, Huszerl et al. [53] define a formalism called guarded statecharts. These structures contain probabilistic choice and stochastic timing. The guarded statecharts language, which is not hierarchic, is then extended to include UML statecharts. The statecharts can be analysed via a translation to Stochastic reward nets, a variant of GSPN.

MoDeST is a powerful language to describe the behaviour of probabilistic and stochastic systems. It has been introduced by D’Argenio et al. [24]. Probabilistic choice is modelled by a palt operator. MoDeST has a formal semantics in terms of stochastic timed automata.

Feldman and Harel [36] and Kozen [65] have defined a probabilistic dynamic logic, where propositions are generalised to measurable functions and action modalities (similar to our $X_a$ from chapter 3) are replaced by function transformers. The logic replaces all nondeterminism present in dynamic logic by probabilistic choice. It seems that all random variables have the same distribution. Feldman and Harel propose an axiomatic reasoning system for the logic.

De Alfaro [26] can express properties about the long-time average (for example, the expected loss when playing the fruit machine) using an annotated graph. PCTL allows to express and verify statements about the median and other percentiles of a random variable, but not about its average.

4.9 Discussion and Conclusion

Adaptation to other statechart semantics. The extension to P-statecharts described in this chapter can be applied to a wide range of other semantic definitions. The main idea of our extension is:

1. Syntactically, probabilities are trigger-guarded, i.e., reactions to triggers may depend on the result of a probabilistic experiment, whereas the triggers themselves
are not subjected to probabilities. This restricts our approach to describing system randomness, opposed to environmental randomness.

2. Semantically, we reduce the P-statechart probabilistically to a (traditional) statechart, and this is done just before a step. The step is constructed and executed in the traditional statechart setting, and the step’s result is interpreted in the P-statechart again. Such a reduction is possible as long as the effects of probabilistic experiments are encapsulated in the steps and cannot be sensed in the states.

In principle it is possible to define a (traditional) statechart semantics which – if interpreted in the probabilistic extension – would break the encapsulation of probabilities within a step. For instance, one could imagine a semantics where a state variable depends on the enabledness of specific outgoing edges (which could only be decided after resolving the probabilism). However, such a feature appears to be rarely used, the overview given by von der Beeck [8] does not mention anything like this.

Possible simplifications. Section 4.3.5 has mentioned BPTS as the most natural model for a P-statechart semantics. Thus, we could have simplified the semantics if there were a BPTS model checker available.

On the other hand, observe that the examples in section 4.3.1 depend on the fact that some P-edge allows a probabilistic choice between edges with different scopes. For many priority schemes, and for consistency, not the actual targets are relevant, but the edge’s scope (which depends only on the parents of the source and of the target). If we disallow probabilistic choices between edges with different scopes, we could resolve nondeterminism and probabilism in a different order and simplify several points: Theorem 4.4.1 can be formulated and proved simpler. It becomes feasible to give a direct semantics in terms of MDPs (i.e., without intermediary states), which is closer to the intuition behind P-statecharts. However, it is no more possible to express behaviours like the examples in section 4.3.1.

Lessons learnt. It is easy to formulate an intuitive extension of statecharts with probabilities. However, when we started formalising and detailing it, a delicate balance had to be found with the other features of statecharts. Our first version of theorem 4.4.1, for example, didn’t work properly in the case that some edge has a higher priority than another edge which belongs to the same P-edge.

We have tried to formulate the extension as powerful as possible. This also revealed the problems of extending statecharts more clearly. For some applications, a simpler extension, or a simpler variant of basic statecharts is enough.

In the hawk–dove example, we have seen that a small-looking example can become a model with several millions of states. In the gambling machine example, P-statecharts simplify the presentation of the prize mechanism considerably by parallelism and hierarchy. This shows that P-statecharts are suitable to describe the behaviour of a nontrivial probabilistic system. It would be impossible to get an overview of these models using one of the basic probabilistic modelling languages.
Chapter 5

Stochastic Extension: StoCharts

5.1 Introduction

Requirements of current embedded and other systems often include soft real-time constraints. It is not a fatal error of such a system if the deadline of a soft real-time constraint is missed every now and then. However, to ensure an acceptable quality of service, the deadline should be attained almost always. Therefore, a stochastic approach lends itself to describe the requirements of these systems and to produce their requirements-level models.

Mathematically speaking, the soft real-time characteristics of a given embedded system induce families of stochastic decision processes, e. g. Markov chains or semi-Markov decision processes. However, these mathematical objects are too fine grained to be directly specifiable by a requirements engineer. Therefore, we want to provide an interface between these mathematical structures and the languages used by requirements engineers.

Often, systems are developed based on UML descriptions. While in principle the UML provides the right ingredients to model discrete event dynamic systems, it lacks support for stochastic process modelling. This issue has been addressed in both the UML profile for schedulability, performance and time [99] and in the request for proposals to model quality of service and fault tolerance [98]. These profiles suggest annotational extensions of UML to specify performance, dependability and other quality of service characteristics.

However, the vague semantics of the UML and of its annotational extensions drastically hampers QoS analysis: It is simply impossible to distill a faithful performance or QoS quantity from a stochastic (decision) process that is only partially defined. Model-based QoS prediction is only possible for UML fragments with a rigorous formal semantics.

In this chapter, we provide a formal semantics of an extension of statecharts, the extension being both simple and easy to understand, yet powerful enough to model a sufficiently rich class of stochastic decision processes.

The extension is twofold. One extension allows state transitions to select probabilistically out of different effects, much like the rolling of a die can have one out of six effects, determined probabilistically. This extension is similar to the one described in the previous chapter. The second extension is novel, yet simple and conservative: The after operator
of statecharts is given a stochastic interpretation, allowing the use of arbitrary probability distributions for modeling.

The resulting statecharts dialect is called StoCharts, and contains UML statecharts (up to some minor features such as deferred events) as a subset.

We provide a formal semantics for StoCharts in terms of stochastic input–output automata. This semantics extends the statechart semantics provided in section 2.2. It enables us to formally verify that a system, described by a number of StoCharts, has some desired properties, including soft real-time properties.

**Stochastic model checking.** The stochastic input–output automata can be reduced to Markov chains or similar structures, under some conditions. These structures serve as the input for stochastic model checkers. Desired properties can be expressed using the logic CSL [4, 5]. For example, $\text{E}^\dagger \text{MC}^2$ [50] can check whether a Markov chain satisfies some CSL property.

**Open system.** In this chapter, we want to handle system randomness as well as environmental randomness. Therefore, we include possibilities to compose a system model with an environment model. We do not restrict ourselves to closed systems on beforehand. We simplify model checking by regarding the complete model (system and environment) as closed.

### 5.2 StoCharts

A StoChart is an extension of a statechart by probabilistic choice and stochastic timing. It is a further extension of P-statecharts from chapter 4. Stochastic delays are defined by an after(cdf) operator, an extension of the after(duration) operator of the UML [83].

**Collection of statecharts.** A system consists of a finite collection of communicating statecharts. In the following, we assume a given finite collection of StoCharts, denoted by $\{\text{SC}_1, \ldots, \text{SC}_n\}$. This very simple system model makes it relatively easy to analyse it automatically.

This system is composed with an environment model. The environment can be described with a Stochastic Input–Output Automaton (a kind of automaton we will define below) or with another StoChart.

**Syntax.** A single StoChart $\text{SC}_i$ consists of the following elements:

- A finite set $\text{Nodes}_i$ of nodes with a tree structure, as for statecharts (see the definition in section 2.2).
- A finite set $\text{Events}_i$ of events, as for statecharts.

Later on, we will also use pseudo events. A pseudo-event is an expression of the form after($F$), where $F$ is a cumulative distribution function (cdf)\(^1\). The set of pseudo-events is denoted $\text{PsEvents}_i$.

---

\(^1\)See appendix A for a formal definition of cdfs.
5.2. StoCharts

- A finite set \( Vars_i \) of variables, as for the extension of statecharts described in section 2.4.1. The initial valuation \( V_{0,i} : Vars_i \rightarrow \mathbb{Z} \), defines initial values to the variables. (We will only allow bounded integer variables for model checking.)

- A set \( Guards_i \) of guard expressions, as for statecharts. In addition to guards of the form \( j.isin(x) \) (for \( j \in \{1, \ldots, n\} \) and \( x \in Nodes_j \)), we allow comparisons like \( expr \leq expr \) and \( expr \geq expr \), for arithmetic expressions made up from the variables and integer constants, as for P-statecharts.

- A set \( Actions_i \) of actions, as for statecharts. In addition to actions of the form \( send \ j \) \( e \) (for \( j \in \mathbb{N} \) and \( e \in Events_j \)), we allow assignments like \( v := expr \), where \( v \in Vars_i \). Note that we allow to send events to an external component here, by specifying a \( j \not\in \{1, \ldots, n\} \).

- A finite set \( PEdges_i \) of P-edges, replacing the set of edges of a statechart, as for P-statecharts. A P-edge is a tuple \( (X,e,g,P) \) where \( X \subseteq Nodes_i \) is a non-empty set of source state nodes, \( e \in Events_i \cup PsEvents_i \cup \{\bot\} \) is the triggering event (here, we allow pseudo events in addition to P-statecharts), \( g \in Guards_i \) is a guard, and \( P \) indicates the possible actions and target state nodes. Formally, it is a probability measure in the discrete probability space \( (\mathbb{P}(Actions_i) \times (\mathbb{P}(Nodes_i) \setminus \{\varnothing\}), P) \). The measure \( P \) has a restriction (different from P-statecharts) which will be explained below.

We assume that there is a bijective index function \( i : \{1, \ldots, |PEdges_i|\} \rightarrow PEdges_i \) to simplify the identification of P-edges. If \( i(j) = (X_0, e_0, g_0, P_0) \), we sometimes write \( i(j).X \) for \( X_0 \) etc.

We will denote a StoChart by \( SC_i = (Nodes_i, Events_i, Vars_i, PEdges_i) \).

Restriction on P-edges. An edge is a triple \( (j, A, Y) \), where \( j \) identifies a P-edge, \( A \subseteq Actions_i \) is a set of actions and \( Y \subseteq Nodes_i \) is a set of target nodes. The set \( Edges_i \) is defined as: \( \{(j, A, Y) \mid \exists X, e, g, P : i(j) = (X, e, g, P) \in PEdges_i \land P(\{(A,Y)\}) > 0\} \).

We require that all edges (with probability \( > 0 \)) belonging to a single P-edge have the same scope. This scope is also called the scope of the P-edge. It is denoted \( scope(j) \).

As a rule of thumb, one could say: When drawing a StoChart, avoid arrows from a P-pseudonode that cross node borders (like interlevel transitions do). There are only a few situations where crossing node borders is allowed. Figure 5.1 gives some examples. This restriction is needed later to simplify the semantics.

Comparison to statecharts. The above definition extends our former definition of statecharts by local variables, P-edges (as for P-statecharts) and pseudo events. Pseudo events are new for StoCharts: it is possible to specify a timeout trigger depending on a timeout by a pseudo-event after\( (F) \). The cdf \( F : \mathbb{R}_+^2 \rightarrow [0, 1] \) describes the probability distribution of the waiting time until the timeout happens.

Drawing a StoChart. StoCharts are drawn in the same way as P-statecharts. A P-edge consists of two parts: first an arrow with event or pseudo event and guard \( e[g] \) that
5.2 Informal Semantics

The behaviour of a StoChart is similar to that of a P-statechart. The StoChart is always in some state (which consists of one or several nodes). A P-edge is enabled if the event happens or the timeout occurs while the StoChart is in the source node(s) and its guard holds. Then, the system chooses one of the possible targets (nondeterministically and probabilistically). The system executes as many enabled P-edges at once as possible without conflict: it leaves the source nodes, executes the chosen actions and enters the chosen target nodes of the P-edges. If some after(cd f) edge leaves the entered node(s), a timer is set according to the distribution; this edge will become enabled when the timer will expire.

5.2.1 Example: A Car Damage Assessor

We extend example 2.2.1 which modelled a simple workflow for a car damage assessor. His task is to assess on behalf of an insurance company whether a damaged car should be repaired and whether the garage offers an acceptable price for the repair. If the estimated damage is beyond € 1000, the assessor has to physically inspect the car before repair.

More exactly, the assessor performs the following tasks:

**Contact garage:** The assessor phones the garage and gets a preliminary estimate of the repair cost. A first estimate of the inspection date is also made (if necessary).

**Assess damage:** If the estimate of the repair costs exceeds € 1000, an assessor (not necessarily the same) physically inspects the car. Otherwise, the damage is assessed via the phone.
5.2. StoCharts

Figure 5.2: Damage assessor process. The labels after(EXP[n min]) mean: the delay is described by an exponential distribution with average duration n minutes.

**Negotiate with the garage:** The assessor negotiates final repair costs with the garage. This takes place if there has been a physical inspection, but also when there has only been an estimate of the damage over the phone.

In some cases the assessor decides that the car is not worth repairing. The car will then be termed a write-off and the process will exit.

**Write report:** The assessor compiles a report about the assessment.

**Check invoice:** The assessor checks the invoice of the garage before forwarding it together with the report to the insurance company.

If the inspection date is later than desirable, the company may decide to assign the inspection to another assessor who is less busy, but is more expensive. This is mainly done with damages of VIP clients.

We have extended the statechart from figure 2.1 to a StoChart by introducing timing information. For example, the time until a physical inspection can be made is 3 days on average; for VIP treatment, it is 1.5 days. We also abstract from the exact amount of the damage by probabilities; for example, the insurance company knows by experience that in 70% of the cases, phone assessment is enough. Figure 5.2 contains a StoChart that models the assessor’s workflow.

**Requirement.** With a stochastic model checker like E-MC^2 [50], one could verify the following requirement: The insurance company wants to provide a certain quality of service. In at least 95% of the cases, the repair should start within 4 days after the damage assessor has started his work (in the formal model, the occurrence of event start).
5.3 Stochastic Input–Output Automata

We use stochastic input–output automata to define a formal semantics for StoCharts. They extend input–output automata with stochastic delays and probabilistic choice.

A Stochastic I/O-Automaton (IOSA) is a structure \((L, T, A, I, O, D, AP, J, l_0)\) that consists of the following parts:

- \(L\) is a finite, non-empty set of locations. (Note that there are no location invariants.)
- \(T\) is a finite set of timers. Each timer \(t \in T\) has an associated cdf \(F_t : \mathbb{R}_0^+ \rightarrow [0, 1]\).
- \(A\) is a finite set of actions, partitioned into a set \(A^{\text{in}}\) of input actions and a set \(A^{\text{out}}\) of output actions.
- \(I : L \times A^{\text{in}} \rightarrow L\) describes the input transition relation.
- \(O \subseteq L \times \text{Prob}(A^{\text{out}} \times P(T) \times L)\) is the probabilistic output transition relation.
- \(D \subseteq L \times T \times L\) is the delay transition relation.
- \(AP\) is a set of atomic propositions.
- \(J : L \rightarrow P(AP)\) is the interpretation of the atomic propositions.
- \(l_0 \in L\) is the initial location.

We require that the input transition relation \(I\) is a total function. This means that we assume input enabledness: the system always accepts any input. (Of course, it may choose to ignore the input, by just looping to the same location.) It also means that the input transition relation is deterministic: there is neither a probabilistic nor a nondeterministic choice of a next state.

The output transition relation assigns to each location a (possibly empty) set of probability spaces, between which the system chooses nondeterministically. After that, the system chooses probabilistically one of the elements of the probability space as a target. This target consists of an action which is generated, a set of timers which are set according to their cdf, and a next location.

Timers. Timers are somewhat different from the clocks used in chapter 3 to measure time. The timers we define here run backwards and enable a delay transition when they reach 0. Therefore, clock constraints are left implicit here. When the IOSA takes an output transition, some timers are set to a (in most cases, positive) value. Timers are defined by D’Argenio [22] to describe models with distributions that are not memory-less. We also use them because parallel composition of stochastic systems can be expressed in a natural way.

A time valuation \(v\) is a function that assigns a value to each timer: \(v : T \rightarrow \mathbb{R}\). The set of all time valuations for a set of timers \(T\) is denoted \(\text{Val}_T\).

We define the following operation on valuations: If \(v\) is a valuation and \(\delta \in \mathbb{R}\), then \(v - \delta\) denotes the valuation that results after waiting \(\delta\) time units. Formally, it is defined by:

\[
\forall t \in T : (v - \delta)(t) = v(t) - \delta
\]
5.3. Stochastic Input–Output Automata

5.3.1 Example

As an example for an IOSA, we model the behaviour of a bank from the perspective of an automatic teller machine (ATM). See figure 5.3 for a schematic drawing of the IOSA.

The ATM sends a request whether a certain withdrawal is allowed (allowed) to the bank; from the bank’s point of view, this is an input. It needs some time to check the allowance; this is modelled by setting a timer \( t \) and waiting until it expires. If the outcome of the check is positive, the ATM may request to actually withdraw the money from the account (withdraw). If the ATM sends a request to withdraw without asking for allowance, the bank reports an error by entering the state Error. On the other hand, it is no error to ask for allowance without actually withdrawing money; therefore, it is also possible to start the next request round from state Check positive.

5.3.2 Semantics of IOSA

Informal semantics. A IOSA is always in a state, which consists of a location \( l \) and the current timer values. The system may choose any of the enabled transitions to proceed to another state. The enabling is subject to the following conditions, and evolving to another state has the following effects:

- Output transitions are always enabled. To take an output transition, the system chooses nondeterministically a probability space \( \mathcal{P} \) such that \((l, \mathcal{P}) \in \mathcal{O}\), then chooses probabilistically one of the possible targets \((a, T, l')\) in \( \mathcal{P} \), generates the associated action \( a \), resets the corresponding timers \( T \), and evolves to the new location \( l' \).

- When some external source provides an action \( a \), the input transition \( I(l, a) \) is enabled. If it is taken, the system evolves to location \( I(l, a) \).

- A delay transition \((l, t, l') \in D\) is enabled when \( t \) reaches 0. If it is taken, the system evolves to location \( l' \).\(^2\)

\(^2\)When several timers expire at the same moment or if an action happens at the same moment as the timer expiration, there is a nondeterministic choice between the transitions. The probability that any of these two
If no transition is enabled in the current state, time may pass until a transition is enabled (i.e., a timer reaches 0 or an external source provides an action).

A IOSA starts its behaviour in the initial location with all timers set according to their cdfs: the cdfs also describe the initial probability distributions over the timer values.

**Formal semantics.** For the special case where we assume no input (a closed IOSA with \( A^{in} = \emptyset \)), we can define a simple formal semantics in terms of Probabilistic Transition Systems [22]. A Probabilistic Transition System is a structure \( PTS = (\Sigma, \to) \) where

- \( \Sigma \) is a set of states;
- \( \to \subseteq \Sigma \times \text{Prob}(\Sigma) \) is the (probabilistic) transition relation that relates states to (not necessarily discrete) probability spaces over \( \Sigma \).

We write \( \sigma \to P \) instead of \( (\sigma, P) \in \to \). A probabilistic transition \( \sigma \to P \) is said to be trivial if its probability space \( P \) is trivial, i.e., probability one is assigned to a single state \( \sigma' \). In this case, we write \( \sigma \to \sigma' \).

Given a IOSA \( B = (L, T, A, I, O, D, AP, J, l_0) \), we define the PTS \( \llbracket B \rrbracket = (\Sigma, \to) \) by:

- A state is a quadruple consisting of a location, the action just generated (or ? to indicate that the last step was not an action step), a set of timers, and a valuation of the timers:
  \[
  \Sigma = L \times (A^{out} \cup \{ ? \}) \times P(T) \times \text{Val}_T
  \]
  The action and the set of timers serve to identify the output transition just taken.

- A PTS transition leads from the source state to a probability space over \( \Sigma \). Before we define the (non-discrete) \( \sigma \)-algebra over \( \Sigma \), we need some additional concepts and notations. A cdf \( F_t \) induces a unique probability space denoted \( (\mathbb{R}, B^1, \mu_{F_t}) \), where \( B^1 \) denotes the (standard) Borel-\( \sigma \)-algebra over \( \mathbb{R} \).

To simplify notation, we now assume that \( T = \{ t_1, t_2, \ldots, t_{|T|} \} \). Assume given some subset \( T = \{ t_{i_1}, \ldots, t_{i_n} \} \subseteq T \). The combination of several cdfs \( F_{t_{i_1}}, \ldots, F_{t_{i_n}} \) similarly induces a unique probability space denoted \( (\mathbb{R}^n, B^n, \mu_{F_T}) \), where \( B^n \) denotes the \( n \)-dimensional Borel-\( \sigma \)-algebra over \( \mathbb{R}^n \).

Then, we assume given a valuation \( v \in \text{Val}_T \), and we extend this probability space to all timers in \( T \): For timers \( t \in T \setminus T \), let

\[
F_t'(r) = \begin{cases} 
0 & \text{if } r < v(t) \\
1 & \text{otherwise}
\end{cases}
\]

(The measure that corresponds to a \( F_t' \) defined this way assigns probability 1 to the value \( v(t) \).) We combine the \( F_t \) (for \( t \in T \)) and the \( F_t' \) (for \( t \in T \setminus T \)) to form a \(|T|\)-dimensional probability space denoted \( (\mathbb{R}^{|T|}, B^{|T|}, \mu_{F_T,F_T'}) \).

Then, we define a mapping \( D : \mathbb{R}^{|T|} \to \text{Val}_T \) from the real space to the space of timer valuations by \( D(x_1, \ldots, x_n)(t_i) = x_i \). Note that this mapping can be used to define situations arise is only > 0 if we use non-continuous cdfs.

Further nondeterminism arises if several delay transitions with the same source state and timer are defined.
5.3. Stochastic Input–Output Automata

A probability space over $\text{Val}_T$ from a probability space over $\mathbb{R}^{\vert T\vert}$. In the following definition, we only use the $\sigma$-algebra $\mathcal{D}^*(\mathcal{B}^{\vert T\vert})$ over $\text{Val}_T$ that is generated via $\mathcal{D}$.

Returning to the $\sigma$-algebra over $\Sigma$: it is a combination of two $\sigma$-algebras:

- A discrete $\sigma$-algebra over $L \times A \times \mathcal{P}(T)$ that corresponds to the probabilistic choice in the output transitions of the IOSA. It is generated by the singleton sets $\{(l, a, T)\}$, with $(l, a, T)$ ranging over the whole set.

- A Borelian $\sigma$-algebra $\mathcal{D}^*(\mathcal{B}^{\vert T\vert})$ over the valuations $\text{Val}_T$ that corresponds to setting the stochastic timers of an output transition. It is generated by the intervals $D([r, s])$ (for $r, s \in \mathbb{R}^{\vert T\vert}$), as $[r, s)$ are the generators of $\mathcal{B}^{\vert T\vert}$.

The $\sigma$-algebra $\mathcal{S}$ over $\Sigma$ is defined as the smallest $\sigma$-algebra that contains the products $\{(l, a, T)\} \times D([r, s))$.

We now regard the PTS transitions starting in $(l, a, T, v) \in \Sigma$. For any IOSA output transition $(l, (\mathcal{A}^{\text{out}} \times \mathcal{P}(T) \times L, \mu)) \in O$, the corresponding PTS transition is:

$$(l, a, T, v) \rightarrow (\Sigma, \mathcal{S}, \mu^+)$$

where the measure $\mu^+ : \mathcal{S} \rightarrow [0, 1]$ is defined by:

$$\mu^+\{(l', a', T')\} \times D([r, s})) = \mu\{(a', T', l')\} \cdot \mu_{F_r, F_{\Sigma, T}}([r, s))$$

- For every delay transition $(l, t, l') \in D$, we have:

$$v(t) = 0$$

$$(l, a, T, v) \rightarrow (l', \bot, \emptyset, v)$$

- In states where no transition is enabled yet, we allow time to pass by:

$$\forall \delta' < \delta : (l, a, T, v - \delta') \not\rightarrow$$

$$(l, a, T, v) \rightarrow (l, \bot, \emptyset, v - \delta)$$

where $(l, v) \not\rightarrow$ holds in a state $(l, v)$ where none of the above rules to define a PTS transition (corresponding to an IOSA output or a delay transition) apply.

- The initial probability space describes the probability that the system starts in some specific state. Although it is not formally not a part of the PTS definition, we give as initial probability space: $\Sigma, \mathcal{S}, \mu_0$ where the $\sigma$-algebra $\mathcal{S}$ is defined as above and the measure $\mu_0 : \mathcal{S} \rightarrow [0, 1]$ is defined by:

$$\mu_0\{(l_0, \bot, \emptyset)\} \times D([r, s)) = \mu_{F_r, F_{l_0}}([r, s))$$

and $\mu_0\{(l, a, T)\} \times D([r, s)) = 0$ if $(l, a, T) \neq (l_0, \bot, \emptyset)$.

---

3A similar probability space is used in D’Argenio’s dissertation [22], page 115 and appendix D, from where we took the idea to define $D$. 


5.3.3 Composition of IOSA

IOSA can be composed similarly to IOA. A composition is useful for environment randomness: a system model can be composed with a model of its environment.

In a network of several automata, one automaton – chosen nondeterministically – takes an output transition at a time. The produced output action triggers the other ones, which immediately execute the corresponding input transition.

The composed automaton takes an input transition on action \( a \) if every component takes an input transition on \( a \). The composed automaton takes a delay transition if one component takes a delay transition and the other components do nothing.

Two IOSA \( \text{Iosa}_i = (L_i, T_i, \mathcal{A}_i, I_i, O_i, D_i, AP, J_i, l_{0,i}) \) (for \( i = 1, 2 \)) are called compatible if their output sets do not overlap: \( \mathcal{A}_{\text{out}} \cap \mathcal{A}_{\text{out}}^2 = \varnothing \). If the IOSA are compatible, their composition \( \text{Iosa}_1 \otimes \text{Iosa}_2 = (L, T, \mathcal{A}, I, O, D, \bar{\mathcal{A}P}, J, l_0) \) is:

- \( L = L_1 \times L_2 \)

- The set of timers is the disjoint union \( T = T_1 \cup T_2 \).

- \( \mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2 \), where \( \mathcal{A}_{\text{out}} = \mathcal{A}_{\text{out}}^1 \cup \mathcal{A}_{\text{out}}^2 \) and \( \mathcal{A}_{\text{in}} = (\mathcal{A}_{\text{in}}^1 \cup \mathcal{A}_{\text{in}}^2) \setminus \mathcal{A}_{\text{out}} \).

- To simplify the definitions of \( I \) and \( O \), we extend \( I_1 \) and \( I_2 \) by \( I_i(l_i, a) = I_i \) if \( a \notin \mathcal{A}_{\text{in}}^i \). \( I \) is defined by the rule:

\[
I((l_1, l_2), a) = (I_1(l_1, a), I_2(l_2, a))
\]

- \( O \) is defined by two symmetric rules. We only give the rule where \( \text{Iosa}_1 \) leads:

\[
(l_1, P_1) = (l_1, (\mathcal{A}_{\text{out}}^1 \times \mathcal{P} (T_1) \times L_1, P_1)) \in O_1 \\
(l_1, l_2, P) = ((l_1, l_2), (\mathcal{A}_{\text{out}}^1 \times \mathcal{P} (T) \times L, P)) \in O
\]

where \( P \) is the lift of the following probability weight to sets:

\[
\mu(a, T, (l_1, l_2, a))) = P_1(\{(a, T, l_1)\}) \quad \text{if} \quad a \in \mathcal{A}_{\text{out}}^1 \quad \text{and} \quad T \subseteq T_1
\]

and \( \mu(\omega) = 0 \) otherwise.

- \( D \) is defined by the two rules:

\[
(l_1, t, l_1') \in D_1 \\
((l_1, l_2), t, (l_1', l_2)) \in D
\]

\[
(l_2, t, l_2') \in D_2 \\
((l_1, l_2), t, (l_1, l_2')) \in D
\]

- The set of atomic propositions is the disjoint union \( AP = AP_1 \cup AP_2 \).

- The interpretation of atomic propositions is defined by the two rules:

\[
p \in J_1(l_1) \quad \quad p \in J_1(l_1) \\
p \in J_2(l_2) \quad \quad p \in J_2(l_2)
\]

- \( l_0 = (l_{0,1}, l_{0,2}) \)
The above definition composes exactly two automata, but could easily be extended to a finite collection of IOSA, as shown by the following lemma:

**Lemma 5.3.1** The composition is associative (up to structural isomorphism).

*Proof:* A structural isomorphism between two IOSA \((L_a, T_a, A_a, I_a, O_a, D_a)\) and \((L_b, T_b, A_b, I_b, O_b, D_b)\) consists of three bijections \(i_L : L_a \rightarrow L_b\), \(i_T : T_a \rightarrow T_b\), and \(i_A : A_a \rightarrow A_b\), with the obvious properties on the transition relations.

Given three (pairwise compatible) IOSA \(l o s a_i = (L_i, T_i, A_i, I_i, O_i, D_i)\) (for \(i = 1, 2, 3\)), we prove that \((losa_1 \otimes losa_2) \otimes losa_3 = (L_a, T_a, A_a, I_a, O_a, D_a)\) and \(losa_1 \otimes (losa_2 \otimes losa_3) = (L_b, T_b, A_b, I_b, O_b, D_b)\) are structurally isomorphic.

- \((L_1 \times L_2) \times L_3 \cong L_1 \times (L_2 \times L_3)\). The mapping is: \(i_L((l_1, l_2), l_3) = (l_1, (l_2, l_3))\).
- The disjoint unions to define the sets of timers are equal: \((T_1 \cup T_2) \cup T_3 = T_1 \cup (T_2 \cup T_3)\). So, \(i_T\) is the identity.
- The definition of compatibility is associative, so we need not bother about the order while checking for compatibility. \(A^o_{\text{out}} = (A^o_{\text{out}} \cup A^o_{\text{out}}) \cup A^o_{\text{out}}\) and similarly for \(A^o_{\text{out}}\), so that \(A^o_{\text{out}} = A^o_{\text{out}}\). A similar argument applies to \(A^o_{\text{in}}\).
- \(I_a\) is defined by the rule:

\[
\frac{I_1(l_1, a) = l_1'}{I_{12}((l_1, l_2), a) = (l_1', l_2')} \quad \frac{I_2(l_2, a) = l_2'}{I_3(l_3, a) = l_3'} \quad \frac{I_4(l_4, a) = l_4'}{I_5(l_5, a) = l_5'}
\]

and \(I_b\) is defined by:

\[
\frac{I_1(l_1, a) = l_1'}{I_{23}((l_2, l_3), a) = (l_2', l_3')} \quad \frac{I_2(l_2, a) = l_2'}{I_3(l_3, a) = l_3'} \quad \frac{I_4(l_4, a) = l_4'}{I_5(l_5, a) = l_5'}
\]

which are clearly equivalent.

- Every probabilistic transition \(((l_1, l_2), l_3), P_a) \in O_a – with \(P_a = (A \times P(T) \times (L_1 \times L_2) \times L_3, P_a))\) – is based on a probabilistic transition in either \(O_1, O_2\) or \(O_3\). From the same basis, we can also construct a probabilistic transition \(((l_1, l_2), l_3), P_b) \in O_b – with \(P_b = (A \times P(T) \times (L_2 \times L_3, P_b)) – such that P_a(\omega) = P_b((i_A \times i_T \times i_L)(\omega))\).
- For \(D\), a similar argument applies as for \(O\).

### 5.3.4 Relation to Probabilistic I/O Automata

Our definition of stochastic I/O automata is based on probabilistic I/O automata (PIOA) as defined by Wu et al. [106]. Note that PIOA actually also provide stochastic timing, because there is a stochastic delay in every state before the next transition is taken. PIOA define a single transition relation for input and output. In (standard) I/O automata, there is only a nondeterministic choice between possible outputs; in probabilistic I/O automata,
Chapter 5. Stochastic Extension: StoCharts

Figure 5.4: A probabilistic I/O automaton. We have denoted transition probabilities with \( p = \ldots \) and rates with rate = \( \ldots \).

Wu et al. replaced all nondeterminism by probabilistic choice. We have split the input and output relations to allow for a combination of nondeterminism and probabilism in the output relation, and we have simplified the input relation. Further, Wu et al. have chosen a presentation that eases comparison to (standard) I/O automata, but blurs the structure of the probability spaces. In addition, we have replaced Wu’s location delays by a different, more general way to describe delays.

A probabilistic I/O automaton is a sextuple \((Q, q^I, E, \Delta, \tau, \delta)\) where

- \( Q \) is a set of states.
- \( q^I \in Q \) is the initial state.
- \( E \) is a set of events – called actions elsewhere in this thesis –, partitioned into input, output, and internal events \( E^{in}, E^{out}, \) and \( E^{int} \).
- \( \Delta \subseteq Q \times E \times Q \) is the transition relation – a combination of our input and output transition relations. \( \Delta \) has to satisfy input enabledness and several finiteness constraints.
- \( \tau : \Delta \rightarrow (0, 1] \) defines transition probabilities.
- \( \delta : Q \rightarrow \mathbb{R}_+^+ \) gives for every state the rate of its (exponential) stochastic delay.

**Example 5.3.1** We want to model a controller for a robot that picks faulty products off a conveyor belt. On average, the robot has to pick a product off every 2 minutes. After some time, the conveyor belt may stop, for example because there is a jam somewhere, or because the personnel takes a break. This happens about once in 60 minutes. When the belt has stopped, it is switched on again after 10 minutes (on average).

We can model this by a simple PIOA as shown in figure 5.4. The annotations for state \( on \) is calculated by: in a state with two transitions with rates \( r \) and \( s \), the combined rate is \( r + s \), and the probability of each transition is proportional to its rate.

To clarify the relation between PIOA and IOSA, we can use the following mapping: Assume given a PIOA \( W = (Q, q^I, E, \Delta, \tau, \delta) \) without internal events (i.e., \( E^{int} = \emptyset \)) and deterministic input (i.e., for every \( q \in Q \) and \( e \in E^{in} \), there is at most one \( q' \in Q \) such that \((q,e,q') \in \Delta\)). Then, we can define a stochastic I/O automaton \( B(W) = (L, T, A, I, O, D, AP, J, l_0) \) with the same behaviours by:

- The set of locations contains two copies of each PIOA state: \( L = Q \times \{0, 1\} \)
5.3. Stochastic Input–Output Automata

There is a timer $t_q$ for each state $q$ in the PIOA: $T = \{t_q \mid q \in Q\}$. Its associated cdf $F_{t_q}$ is the negative exponential distribution with rate $\delta(q)$.

The set of actions is $A = E$, partitioned into $A^\text{in} = E^\text{in}$ and $A^\text{out} = E^\text{out}$.

The input transition relation is defined by:

$$
e \in E^\text{in} (q, e, q') \in \Delta i \in \{0, 1\} \\
I((q, i), e) = (q', 0)$$

The output transition relation is defined by:

$$q \in Q \quad \exists e \in E^\text{out}, q' \in Q : (q, e, q') \in \Delta \\
((q, 1), (A^\text{out} \times P(T) \times L, \mu)) \in O$$

where $\mu$ is probability measure defined by:

$$\mu(\{(e, \{t_{q'}\}, (q', 0))\}) = \tau(q, e, q')$$

and $\mu(\{(e, T, I)\}) = 0$ otherwise.

For all $q \in Q$, there is a delay transition $((q, 0), t_q, (q, 1)) \in D$.

The set of atomic propositions is empty, as PIOA have no propositions: $AP = \emptyset$.

The interpretation of atomic propositions, therefore, is trivial: $J \equiv \emptyset$.

$l_0 = (q', 0)$

In a PIOA, an output transition from state $q$ is enabled only after an exponential delay with rate $\delta(q)$. Therefore, we need two copies of $Q$ in $L$: the copy $(q, 0)$ where the system is waiting, and the copy $(q, 1)$ where output transitions are actually enabled.

Example 5.3.2 When we map the PIOA from example 5.3.1 to an IOSA, we get an automaton with four states and two timers. It could be drawn like figure 5.5.
Mapping PIOA to PTS. To illustrate the relation between IOSA and PIOA further, we also define a mapping from closed PIOA to probabilistic transition systems (PTS), introduced in section 5.3.2. Given a PIOA $W = (Q, q^i, E, \Delta, \tau, \delta)$ with $E^{\text{init}} = \emptyset$, we define the PTS $[W] = (\Sigma, \rightarrow)$ by:

- A PTS state is a triple consisting of a PIOA state, the event just generated, and a time value that indicates how long the PIOA will stay in the state before taking the next transition.
  \[ \Sigma = Q \times E \times \mathbb{R}_0^+ \cup \{\text{Init}\} \]

- The transition relation $\rightarrow$ consists of transitions of the form:
  \[ (q, e, r) \rightarrow (\Sigma, P(Q) \times P(E) \times B, P) \]

where $B$ is the one-dimensional Borel space over $\mathbb{R}_0^+$ and $P$ is the probability measure satisfying the condition

\[ \frac{\tau(q, e, q') \cdot P_{\delta(q')}(B)}{P((q', e, r') \mid r' \in B)} = \frac{\tau(q, e, q') \cdot P_{\delta(q')}(B)}{P_{\delta(q')}(B)} \]

where $P_{\delta(q')}$ is the probability measure that corresponds to the negative exponential cdf with rate $\delta(q')$.

- The initial probability space is:
  \[ (\Sigma, P(Q) \times P(E) \times B, P_0) \]

where $P_0$ is the probability measure satisfying the condition

\[ \frac{B \in B}{P_0((q', e_0, r) \mid r \in B)} = P_{\delta(q')}(B) \]

Proposition 5.3.1 Given a closed PIOA $W$ as above, the following diagram is commutative:

\[ \begin{array}{ccc}
W & \longrightarrow & B(W) \\
\downarrow & & \downarrow \\
[W] & \xrightarrow{\text{isomorph}} & [B(W)]
\end{array} \]

in the sense that there is a mapping from the probability space of complete runs of $[W]$ into the probability space of complete runs of $[B(W)]$ that preserves probabilities for measurable sets of runs. $[B(W)]$ denotes the formal IOSA semantics defined in section 5.3.2.

Proof: The mapping is based on the following mapping of steps: Given a transition $(q, e, r) \rightarrow (q', e', r')$ of $[W]$ and a state $s = ((q, 0), a, T, v)$ of $[B(W)]$ corresponding to $(q, e, r)$, we find a state corresponding to $(q', e', r')$, which is reachable within three steps. The only possible development from $s$ is to wait until $t_q$ expires. Let $\delta = v(t_q)$. Then, we can take the delay transition to $((q, 1), \perp, \emptyset, v - \delta)$. From this state, there is a pure probabilistic choice of output transition (that consists of an output action to be generated,
5.4. Semantics of StoCharts

a set of clocks to be reset, and a new state). We choose the transition \((e', \{t_{q'}\}, q')\) and set the clock \(t_{q'}\) to \(r'\). This leads to the state \(((q', 0), e', \{t_{q'}\}, (v - \delta)[t_{q'} := r'])\). In a formula:

\[
\begin{align*}
((q, 0), a, T, v) & \xrightarrow{\text{time passes}} ((q, 0), \bot, \emptyset, v - \delta) \\
& \xrightarrow{\text{delay transition}} ((q, 1), \bot, \emptyset, v - \delta) \\
& \xrightarrow{\text{output transition}} ((q', 0), e', \{t_{q'}\}, (v - \delta)[t_{q'} := r'])
\end{align*}
\]

Remains to prove that the probabilities are preserved. Note that the probability space of complete runs is defined based on cylinder sets, that are sets of runs with similar (timed) run prefixes, i.e., with the same state sequence (up to some point) and delays in a defined interval. For a formal definition, see Baier et al. [4].

In the above situation, given a left-open interval \(I = (a, b]\), the set of targets \(\{(q', 0), e', r' \mid r' \in I\}\) has probability \(\tau(q, e, q') \cdot P_{\delta(q')}\). From state \(s\), the corresponding set of targets is \(\{(q', 0), e', \{t_{q'}\}, (v - \delta)[t_{q'} := r'] \mid r' \in I\}\), which has the same probability.

5.4 Semantics of StoCharts

The model for a finite collection of StoCharts is a single IOSA. The semantics is based on the following principles:

- It is an extension of the semantics defined in section 2.2.

- A finite collection of StoCharts is mapped to a single IOSA. A location consists of, per StoChart, a configuration (that indicates in which nodes the StoChart is), a set of input events (events that have been received but not yet processed), and a valuation of local variables.

- Sets of StoChart events and pseudo events are actions of the IOSA, so a set of actions is a set of sets of events.

- The input transition relation of the IOSA just collects events by adding them to the set of input events of the appropriate StoChart.

- In the output transition relation, the collected set of events is examined to construct a step. The steps of all StoChart are then executed together in one large output transition.

In the step construction algorithm, there is first a nondeterministic choice (if there are several conflicting possibilities to construct a step), followed by a probabilistic choice (between the possible targets). The order of these choices differs from the order chosen in chapter 4, which is more general, but also leads to a more complex semantics.

- Timeouts as pseudo events. For every P-edge \(i,(j)\) labelled with after\((F_{ij})\), we introduce a timer \(t_{ij}\) that is reset every time one of the source nodes of the P-edge is entered. We add delay transitions for \(t_{ij}\) which add the pseudo-event after\((F_{ij})\) to the set of input events. In the next output transition, this pseudo-event is processed, so the P-edge can be taken once the timer expires.
5.4.1 Step Construction

This section describes how a step is constructed for a single StoChart $SC_i$.

Configurations are defined as for statecharts in section 2.2. The set of all configurations of $SC_i$ is denoted $Conf_i$.

Locations are similar to plain statechart states, but we include variables like with P-statecharts: A state of $SC_i$ is a triple $(C_i, I_i, V_i)$ where $C_i$ is a configuration, $I_i \subseteq Events_i \cup PsEvents_i$ is a set of events and pseudo events (to which the StoChart still has to react), and $V_i : Vars_i \rightarrow \mathbb{Z}$ is a valuation of the variables. The set of all valuations of $SC_i$ is denoted $Val_i$. The validity of guard $g$ in a state depends on the configurations $C_1, \ldots, C_n$ and the valuations $V_1, \ldots, V_n$. We write $(C_1, \ldots, V_{1..n}) \models g$ iff $g$ holds in the state of the collection of P-statecharts.

P-steps and steps. When the system reacts to an event, it first makes some nondeterministic choices, mainly because there may be several possible reactions which are inconsistent. This results in a P-step, a set of P-edges. After that, it chooses probabilistically a step, a set of edges. The step is then executed. A P-step has to obey the same principal constraints as a step in a plain statechart or in a P-statechart:

Enabledness. All P-edges in the P-step must be enabled.

Consistency. All P-edges in the P-step must be pairwise consistent. We introduced the restriction on P-edges ("all edges belonging to a single P-edge shall have the same scope") because consistency depends on the scope.

Priority. We assume a given priority scheme (a partial order on the P-edges) that resolves some of the inconsistencies: If an enabled P-edge $e$ is not in the P-step, then there must be a P-edge in the P-step that is inconsistent with $e$ and does not have lower priority than $e$.

Note that this definition of a priority scheme differs from the definitions in sections 2.2 and 4.3.1. By this definition, all edges belonging to a single P-edge have the same priority, so the difficulty encountered in P-statecharts does not occur here. This is necessary if we want to resolve nondeterministic choice before probabilistic choice.

Maximality. A P-step must be maximal. This means that adding any P-edge leads to a violation of the above conditions.

We now give an algorithm to construct a P-step and a step which – by construction – satisfies the conditions above. Assume that the current location of $SC_i$ is $(C_i, I_i, V_i)$.

Algorithm 5.4.1 Step Construction Algorithm for StoCharts.

1. Calculate the set of enabled P-edges: for $j \in \{1, \ldots, |PEdges_i|\}$,
   \[ j \in EnP(C_i, I_i, V_i) \iff I_i(j).X \subseteq C_i \land I_i(j).e \in I_i \cup \{\bot\} \land (C_{1..n}, V_{1..n}) \models I_i(j).g \]

2. Calculate $PSteps(EnP(C_i, I_i, V_i))$, where $PSteps(E)$ (for $E \subseteq PEdges_i$) contains all maximal, prioritized, consistent sets of P-edges $\subseteq E$. 
3. Choose nondeterministically an element $PT$ of $PSteps(EnP(C_i, I_i, V_i))$. $PT$ is a P-step.

4. Draw samples from the probability spaces of the P-edges in $PT$, resulting in a set of edges $T$. $T$ is a step.

Tasks 2 and 3 of this algorithm can be achieved in a very similar way to the original algorithm for $\text{nextstep}$ (see 2.3.1) to the set $EnP(C_i, I_i, V_i)$.

Task 4 of this algorithm can be described by a discrete probability space over $P\left(\text{Edges}_i\right)$. Its probability measure is the lift of the following probability weight $\mu$ to sets of sets: for any selection of $A_j$ and $Y_j$ (for $j \in PT$),

$$
\mu((\{j, A_j, Y_j\} | j \in PT)) = \prod_{j \in PT} (t_i(j).P)(\{(A_j, Y_j)\})
$$

and $\mu(T) = 0$ otherwise.

The algorithm first resolves the nondeterministic choices: if some inconsistent P-edges are enabled, one of them is chosen. After that, it resolves the probabilistic choices: from every P-edge, one of the possible outcomes is selected, with the corresponding probability. This order is different from the order we described in chapter 4. There, we first resolved probabilistic choices, because we allowed P-edges having edges with different scopes or priorities. This is excluded here (by a restriction on P-edges and a different definition of priorities) to simplify the semantics.

### 5.4.2 Step Execution

After having settled how steps are selected within a single StoChart, we now consider their joint execution in the collection $\{SC_1, \ldots, SC_n\}$. The execution of a step is similar to the execution of a plain or probabilistic statechart step, as probabilistic or stochastic aspects are not involved anymore.

On the level of a single StoChart, executing a step consists of two parts: updating the variables and events occurring in the actions and determining the new state.

### 5.4.3 IOSA Model for a Collection of StoCharts

Assume given a finite collection of StoCharts $(\text{Nodes}_i, \text{Events}_i, \text{Vars}_i, \text{PEdges}_i)_{i=1}^n$. We translate this collection to a single IOSA $(L, T, A, I, O, D, AP, \mathcal{F}, I_0)$ as follows:

- The set of states consists of the “normal” states of the StoChart.

$$
L = \prod_{i=1}^n \text{Conf}_i \times \text{P}(\text{Events}_i \cup \text{PsEvents}_i) \times \text{Val}_i
$$

- For each P-edge $t_i(j)$ with label after($F_{ij}$), there is a timer $t_{ij} \in T$ whose cdf is $F_{ij}$. 

The set of actions is defined (and partitioned) by:

\[ A_{\text{in}} = P \left( \bigcup_{i=1}^{n} \text{Events}_i \right) \]

\[ A_{\text{out}} = P \left( \bigcup_{k=1}^{n} \{ i.e | \exists (j, A, Y) \in \text{Edges}_k : \text{send } i.e \in \text{Actions}_k \wedge i \notin \{1, \ldots, n\} \} \right) \]

- \( I \) is defined by the rule:

\[ E \in A_{\text{in}} \]

\[ (C_i, I_i, V_i)_{i=1}^{n} \xrightarrow{E} (C_i, I_i \cup (E \cap \text{Events}_i), V_i)_{i=1}^{n} \]

- If \( PT_i = (P(\text{Edges}_i), P_i) \in P\text{Steps}(\text{EnP}(C_i, I_i, V_i)) \) for each \( i \), then

\[ ((C_i, I_i, V_i)_{i=1}^{n}, (A_{\text{out}} \times P(T) \times L, \mu)) \in O \]

where \( \mu \) is the probability measure defined by:

\[ \mu(\{(E, T, \text{Execute}(C_{1..n}, T_{1..n}, V_{1..n}))\}) = \prod_{i=1}^{n} P_i(\{T_i\}) \]

where \( E \) is the set of events that are sent to the environment:

\[ E = \bigcup_{k=1}^{n} \{ i.e | \exists (j, A, Y) \in T_k : \text{send } i.e \in A \wedge i \notin \{1, \ldots, n\} \} \]

and \( T = \{t_{ij} | i; (j) \text{ gets enabled} \} \). Let \( \mu(\{(E, T, \omega)\}) = 0 \) otherwise.

- \( D \) is defined by the rule:

\[ X \subseteq C_0 \quad i_0(j) = (X, \text{after}(F_{i_0}), g, P) \in P\text{Edges}_{i_0} \]

\[ I'_{i_0} = I_{i_0} \cup \{ \text{after}(F_{i_0}) \} \quad \forall i \neq i_0 : I'_i = I_i \]

\[ (C_i, V_i, I_i)_{i=1}^{n} \xrightarrow{I'_{i_0}} (C_i, V_i, I'_i)_{i=1}^{n} \]

for some \( i_0 \in \{1, \ldots, n\} \).

- \( l_0 = (C_{0,i}, I_i, V_{0,i})_{i=1}^{n} \) where \( C_{0,i} \) is the initial configuration of \( SC_i \) and \( V_{0,i} \) is its initial valuation.

### 5.5 Example: Automatic Teller Machine

As an example to show how StoCharts can be analysed, we use a performance model of an automatic teller machine (ATM) or cash dispenser. It distributes money to clients that identify themselves with a card and a PIN. We want to find out how fast clients are served.
5.5. Example: Automatic Teller Machine

Figure 5.6: A typical interaction of the ATM with its environment

Figure 5.7: The automatic teller machine behaviour

The model we use consists of three parts: the ATM system and the two external entities Bank and User. Figure 5.6 shows a typical interaction of the ATM with its environment. The ATM is described by the statechart in figure 5.7. Note that the statechart is deterministic and contains neither probabilistic choice nor stochastic timing.

The bank’s behaviour is modelled in a very abstract way as shown in figure 5.3. The bank answers queries about whether a withdraw is allowed and, if desired, withdraws the money from the account. We abstract from the internal process of the bank (database queries and updates) and only model the delays to handle the ATM’s requests. The client, although not part of the system, is also modelled as a StoChart in figure 5.8. After inserting the card, the client expects to enter the PIN and the desired amount (in either order); then, he expects to get money and his card back (also in either order). He needs some time to answer the prompts of the ATM; this is modelled by after-delays in the StoChart. The nodes with an after delay are drawn with half-round sides to distinguish them more easily from “normal” nodes where the user waits for some event.

If the ATM reports a denial of the bank, the client doesn’t expect any money, but still waits to get his card back.\(^5\)

In a full case study, the mean durations and other probabilities should be found by statistical analysis of past behaviour. We have chosen plausible, but unvalidated dura-

---

\(^4\)This can be defined as follows: The P-edge \(e_t(j)\) gets enabled if \(e_t(j).X \subseteq C'_j\) and \(e_t(j).X\) contains a subnode of \(scope(f')\), for some \(f' \in T_i\).

\(^5\)More advanced ATMs sometimes don’t return the card, so this client might eventually wait forever at an advanced ATM.
Chapter 5. Stochastic Extension: StoCharts

Figure 5.8: The client's behaviour
tions and probabilities. We have modelled all after-delays by exponential distributions. If only the mean duration is known from domain analysis, the stochastic behaviour is best reflected by an exponential distribution, since this class of distributions has maximal entropy.

5.5.1 Analysis

Models. We have examined two variants of the ATM example:

1. A simple model with one client, one ATM and the bank.

2. A model with multiple clients, two ATMs and the bank. A mutual exclusion module was added (not modelled as a statechart) to guarantee that the bank served one ATM at a time.

Desired property. The only question we try to answer in this analysis is: How large is the probability that the client gets his money within a specified amount of time?

The answer can be found by checking a number of CSL formulas. See Baier et al. [4, 5] for a full introduction into CSL. The interesting property is:

\[ P_{\geq p}(\Diamond \leq t \text{client takes money}) \]

which means: “The probability that client takes money happens within time \( t \) is \( \geq p \).”

By asking the tools whether a number of these properties hold in the initial state, we can find the answer to our question.

Tools. We have used two tools to analyse the model:

E\textsuperscript{M}C\textsuperscript{2} [50] (Erlangen–Twente Markov Chain Checker, ETMCC) is a model checker for Markov chains. Markov chains are a simpler model than our IOSA: a Markov chain is a closed system, all distributions are exponential, and there is no nondeterminism. If the StoChart satisfies these conditions, (the closure of) its semantic model can in principle be simplified to a Markov chain.

We used TIPPtool [49] as a front end for E\textsuperscript{M}C\textsuperscript{2}. It is a tool that can reduce a specification in a variant of LOTOS [11] to a Markov chain. The StoChart was translated by hand to the input language of TIPPtool. In the first experiment, the model had 113 states; TIPPtool reduced it to a Markov chain with 8 partitions. (Partitions are equivalence classes of states with immediate transitions between each other; to examine the timing behaviour, it is possible to declare these states equivalent, if there is no (relevant) nondeterministic choice.) This calculation lasted 5.1 seconds (all times measured on a Sun UltraSparc 10 with 256 MB RAM under SunOS 5.8). In the second experiment, the model had 2804 states and was reduced to a Markov chain with 65 partitions. (In principle, this are \( 8 \times 8 \) partitions, with one partition split into two because of the mutual exclusion module.) This calculation lasted 48 minutes and 32.2 seconds.
Because TIPPtool can only handle models with some thousands of states, we decided not to try more experiments with more than two ATMs.\footnote{The main difficulty arises from the interleaving of multiple client–ATM dialogues. Without any trouble, one could add more clients, or more ATMs, as long as they interact one after another – but that is not very interesting; if only 2 out of more ATMs can be used at once, one could as well only install 2 ATMs. Note that the mutual exclusion module mentioned in the 2nd experiment is less restrictive: it only sequentialises the communication between the ATM and the bank; the user dialogues of the ATMs are not critical.}

After generating Markov chains, we used \( \text{E-MC}^2 \) to analyse the behaviour of the ATM together with its environment. After the reduction, the actual model checking by \( \text{E-MC}^2 \) was very fast.

**Results.** In figure 5.9, we see the results obtained with \( \text{E-MC}^2 \) for both variants. We observe that in the second scenario, the probabilities are slightly lower than in the first one, with the maximum difference around 40 seconds (0.58 vs. 0.53). The reason is that although the throughput of the bank is higher with more ATMs, the individual customer perceived delay increases, with the bank being the bottleneck.

\( \text{ProVer} \)\cite{107} is a tool that estimates probabilities by discrete event simulation. It can handle different kinds of stochastic distributions; we have used it with exponential and uniform distributions. The tool’s answer is typically “The property is true with confidence 0.99”.

\( \text{ProVer} \) can also directly give an estimate for the probability that \( \Diamond \leq t \) client takes money; this is how we have used \( \text{ProVer} \).

The model was directly entered in the input language of \( \text{ProVer} \). This language actually made it easier to reflect the structure of the statecharts, as it clearly distinguishes...
5.5. Example: Automatic Teller Machine

The checks done by ProVer were very quick. In the case of all exponential distributions, all probabilities were estimated for the first experiment within 48 seconds. For the second experiment, the calculation lasted 2 min 24 sec.

Results. Figure 5.10 shows the approximations obtained with ProVer, where the confidence was set to 0.999. The solid curve corresponds to the second variant (two ATMs) studied with $E\mu-MC^2$, and the numerical results obtained with both tools are very close. This can be considered as a simple sanity check, indicating that (1) the currently manual steps in the tool chain were performed correctly and that (2) ProVer produces rather accurate results. The dashed curve is obtained when replacing all exponential distributions with uniform distributions (over a fixed interval around the mean durations). This choice can be justified by the fact that a user needs some minimum time to react. Since there is now some minimum time before significant behaviour can take place with positive probability, the calculated probability stays below the other plot for small $t$. For large $t$ we are on the other end of the uniform intervals involved, and it is almost sure (except when the bank refuses the transfer) that the money will have been received.

Problems encountered. With $E\mu-MC^2$, we had a hard time in eliminating nondeterminism. In the first model for experiment two, there was a nondeterministic choice which client would be checked first by the bank. We solved this problem by adding the mutual exclusion module.

Another source of nondeterminism was an error we made in the translation of the StoChart to TIPP: it allowed that events “overtake” each other, i.e., the reaction to a latter event could be produced before the reaction to an earlier event. After we had eliminated this error, the translation worked well.
With ProVer, we sometimes got conflicting answers to questions. This stems from the fact that ProVer also tries to answer “yes” or “no” when the probability mentioned in a formula is almost the same as the estimated probability. As the estimate may, in that case, be just below or just above the mentioned probability, we can get conflicting answers when we run ProVer more than once. This problem can be addressed by asking ProVer to give a direct estimate of some probability; this is how we have used ProVer.

5.6 Related Work

Wu, Smolka and Stark [106] have defined probabilistic input–output automata (PIOA). Actually, these automata also include some stochastic elements in the form of a location delay, which is always distributed exponentially. See section 5.3.4 for a more detailed description of PIOA. Our model can be seen as an extension of PIOA; however, we define the composition of two IOSA differently: we choose the leading automaton (i.e., the one taking an output transition) nondeterministically, while [106] chooses it probabilistically.

Balsamo and Simeoni [7] have provided an overview of several performance analysis methods based on UML models. They distinguish mainly queuing network models, stochastic Petri nets and stochastic process algebras.

Gnesi, Latella and Massink [39] have described a stochastic extension of the hierarchical automata model for statecharts [69]. The user defines explicit timers and describes on which transitions timers are checked or set. This requires more bookkeeping from the user’s part than our extension, which keeps the syntax unchanged, while timers are used under the hood. In addition, our extension relies on the existing real-time facilities provided with UML statecharts.

Huszerl, Majzik, Pataricza, Kosmidis and Dal Cin [53] map a statechart to a stochastic reward net, a variant of a GSPN (generalised stochastic Petri net). They annotate timed transitions with a rate that indicates how fast the triggering event is to be expected (on average). The rates are used to give exponential distributions over the transitions. Probabilistic choice is possible using immediate transitions with a weight. We model environment randomness by providing the timing information in the environment model itself, on the side of the event generator.

Lindemann, Thümmler, Klemm, Lohmann and Waldhorst [73] introduce timed events which are very similar to our after pseudo-events. They analyse a fixed collection of UML statecharts by defining a semantics in terms of a generalised semi-Markov process (GSMP). Their tool DSPNexpress automates the analysis for a subclass of GSMP where most stochastic distributions are exponential distributions.

Stochastic Automata (SA) are defined by Pedro D’Argenio [22] as a compositional structure to describe stochastically timed behaviours. Our output transitions are very similar to a SA’s edges: both are labelled with an action and a set of timers as a guard. Timer resets are defined slightly differently: in a SA, every location has a set of timers; upon entering a location, the timers assigned to it are set. We distinguish between input and output; if two SA synchronise, it is not specified which of the two takes the initiative. We also include probabilistic choice in StoCharts; in a SA, there are only stochastic timers and nondeterministic choice. On the other hand, SA allow for asynchronous composition with interleaving; in IOSA, the composition is always synchronous.
5.7. Conclusion

The UML profile for schedulability, performance, and time specification [99] defines annotations to indicate performance and timing of a system modelled in UML. It is intended, among other goals, to make quantitative predictions on time-, schedulability- and performance-related aspects of a real-time system. It is the first part of a larger real-time modelling initiative; for example, one of the next planned steps is a quality of service profile [98]. The profile for schedulability proposes to extend collaboration and activity diagrams to be used for performance analysis. Annotations (the examples in the profile [99] present them using noteboxes) indicate how long an activity lasts or should last. To bring the approaches of the UML profiles and our StoCharts together, one could define P-edges as syntactic sugar to abbreviate a number of edges with these annotations. The UML profile does not define an analysis method, though.

Bernardi, Donatelli, and Merseguer [9] present a translation from a number of statecharts, combined with a sequence diagram, to a GSPN.

Mitton and Holton [81] propose to map a UML statechart with timeout pseudo-events (a simple extension of our after pseudo-events) to the probabilistic process algebra PEPA.

5.7 Conclusion

What we have done. We have introduced StoCharts, an extension of statecharts for quality of service modelling and prediction. We have presented a formal semantics of StoCharts in terms of an IOSA.

This automata model can serve as a basis for stochastic model checking; we have demonstrated this by an ATM example.

The IOSA model also provides means to incorporate communication with the environment and composition with an environment model. This is important, as not only the system under development influences the quality of service, but also the quality provided by the environment.

StoCharts have several advantages over other formalisms for specifying stochastic systems. They are easy to learn for requirements engineers, as they are based on statecharts, which are better known than, for example, continuous time Markov chains.

StoCharts allow the use of hierarchy and parallelism, like in statecharts, which makes the specified behaviours still feasible.

StoCharts provide the engineer with an intuitive and simple syntax and hide the internals with timers under the hood.

Possible improvements. The tool chain used in the model checking example is incomplete; we had to perform some steps manually, namely the translation from a StoChart to the TIPPtool or ProVer input language.
Chapter 6

Case study: RTnet protocol

6.1 Introduction

This chapter illustrates the languages presented in the previous chapters. It does so by describing and analysing parts of the RTnet protocol, a real-time protocol to share a LAN. This protocol is currently under development in the At Home Anywhere (@HA) project [3] of the distributed and embedded systems group (DIES) at the Universiteit Twente. The @HA project aims to integrate home appliances into one coherent distributed architecture. One of its main objectives is to design, build and evaluate one common integrated network for entertainment, control and information, that supports both realtime and non-realtime traffic with a high degree of robustness.

The real-time protocol is a larger case study than the examples of the previous chapters. It also serves to examine whether the methods and techniques presented before scale up. We will see that current probabilistic model checkers are still rather limited.

6.2 The Protocol

Presently, a house has several separated distribution and communication infrastructures: telephone, cable TV and radio, satellite TV, PC network, connection between thermostat and central heating, etc. In most cases, these infrastructures are isolated islands that interconnect only on rare occasions. The @HA project wants to provide a common network to integrate these islands. It divides the infrastructures in three classes:

**Entertainment:** audio, video, games, etc. This class requires high bandwidth and realtime responses. Characteristic for this class is isochronous, streaming data.

**Control:** sensors and actuators, e.g. central heating control, fire detection, burglar alarm. Control uses low bandwidth, but requires a high degree of dependability. Some devices may need realtime services;

**Information:** PC applications, WWW browsing, etc. This class uses bandwidth in bursts of data, and only needs best-effort responses.

The first step to connecting appliances is one common inexpensive infrastructure that supports entertainment, control and information. Most efforts till now concentrate on
Chapter 6. Case study: RTnet protocol

only one class of appliances, mostly entertainment. The @HA project, in contrast, looks for a single network for entertainment, control and information that supports both real-time and non-realtime data, as well as streaming media.

In the following, we will use the terminology: a node is one device that participates in the network. A stream is a one-way communication path from some application running on a node to an application running on a different node. Normally, a node has one or more applications with streams; often, there is a pair of streams between two nodes that communicate with each other, e.g., the TV set and the video recorder: the TV set sends control commands to the recorder, and the recorder sends the requested video data.

The protocol is based on the following principles:

- It is token-based. The token is a symbol for the right to use the network. The stream that has the token may use the network for some predefined time and then passes it on to the next stream.

- The protocol schedules real-time streams according to the earliest deadline first (EDF) scheme: the token is sent to the stream that has the first deadline and is ready. This scheme ensures that all deadlines are reached as long as the network is not overloaded.

- The token contains the deadlines and all other information needed for scheduling and routing. Every node calculates at the end of its time which stream is going to get the token next and for how long. This ensures that the network is used optimally; other token-based protocols have a lower performance because also the nodes that do not need it receive the token.

- A monitor node that holds a copy of the token checks the active node. If the active node dies before it has passed on the token, the monitor will recover the token and keep the network running. The monitor makes the @HA network fault-tolerant. Normally, a node becomes monitor just after it has held the token, until the moment that the active node changes again.

- Network time that is not consumed by the real-time streams can be used to handle non-real-time traffic. In this chapter, we concentrate on the real-time behaviour and omit the handling of non-real-time traffic completely.

There are two extensions of the protocol:

- To add a new node, there is an announcement protocol. From time to time, a node (the announce master) broadcasts a request for nodes. The nodes that would like to participate but aren’t yet listed in the token reply.

  If a node wants to leave the network, or if it wants to add or drop a stream, it has to wait until it gets the token next time and then modifies the token appropriately.

  A node is removed from the token if it appears to have died.

- To ensure that the different nodes interpret deadlines in the same way, they have to synchronise their clocks. From time to time, a special synchronisation stream is started to ensure that the clocks stay in sync. The synchronisation is described by
van den Boom [13]. In this chapter, we have not considered the synchronisation extension. Detailed descriptions of the protocol can be found in the master’s theses of Hattink [47], Wijnberg [105] and van den Boom [13].

6.2.1 States
The original state machines for a node consist of the following main states:

**Offline.** This is the initial state: the node wants to get into the network but is not yet connected. In this state, it either waits for an announcement (if there is already a network running) or it starts a new network.

**Idle.** The node is involved in the network but currently does not hold the token.

**Transmit.** Some stream on this node holds the token and is allowed to use the network.

**Dispatch** (unstable state). The node calculates who is going to get the token next. If it is a stream on the same node, the monitor is informed to keep monitoring. Otherwise, the token is sent to the node of the next stream, the old monitor is released from its task, and this node becomes the monitor.

**Monitor.** The node watches whether the currently active node dies. If it does not get a reaction at the end of the token holding time, it polls the active node. If it still does not get a reaction, the node supposes that the active node has died, removes it from the token and reenters the dispatch state.

**Announce.** The node is running the announcement protocol as announce master. It first broadcasts an announcement and then adds all nodes to the token that request for participation.

**Synchronise.** The node synchronises the local clocks of all participants. It first asks each participant for the time and then broadcasts how every node should adjust its clock.

We will find these states also in the StoChart model; some states are refined.

6.2.2 Messages
The nodes exchange the following messages. Most messages are directed to a single receiver.

**Token(t).** The sender transfers the token $t$ (with all information about streams and deadlines) to the receiver and moves to state Monitor.

**Stop monitoring.** The sender moves from state Dispatch to state Monitor; the receiver (which is the previous monitor) should stop monitoring.
Keep monitoring\((t)\). The sender moves from state Dispatch to state Transmit again; the receiver should keep monitoring longer than planned at first hand. The sender also sends a copy of the new token \(t\) to the receiver, so that it knows how long the sender still wants to hold the token.

In addition, this message may be sent in reply to a Poll.

Poll. The sender, which is in state Monitor, has received no reaction from the receiver, which the sender supposed to be the active node. It may be that a communication was lost or that the receiver has died.

Never received token. The sender has received a Poll message but has not received the token recently. Probably, the Token message was lost.

Please resend token. The sender has received a garbled Token message from the receiver and requests to send it again.

Token announce (broadcast). The sender has found no running RTnet and is about to generate a new token. All receiving nodes should wait. This message ensures that only one token is generated, even if multiple nodes diagnose around the same time that the RTnet has died.

Announcement\((l)\) (broadcast). The sender sends a list \(l\) of all nodes that participate in the RTnet according to the token. All receiving nodes that would like to participate but are not listed, are invited to reply.

Announce reply. The sender has just received an Announcement message and would like to participate in the RTnet.

Sync. The sender is in state Synchronise and asks the receiver to tell the value of its local clock.

Sync reply\((time)\). The sender has received a Sync message and tells its local time.

Sync done\((al)\) (broadcast). The sender is about to leave state Synchronise and notifies every participant in the RTnet how it should adjust its clock. The message therefore contains a list \(al\) of clock adjustments.

6.3 Model of a Node

We have constructed a StoChart that describes the behaviour of a node. It is based on two simple Moore machines (found in Hattink \[47\] and Wijnberg \[105\]). Some differences between these Moore machines can be explained by the development of the protocol. We have added the following points, mostly collected from the textual descriptions:

- The normal responses of the node to the triggers. These responses were partly described in a state–transition table.

- The actions taken to recover from an error. For example, a node sends the Please resend token message when it receives a garbled token.
6.3. Model of a Node

- When the node receives a *Stop monitoring* message, Wijnberg [105] did not specify how to react. On the other hand, Hattink [47] omitted what happens when the node receives an *Announcement* message.

- Some details were only mentioned in the text and not described in the Moore machines. For example, just before a generates a new token it broadcasts a *token announce* message, to avoid that two nodes generate a new token almost at the same time.

- We have decided some small incompletenesses in the protocol description based on a prototype implementation.

- Hierarchy to give a unified description of behaviour that is the same in several states, for example the reaction to a *Sync* message in any of the not-active states.

Our description of a node’s behaviour is the StoChart in figure 6.1. We can see that the node itself contains only a few stochastic delays, mainly around the announce protocol, to reduce the probability that two nodes reply to the announcement master at the same time. For the rest, there is no probabilistic choice in the node.

The StoChart has three top-level states:

**Offline.** This is the initial state, already mentioned in section 6.2.1. In this state, the node listens whether there is already a running RTnet during two times the announcement period. When it receives an *Announcement* message, it proceeds to the state *Not token owner* and reacts to the announcement after a short delay.

Otherwise, if it receives a *Token announce* message, it assumes that some other node is going to start up RTnet and waits for two announcement periods again.

If no other node announces it is going to start up RTnet, this node is doing so, by entering the substate Init. In this substate, a token is created in a way that the announcement stream will be the first stream to be handled, so that the *Announcement* message follows shortly.

**Not token owner.** In this state, the node participates in the RTnet, but some other node holds the token. As soon as it receives the token, it proceeds to state Active. It reacts to enquiries of the announcement (substate *Still online?*) and synchronisation protocol (static reactions).

The substate Monitor is refined to describe what the monitor does if it receives no reaction from the currently active node: after the token holding time THT and a short respite, the monitor sends a poll; if it still receives no reaction, it assumes the node that should be active has died.

In this state, the node tries to ensure that the RTnet has got at least one token. If it has waited for two token receive periods without getting the token, it assumes that the network has died and proceeds to the Offline state, where it eventually may set up a new RTnet. If the node receives a *Token announce* message, it assumes also that the network has died and some other node is setting up a new RTnet.
Active. In this state, the node holds the token. First, some stream is allowed to use the network in substate Transmit until the token holding time THT has expired. Then, the node calculates which stream and node get the token next.

In this state, some action is taken to ensure that the RTnet only has got a single token. If it detects that another node also holds a token, it forgets the token it holds. If the node gets sent a second token, it forgets the second token and tells the monitor of that token to stop.

The network itself is modelled probabilistically: with a high probability, a message sent via the network arrives at the other end; otherwise, it gets lost.

6.3.1 Uses of the Model

The StoChart was used for the following purposes:

- It presents a complete description of the behaviour of a single node. Formerly, the behaviour was only described by a combination of a Moore machine, a state–transition table and many remarks spread over the Master’s theses.

- The correctness of the StoChart was assessed in discussion with a domain expert (Ferdy Hansen, who is working on the protocol).

- The StoChart helped in finding a simplified model for the model checking experiments. After we had chosen that we would not model the extensions of the protocol (announcement and synchronisation), we could easily “cut off” the parts of the statecharts that are related to these extensions. See figure 6.2 for a simplified version of the StoChart.

After that, the simplified StoChart helped in creating the model checker input. While we created the input files for PRISM by hand, we could use the StoChart as a reference. Without the StoChart, we feel that we would have omitted some transitions easily.

6.4 Experiments

We have done the following experiments:

- A simple version of the protocol (without announcements, without synchronisation) was checked with Prism. The checked model only involved two nodes and served to find out how the probability that the network survives depends on the probability that a single communication fails.

- The announcement part of the protocol was checked with ProVer. The probabilities found in the previous experiment were used in this model.
Figure 6.1: The behaviour of a single node in the @HA network. The choice which kind of stream is activated in the Transmit state is left unspecified.
Figure 6.2: Simplified behaviour that served as the basis for the PRISM model
6.4. Experiments

6.4.1 Prism Experiments: Base Protocol

A simple version of the node behaviour is shown in figure 6.2. We have checked a (non-real-time) instantiation of this protocol with two nodes. One node claims 50% of the bandwidth, the other node 33%. Several tries with three or more nodes led to crashes, which are likely caused by a memory overflow. In the model, we have chosen (arbitrarily) to let 95% of the communications succeed.

Since our model includes the possibility that a node is excluded from the RTnet but has no provision to reintegrate it into the RTnet again, the probability that all nodes will stay online forever is 0. To enable the expression of sensible properties about the probability that some node will stay online, we have introduced a counter \( \text{tht sum} \) for the first transmission steps. It saves the sum of the token holding times. \( \text{tht sum} \) is bounded to 7 to save memory; this is enough to count through a complete cycle of the RTnet. There are 231 193 reachable states in the model.

After the construction of the model (2 min 51” on a Sun UltraSparc 10 with 256 MB RAM under SunOS 5.8), the following properties were checked:

**Both nodes reach their deadlines.** If there are no problems with the communication, both nodes reach their deadlines. Therefore, this is a sign that communication works well.

The probability that node 1 misses some of its deadlines in the first cycle is smaller than 0.1, for every fair scheduler. This is proven because the formula

\[ P < 0.1 \left[ \text{tht sum} < 7 \land \text{next deadline} < \text{rest duration} \right] \]  

holds (14 min 57”).

The variable \( \text{next deadline} \) indicates how much time there is left until the next deadline of node 1; the variable \( \text{rest duration} \) indicates how long node 1 still needs the token before the deadline. If node 1 does not participate in the RTnet, \( \text{rest duration} = 0 \). For node 2, this probability is smaller than 0.2, for every fair scheduler (17 min 48”).

**The active node may appear to have died.** If several communication failures occur, the monitor may get the impression that the active node has died. In that case, the monitor reconstructs the token and, as it appears to be the only surviving node, keeps it all for itself. The maximal probability that this happens is in the range \([0.15, 0.2)\) for both nodes, depending on the scheduler. This follows from the formulas:

\[ P < 0.2 \left[ \text{tht sum} < 7 \land \text{rest duration} = 0 \land \text{tht} > 0 \right] \]  

holds (21 min 0”)  

\[ P < 0.15 \left[ \text{tht sum} < 7 \land \text{rest duration} = 0 \land \text{tht} > 0 \right] \]  

is refuted (20 min 54”)

and similar properties for node 2. We require that the token holding time \( \text{tht} \) be \( > 0 \) to avoid the property being checked in intermediary states.

6.4.2 ProVer Experiments: Announcement

We have also modelled the announce protocol. This part of the protocol ensures that a node can reenter the RTnet after contact with that node was lost because of communication failures. We used the results of the previous experiment to estimate the probability
that contact with a node gets lost. For simplicity, we assume that the probabilities are independent.\footnote{In some practical situations, this is not the case; for example, if the token itself gets lost, all nodes are locked out at once.}

We have created a model as follows: once per second, a fixed node (the announce master) sends an announcement to all other nodes. It includes a list of nodes that still participate in the RTnet (according to the token). The probability that any node is no more in the token is set to 3\%. Nodes that are not mentioned react to the announce master. There is, again, 5\% probability that the communication fails. In addition, we have modelled the possibility of collisions between announce replies: if two nodes want to reply almost at the same moment, their messages get lost.

To find the probability that a node reenters the network under these circumstances, we started the system in a state where nodes 2 to 4 are online (in addition to the announce master) and node 1 is offline. The following property was checked in this state: “The probability that node 1 enters the RTnet within 1.25 sec is at least 0.85.”

$$P_{\geq 0.85}[\bigcirc \leq 1.25 \text{ sec in } \text{token}(1)]$$

ProVer reported after 41 sec that the property holds with confidence level 0.9999. The actual probability is around 0.89.

### 6.5 Conclusion

**Limitations of the tools.** Sometimes, PRISM produces an exception, not giving a clear indication which error has happened. When we simplify the model a bit, the properties are checked as expected; so we conclude that the exception occurs in case the model is too complex and causes a memory overflow or similar error.

ProVer, on the other hand, does not include variables. Therefore, it is almost impossible to model a substantial portion of the RTnet in ProVer: In the RTnet, token holding times are calculated during the run and cannot be coded into the model. Also, comparisons between deadlines to find the earliest deadline cannot be modelled in ProVer.

**The StoChart was helpful.** The StoChart in figure 6.1 was mainly helpful in finding an overview over the current specification of the protocol, formerly only specified by a combination of a Moore machine, a state–transition table, and many textual remarks.

The StoChart also was the basis of the (hand-generated) translations to Prism. The quality of the PRISM model is higher because one can translate systematically, based on the formal semantics, instead of writing PRISM code without guidelines.

However, the StoChart did not help directly in finding the model for the announce experiments.

**Properties of the protocol.** We have not found any errors in (our interpretation of) the protocol, where we have corrected the incompletenesses mentioned in section 6.3.
Chapter 7

Conclusion

7.1 Achievements

In the introduction, we stated as the goal of this thesis:

To use statecharts to render model checking more widely usable.

Let us see in how far this goal has been reached.

A property language for real-time statecharts. Statecharts can be used to describe the model to be input into a model checker, because there are already formal semantics of statecharts in terms of a structure understood by a model checker. For example, the semantics we presented in section 2.2 can be easily reduced to a Kripke structure semantics.

For these (i.e., non-probabilistic) real-time statecharts, we have provided a property language that fits nicely with the features of statecharts.

Statecharts include events and actions, so the property language should be able to express properties of actions. They also include states; therefore, the language should provide means to name a state. We use real-time statecharts, so the language should also include some real-time constraints.

The property language ATCTL defined in chapter 3 includes all three elements: Action properties are covered by modal operators with action annotations like $\mu U$. State properties are expressed using propositional logic. Clocks and clock constraints serve to express real-time properties.

For model checking purposes, we defined reductions of ATCTL to the languages it is based on: TCTL and ACTL. As there are model checkers to verify properties stated in these languages, we can also check whether an ATCTL model satisfies an ATCTL property.

So, an engineer can write desired properties of a collection of statecharts in ATCTL and have them checked using one of the existing model checkers.

Probabilistic and stochastic model checking. We have provided two extensions of statecharts: P-statecharts and StoCharts. These extensions are syntactically simple, therefore an engineer that already knows statecharts can use them after a short introduction. Many software engineers already know statecharts as a part of the UML.
The interplay with nondeterminism and priorities raises some subtle semantic issues. We succeeded in defining a formal semantics that tackles these issues while keeping them “under the hood”, so that they do not burden the engineer.

Both extensions map to structures that can be input to a model checker. The engineer may specify desired properties of the statechart in an existing probabilistic (PCTL) or stochastic language (CSL). In this way, it is possible to verify desired probabilistic and stochastic properties of a collection of statecharts.

So, we can conclude that these extensions of statecharts render probabilistic model checking more widely usable: with an easy-to-learn extension of statecharts, an engineer can create a model of a probabilistic or stochastic system without much effort.

Examples and case study. To provide evidence that the extensions of statecharts and property languages can be used in practice, we have included some smaller examples and a somewhat larger case study. The case study also shows the limitations of current probabilistic and stochastic model checking: it was only possible to verify properties of a heavily simplified model.

7.2 Overview of the Mappings and Translations

Figure 7.1 provides an overview of the mappings and translations defined in chapters 2, 3 and 4. We have implemented the mapping from timed output automata to the TCTL model checker Kronos. Figure 7.2 shows the mappings defined in chapter 5. For the main part of the translation from stochastic output automata to E\(^{-}\)MC\(^{2}\), we have reused TIPPtool, but all other translations and reductions must be done by hand.
7.3 Possibilities for Future Research

Some possibilities to continue this research are:

- **More tool support**: Most translation steps in this thesis are formal and therefore can be automated. However, we have only implemented a few of them in TCM. The most desired addition would be presumably the translation from P-statecharts or StoCharts to a model checker.

  Automation can help in translating larger models because it avoids the errors made by hand translations.

- **We have chosen a simple statechart variant with a requirements-level semantics.** This is in accordance with our intended use of the UML as a language for requirements-level models. However, the UML is often used to describe a detailed implementation of a system, so that an implementation-level semantics would fit better. Probabilistic choice and stochastic timing are sensible extensions of the implementation-level semantics, for example to implement a randomised algorithm.

  We expect that it will still be possible to define P-statecharts and StoCharts under an implementation-level semantics, but without doubt, some difficulties will occur. Will these difficulties affect the expressive power of StoCharts and P-statecharts?

- **UML statecharts contain many extra features**, for example history connectors and deferred events. We have mentioned some in section 1.1.2; another list can be found in Eshuis and Wieringa [33]. Which of these features can be included in P-statecharts and StoCharts?

- **The UML profile for quality of service is still under development**; up to now, only the request for proposals [98] is publicly available. StoCharts are, in our opinion,
excellent means for quality of service specification, as far as it concerns soft real-time properties: they provide a simple notation that closely follows the UML, and they are powerful enough to specify soft deadlines and duration distributions. They require no change of the UML metamodel.

• While the logic ATCTL provides means to specify properties that fit with state-charts, it is still rather technical. We could help engineers with little experience in formal languages by providing property patterns. A tool can provide natural-language descriptions of often-used, simple properties – for example, “an error state is unreachable” – so that the user only has to indicate what the error states are. The tool would then translate this desired property to an ATCTL formula.

Dwyer, Avrunin, and Corbett [29] have researched what property patterns would be sensible in the case of plain temporal logic. Are there other patterns that are more relevant in real-time logics? Or for probabilistic or stochastic systems?
Appendix A

Notations and Notions

This appendix explains some of the basic notations and concepts of set and probability theory used in the thesis.

A.1 General Notations

For a set $A$, the powerset of $A$ is denoted $P(A)$.

Given a function $f : A \rightarrow B$ and some subset $A' \subseteq A$, the restriction of $f$ to $A'$ is denoted $f \upharpoonright A'$. This is the function $(f \upharpoonright A') : A' \rightarrow B$ such that $(f \upharpoonright A')(a) = f(a)$ for all $a \in A'$.

**SOS rules.** Plotkin [84] has introduced a style to describe semantic rules called *structural operational semantics* (SOS) rules. An SOS rule has the form

$$\frac{A_1 \quad A_2 \quad \ldots \quad A_n}{C}$$

where $A_1, A_2, \ldots, A_n$ and $C$ are formulas in a suitable logic. The above SOS rule has the meaning:

If the antecedents $A_1$, $A_2$, ..., and $A_n$ hold, then also the consequence $C$ holds.

A.2 Measure Spaces

For an extensive introduction to measure and probability theory, see e.g. the book by Shiryaev [94]. We introduce measure spaces only insofar that probability spaces are a special kind of measure spaces.

Given a set $\Omega$, a system $\mathcal{A}$ of subsets of $\Omega$ is called a *$\sigma$-algebra if:

- $\Omega \in \mathcal{A}$;
- if $A_n \in \mathcal{A}$ (for $n \in \mathbb{N}$), then $\bigcup_{n=1}^{\infty} A_n \in \mathcal{A}$; and
- $A \in \mathcal{A}$ implies $\Omega \setminus A \in \mathcal{A}$. 
A measure is a function \( \mu : \mathcal{A} \rightarrow [0, \infty] \) defined on a \( \sigma \)-algebra \( \mathcal{A} \) with the following properties:

- \( \mu(\emptyset) = 0; \)
- For each sequence of pairwise disjoint sets \( A_1, A_2, \ldots \) with \( A_i \in \mathcal{A} \) for all \( i \), we have

\[
\mu \left( \bigcup_{i=1}^{\infty} A_i \right) = \sum_{i=1}^{\infty} \mu(A_i)
\]

A measure space is a triple \((\Omega, \mathcal{A}, \mu)\) where \( \Omega \) is a set, \( \mathcal{A} \subseteq \mathcal{P}(\Omega) \) is a \( \sigma \)-algebra over \( \Omega \) and \( \mu : \mathcal{A} \rightarrow [0, \infty] \) is a measure.

**Borel-\( \sigma \)-algebra.** An often used \( \sigma \)-algebra is the so-called Borel-\( \sigma \)-algebra \( \mathcal{B} \) over the real numbers \( \mathbb{R} \). It can be defined as the smallest \( \sigma \)-algebra that contains the half-open intervals \([r, s)\), where \( r \) and \( s \) range over \( \mathbb{R} \).

The Borel-\( \sigma \)-algebra can also be defined for higher-dimensional spaces. The \( n \)-dimensional Borel-\( \sigma \)-algebra \( \mathcal{B}^n \) is the smallest \( \sigma \)-algebra over \( \mathbb{R}^n \) that contains the left-open intervals \([r, s)\), where \( r \) and \( s \) range over \( \mathbb{R}^n \). These intervals are defined by:

\[
[r, s) = \{(t_1, t_2, \ldots, t_n) \mid r_1 \leq t_1 < s_1 \land r_2 \leq t_2 < s_2 \land \cdots \land r_n \leq t_n < s_n\}
\]

**Mapping a measure space.** A function from a measure space to a set can be used to define a measure space over the latter.

**Lemma A.2.1** Given a measure space \((\Omega, \mathcal{A}, \mu)\), a set \( \Omega' \) and a function \( T : \Omega \rightarrow \Omega' \). Then, we can define a measure space \((\Omega', \mathcal{A}', \mu')\) over \( \Omega' \) by:

\[
\mathcal{A}' = T^*(\mathcal{A}) = \{ O \subseteq \Omega' \mid T^{-1}(O) \in \mathcal{A} \}
\]

\[
\mu' = T(\mu) = \mu \circ T^{-1}
\]

**Proof:** We rely on the property that for any sequence \( A'_1, A'_2, \ldots \in \mathcal{A}' \) (for \( i \in \mathbb{N} \)),

\[
\bigcup_{i=1}^{\infty} T^{-1}(A_i) = T^{-1}\left( \bigcup_{i=1}^{\infty} A_i \right) \tag{*}
\]

\( \mathcal{A}' \) is a \( \sigma \)-algebra: (i) \( \Omega' \subseteq \mathcal{A}' \), as \( T^{-1}(\Omega') = \Omega \in \mathcal{A} \). (ii) Given a sequence \( A'_1, A'_2, \ldots \in \mathcal{A}' \), we have \( T^{-1}(A_i) \in \mathcal{A} \) for all \( i \). Because of (*), \( \bigcup_{i=1}^{\infty} A'_i \in \mathcal{A}' \). (iii) If \( A' \in \mathcal{A}' \), then \( T^{-1}(\Omega' \setminus A') = \Omega \setminus T^{-1}(A') \in \mathcal{A} \), therefore \( \Omega' \setminus A' \in \mathcal{A}' \).

\( \mu' \) is a measure: (i) \( \mu'(\emptyset) = \mu \circ T^{-1}(\emptyset) = \mu(\emptyset) = 0 \). (ii) Given a sequence \( A'_1, A'_2, \ldots \) of pairwise disjoint subsets of \( \mathcal{A}' \), note that \( T^{-1}(A_1), T^{-1}(A_2), \ldots \) are pairwise disjoint again. We then use (*) to prove \( \mu' \left( \bigcup_{i=1}^{\infty} A_i \right) = \sum_{i=1}^{\infty} \mu'(A'_i) \).
A.3 Probability Spaces

A probability measure is a measure \( \mu : \mathcal{A} \to \mathbb{R}_0^+ \) (where \( \mathcal{A} \subseteq \mathcal{P}(\Omega) \)) with the property that \( \mu(\Omega) = 1 \).

A probability space is a measure space \((\Omega, \mathcal{A}, \mu)\) whose measure \( \mu \) is a probability measure.

This definition has the following intuitive background: \( \Omega \) is the set of possible outcomes of a probabilistic choice process. A subset \( A \subseteq \Omega \) has probability \( \mu(A) \), as far as a probability can be assigned to \( A \). The \( \sigma \)-algebra \( \mathcal{A} \) describes to which sets such a probability can be assigned. The fact that \( \mu \) is a measure ensures that the probability assignments are consistent with each other.

Discrete probability spaces. A probability space \((\Omega, \mathcal{A}, \mu)\) is called discrete, if every subset of \( \Omega \) is measurable: \( \mathcal{A} = \mathcal{P}(\Omega) \). For discrete probability spaces, we sometimes write \((\Omega, \mu)\) instead of \((\Omega, \mathcal{P}(\Omega), \mu)\). For a set \( \Omega \), the set of all discrete probability spaces over \( \Omega \) is denoted \( \text{Prob}(\Omega) \).

Probability spaces over the real numbers. Since we model real-time like the real numbers \( \mathbb{R} \), we use probability spaces over \( \mathbb{R} \) for stochastic timing. One normally uses the Borel-\( \sigma \)-algebra \( \mathcal{B} \), leading to probability spaces of the form \((\mathbb{R}, \mathcal{B}, \mu)\).

One can define the probability measure \( \mu \) using a cumulative distribution function (cdf). A cdf is a function \( F : \mathbb{R} \to [0, 1] \) which describes the probability measure \( \mu \) of a space \((\mathbb{R}, \mathcal{B}, \mu)\) by:

\[
F(x) = \mu((\infty, x]) \quad \text{or, equivalently,} \quad \mu([r, s]) = F(s) - F(r)
\]

i.e., \( F(x) \) is the probability that the outcome is \( \leq x \). A cdf is monotonically nondecreasing and right-continuous; \( \lim_{x \to -\infty} F(x) = 0 \) and \( \lim_{x \to +\infty} F(x) = 1 \).

Sometimes, we define a cdf only on the nonnegative reals: \( F : \mathbb{R}_0^+ \to [0, 1] \). In that case, it is understood that \( F(x) = 0 \) for \( x < 0 \).

Given a cdf \( F \), the \( n \)-percentile is the real number \( r \) such that \( F(r) = n\% \). The median is the 50-percentile, i.e., the real number \( r \) such that \( F(r) = 0.5 \).

\[\text{A random variable is another way to describe a probability space over the real numbers. Its relation with our normal notation is: if } X \text{ is a random variable, then } \mu(A) \text{ (for some set } A \subseteq \mathbb{R}) \text{ is } P(X \in A), \text{ the probability that the outcome of the probabilistic selection for } X \text{ is in } A. \text{ We will not use random variables in this thesis any further.}\]
Appendix B

Abstract

Statecharts are a graphical language to describe the behaviour of a system. For example, in the UML, a statechart can be used to describe the behaviour of an object. Model checking is a method to verify automatically whether a system satisfies some desired property.

The goal of this thesis is:

To use statecharts to render model checking more widely usable.

We show this in two respects: For real-time statecharts, we provide a property language that fits nicely with the features of statecharts. For probabilistic model checking, we provide an extension of statecharts as input language.

A property language for statecharts. The logic ATCTL is a temporal logic to express properties about actions, states and real time. It is based on CTL. Its intention is to make it easier to express the desired properties of a system described by statecharts. The thesis defines reductions to ACTL and TCTL, so that existing model checkers can be used to verify a system against an property in ATCTL.

ATCTL extends CTL with concurrent actions and real time. It is based on earlier extensions of CTL by De Nicola and Vaandrager (ACTL) and Alur et al. (TCTL); see figure B.1. This makes it easier to adapt existing model checkers to ATCTL. To show that we can check models specified using statecharts against properties specified in ATCTL, we give a small example using the model checker Kronos [14, 108].

\[
\begin{array}{ccc}
\text{CTL} & \text{ACTL} & \text{TCTL} \\
\text{De Nicola, Vaandrager [28]} & \text{Alur, Courcoubetis, Dill [2]} & \text{section 3.8, section 3.10}
\end{array}
\]

Figure B.1: Reductions of ATCTL
Probabilistic choice. The two other extensions are concerned with probabilities. Traditional statecharts do not include probabilities, although Harel mentioned the possibility of probabilistic choice in his seminal article [43].

In probabilistic choice, it is uncertain which possibility out of a discrete set the system will take. A probability space over this set describes how likely the single possibilities are. For example, the roll of a die can be seen as the result of a probabilistic choice. The extension of statecharts with probabilistic choice introduces means to specify system randomness, and to verify probabilistic temporal properties over P-statecharts. For an example of a P-statechart, see figure B.2. The figure depicts the behaviour when playing with an unreliable, but fair coin: the event “toss” may or may not be ignored. If the system reacts, it generates “heads” or “tails”, each with 50% chance; this is indicated by the symbol $P$ (a so-called P-pseudonode). If the output is heads, the system stops playing. It is unspecified how (un)reliable the system is.

Although the natural model for P-statecharts would be bundle probabilistic transition systems (BPTS), we defined a semantics in terms of a Markov decision process (MDP), so that we can make use of existing probabilistic model checkers for MDPs.

Properties of interest for probabilistic statecharts are expressed in PCTL, a probabilistic variant of CTL for processes that exhibit both non-determinism and probabilities. Verification is performed using the model checker PRISM. Two model checking examples show the feasibility of the suggested approach.

Stochastic timing. In stochastic timing, it is uncertain how long a specific process will take. The timing is described by a probability space over the (nonnegative) real numbers. For example, how long somebody phones on average is a stochastic process.

The stochastic extension of statecharts, called StoCharts, allows to quantify the time between events according to a stochastic distribution. Also in this case, the thesis defines a formal semantics of a collection of StoCharts in terms of a structure that can be analysed using existing tools.

Figure B.3 shows a StoChart for the workflow of a car damage assessor. His task is to assess on behalf of an insurance company whether a damaged car should be repaired and whether the garage offers an acceptable price for the repair. The StoChart shows that processes may take a shorter or longer time, depending on several circumstances that we do not model. Labels of the form $after(\sim n \text{ min})$ show that the average delay is $n$ minutes. As we can see, StoCharts also encompass a (slightly restricted) form of probabilistic choice.

Stochastic statecharts have a semantics in stochastic input–output automata. The possible analysis methods depend on the stochastic timing distributions used: if all distri-
Figure B.3: Example StoChart

Distributions are exponential (and there is no nondeterminism), the IOSA can be reduced to a Markov chain. There are several tools to analyse Markov chains, for example the model checker Ei-MC\(^2\). The tool ProVer is able to deal with more general distributions, but is not a model checker in the strict sense, as it uses discrete event simulation to test desired properties of a stochastic system. We have used both tools to analyse the performance of the model of an automatic teller machine.

Both extensions of statecharts provide software engineers (of which many already know basic statecharts) with an easy way to describe probabilistic resp. stochastic systems so that they can be analysed formally.

**Case study: RTnet protocol.** The @HA project of the DIES group at the Universiteit Twente aims to integrate home appliances into one coherent distributed architecture. One of its main objectives is to design, build and evaluate one common integrated network for entertainment (e.g., video), control (e.g., heating) and information (e.g., WWW). The HOTnet or RTnet protocol is a real-time communication protocol designed to be used in the @HA network. We investigate how error-prone the network is, given that communication between nodes fails with some probability.
Bijlage C

Samenvatting

Statecharts zijn een grafische tool om het gedrag van een systeem te beschrijven. In de kontekst van de UML kan een statechart bij voorbeeld gebruikt worden om het gedrag van een object te beschrijven. Model checking is een methode om automatisch te controleren of een systeem een bepaalde eigenschap heeft.

Het doel van dit proefschrift is:

Met behulp van statecharts, model checking gebruiksvriendelijker maken.

Wij doen dit op twee manieren: Voor statecharts met meetbare tijdsaanduiding (“real-time” statecharts) bieden we een eigenschappentaal aan die goed aansluit bij de voorzieningen van statecharts. Voor probabilistisch model checking bieden we een uitbreiding van statecharts aan, die als invoertaal gebruikt worden kan.

Een eigenschappentaal voor statecharts. De logica ATCTL is een temporele logica die eigenschappen van acties, toestanden, en meetbare tijd uitdrukken kan. Zij is gebaseerd op CTL en is bedoeld om de gewenste eigenschappen van een met statecharts beschreven systeem eenvoudiger uit te kunnen drukken. Dit proefschrift definiëert reducties naar de bestaande logica’s ACTL en TCTL, waarvoor al model checkers bestaan. Deze kunnen daaronder hergebruikt worden om te controleren of een systeem een ATCTL-eigenschap heeft.

ATCTL breidt CTL uit met (parallele) acties en meetbare tijd (real-time). De logica bouwt voort op bestaande uitbreidingen van CTL door De Nicola en Vaandrager (ACTL)

\[ \text{sectie 3.8} \rightarrow \text{ATCTL} \rightarrow \text{sectie 3.10} \]

\[ \text{TCTL} \rightarrow \text{ACTL} \]

\[ \text{Alur, Courcoubetis, Dill [2]} \rightarrow \text{De Nicola, Vaandrager [28]} \rightarrow \text{CTL} \]

Figuur C.1: Reducties van ATCTL
en Alur et al. (TCTL); zie figuur C.1. Daarom is het eenvoudig mogelijk bestaande model checkers uit te breiden voor ATCTL. Om aan te tonen dat het ook daadwerkelijk mogelijk is modellen van statecharts op ATCTL-eigenschappen te controleren, voegen we een klein voorbeeld toe dat de model checker Kronos gebruikt [14, 108].

**Probabilistische keuze.** De twee andere uitbreidingen liggen op het gebied van waarschijnlijkheidsrekening. Traditioneel kan men met statecharts geen waarschijnlijkheden opgeven, alhoewel Harel al in zijn overzichtsartikel [43] op deze mogelijke uitbreiding wees.

Met „probabilistische keuze“ bedoelen we de situatie dat het in een systeem niet bepaald is welke keuze uit een discrete verzameling het systeem zal nemen. Bij voorbeeld kan men de waarde van een dobbelsteen als een uitkomst van probabilistische keuze beschouwen. De uitbreiding van statecharts met probabilistische keuze introduceert middelen om systeem-interne toeval te beschrijven. Het is mogelijk om gewenste probabilistische eigenschappen van P-statecharts op te geven. Een voorbeeld van een P-statechart staat in figuur C.2. De P-statechart laat het verloop van een spel met een onbetrouwbare, maar eerlijke munt zien de gebeurtenis „werp” kan wel of niet genegeerd worden. Als het systeem reageert, antwoordt het met „kop” of „munt”, elk met 50 % kans; dit wordt met het symbool $P$, de zogenaamde P-pseudonode, aangegeven. Als de reactie „kop” is, stopt het systeem met spelen. De P-statechart geeft niet aan hoe (on)betrouwbaar het systeem is.

Het meest voor de hand liggende wiskundige model voor P-statecharts zou een structuur zijn met de naam “bundle probabilistic transition system” (BPTS). Wij hebben echter gekozen Markov-beslisprocessen (MBPs) als model te gebruiken, omdat er model checkers voor MBPs bestaan.

Relevante eigenschappen van probabilistische statecharts worden opgeschreven in PCTL, een probabilistische variant van CTL voor systemen die zowel nondeterminisme als ook waarschijnlijkheden bevatten. Wij gebruiken de model checker PRISM. Twee voorbeelden laten zien dat deze benadering werkbaar is.

**Stochastische tijdsverdeling.** Met „stochastische tijdsverdeling” bedoelen we de situatie dat het in een systeem niet bepaald is hoe lang een specifieke activiteit zal duren. Bij voorbeeld kan men de duur van een telefoongesprek niet (vooraf) vastleggen.

De stochastische uitbreiding van statecharts, StoCharts genoemd, staat de gebruiker toe de tijdsduur tussen twee gebeurtenissen met een stochastische distributie te beschrijven. Ook hier definieert het proefschrift een formele semantiek van een collectie van StoCharts, gebaseerd op een model dat met bestaande gereedschappen kan analyseren.
Afbeelding C.3 laat een StoChart zien voor de werkstroom van een autoschadeadviseur. Zijn taak is het, om in opdracht van een verzekeringsmaatschappij te onderzoeken of een autoschade moet worden hersteld en of de garage een redelijke prijs voor de reparatie vraagt. De StoChart laat zien dat verschillende stappen in de werkstroom korter of langer kunnen duren, afhankelijk van omstandigheden die we niet in het model hebben opgenomen. Bijschriften als after(EXP[1 min]) geven aan dat de gemiddelde duur 1 minuut is en hoe de daadwerkelijke duur van de gemiddelde kan afwijken (namelijk volgens een EXPonentiële distributie). We kunnen uit het voorbeeld opmaken dat StoCharts ook een (beperkte) vorm van probabilistische keuze toelaten.

Het wiskundige model achter stochastische statecharts zijn stochastische input-output-automaten (afgekort IOSA). De analysemethoden die we kunnen toepassen hangen van de distributies af: indien alle distributies exponentieel zijn (en er geen nondeterminisme overblijft), kan de IOSA gereduceerd worden tot een Markovketen. Er bestaan een aantal gereedschappen om Markovketens te onderzoeken, bij voorbeeld de model checker E-MC². – Het programma ProVer kan ook met andere distributies omgaan, maar is strict genomen geen model checker, omdat het discrete-event-simulatie gebruikt om gewenste eigenschappen van een stochastisch systeem te testen. Wij hebben deze twee gereedschappen gebruikt om de gemiddelde snelheid van een PIN-automaat te onderzoeken.

Met deze uitbreidingen van statecharts kan een software-ontwerper (die vaak al statecharts kent) op eenvoudige wijze probabilistische en stochastische systemen beschrijven en formeel laten analyseren.

Casusstudie: RTnet protocol. Het @HA-project van de DIES-groep aan de Universiteit Twente wil huishoudelijke apparaten tot een samenhangende gedistribueerde architectuur verbinden. Een van de hoofddoelen van dit project is het een gemeenschappelijk netwerk voor ontspanning (b.v. video), besturing (b.v. verwarming) en informatie (b.v. video).
WWW) te ontwerpen, bouwen en te evalueren. We onderzoeken hoe foutgevoelig het netwerk is, onder voorwaarde dat de communicatie tussen apparaten met een bepaalde waarschijnlijkheid faalt.
Appendix D

Curriculum Vitae

David N. Jansen was born on May 9, 1971 in Switzerland. He lived in the Netherlands as a child during four years, but attended most schools in Switzerland. He studied Mathematics, Computer Science and Comparative Linguistics at the Universität Bern in Switzerland from 1990 to 1997. His Master’s thesis in Theoretical Computer Science, entitled Ontologische Aspekte Expliziter Mathematik, shows the inconsistency of some straightforward extensions of the theory EET. He became a Ph. D. student at Universiteit Twente in 1998 and was supervised by Roel Wieringa.
Appendix D. Curriculum Vitae
Bibliography


Index

@HA, see At home anywhere

Action location, 47
Action logic, 40
Action step, 42
Action symbol, 40
Action term, 41
ACTL, 51
after(\(F\)), 96
At home anywhere, 121
ATM, see Automatic teller machine
Automatic teller machine, 101, 112
Auxiliary state, 76

Backgammon, 73
Borel-\(\sigma\)-algebra, 136
BPTS, see Bundle probabilistic transition system
Branching temporal logic, 14
Bundle probabilistic transition system, 72

Car damage assessor, 11, 98
cdf, see Cumulative distribution function
Clock, 41
Clock constraint, 41
Clock reset step, 57
Closed system, 23
Communication, 29
Compatible automata, 22, 104
Complete run, 22, 76
Composition, 22, 104
Computation tree logic, 38
Configuration, 26
Consistency, 27, 31, 71
Continuous real time, 14
CTL, see Computation tree logic
Cumulative distribution function, 137

Damage assessor, see Car damage assessor
Default completion, 28
Descendant, 24
Discrete probability space, 137
Discrete real time, 14
Double until condition, 45, 53
Earliest deadline first, 122
Edge, 68, 97
Enabled, 26, 68
Environmental randomness, 16
\(E^\text{-MC}_2\), 115
Fair run, 76
Fair scheduler, 84
Fruit machine, see Gambling machine

Gambling machine, 88
Hawks and doves, 85
HOTnet, see RTnet

Implementation level, 12
Input enabledness, 100
Input–output automaton, 21, 100
Interlevel transition, 97
Internal travel office, 59
IOSA, see Stochastic input–output automaton

Linear temporal logic, 15
Local variable, 31
Location, 42

Markov decision process, 75
MDP, see Markov decision process
Measure, 136
Measure space, 135
Median, 137
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model checking</td>
<td>17, 40</td>
</tr>
<tr>
<td>Monitor</td>
<td>122</td>
</tr>
<tr>
<td>n-percentile</td>
<td>137</td>
</tr>
<tr>
<td>N-similarity</td>
<td>54</td>
</tr>
<tr>
<td>Open system</td>
<td>23</td>
</tr>
<tr>
<td>Output automaton</td>
<td>21</td>
</tr>
<tr>
<td>P-edge</td>
<td>67, 97</td>
</tr>
<tr>
<td>P-pseudonode</td>
<td>67, 98</td>
</tr>
<tr>
<td>P-statechart</td>
<td>66</td>
</tr>
<tr>
<td>P-step</td>
<td>110</td>
</tr>
<tr>
<td>Partial time stamp</td>
<td>41</td>
</tr>
<tr>
<td>PCTL</td>
<td>83</td>
</tr>
<tr>
<td>Perfect technology</td>
<td>13</td>
</tr>
<tr>
<td>PIOA, see Probabilistic I/O automaton</td>
<td></td>
</tr>
<tr>
<td>Position</td>
<td>43</td>
</tr>
<tr>
<td>Priority</td>
<td>27, 70</td>
</tr>
<tr>
<td>Probabilistic choice</td>
<td>16, 65</td>
</tr>
<tr>
<td>Probabilistic I/O automaton</td>
<td>105, 118</td>
</tr>
<tr>
<td>Probabilistic transition system</td>
<td>102</td>
</tr>
<tr>
<td>Probability measure</td>
<td>137</td>
</tr>
<tr>
<td>Probability space</td>
<td>137</td>
</tr>
<tr>
<td>ProVer</td>
<td>116</td>
</tr>
<tr>
<td>Pseudo event</td>
<td>96</td>
</tr>
<tr>
<td>PTS, see Probabilistic transition system</td>
<td></td>
</tr>
<tr>
<td>Random variable</td>
<td>137</td>
</tr>
<tr>
<td>Real time</td>
<td>14</td>
</tr>
<tr>
<td>Real-time statecharts</td>
<td>31</td>
</tr>
<tr>
<td>Region automaton</td>
<td>54</td>
</tr>
<tr>
<td>Requirements level</td>
<td>12</td>
</tr>
<tr>
<td>RTnet</td>
<td>121</td>
</tr>
<tr>
<td>Run</td>
<td>22, 76</td>
</tr>
<tr>
<td>$\sigma$-algebra</td>
<td>135</td>
</tr>
<tr>
<td>Scheduler</td>
<td>84</td>
</tr>
<tr>
<td>Scope</td>
<td>26, 69</td>
</tr>
<tr>
<td>Scope of a P-edge</td>
<td>97</td>
</tr>
<tr>
<td>Simple until condition</td>
<td>45, 52</td>
</tr>
<tr>
<td>Soft real-time constraint</td>
<td>95</td>
</tr>
<tr>
<td>SOS, see Structural operational semantics</td>
<td></td>
</tr>
<tr>
<td>State</td>
<td>42</td>
</tr>
<tr>
<td>Statecharts</td>
<td>11, 23</td>
</tr>
<tr>
<td>Step</td>
<td>27, 42, 69, 76, 110</td>
</tr>
<tr>
<td>StoChart</td>
<td>96</td>
</tr>
<tr>
<td>Stochastic automaton</td>
<td>118</td>
</tr>
<tr>
<td>Stochastic input–output automaton</td>
<td>100</td>
</tr>
<tr>
<td>Stochastic timing</td>
<td>16, 95</td>
</tr>
<tr>
<td>Strictly fair scheduler</td>
<td>84</td>
</tr>
<tr>
<td>Structural isomorphism</td>
<td>105</td>
</tr>
<tr>
<td>Structural operational semantics</td>
<td>135</td>
</tr>
<tr>
<td>System randomness</td>
<td>17</td>
</tr>
<tr>
<td>TCM, see Toolkit for conceptual modelling</td>
<td></td>
</tr>
<tr>
<td>TCTL</td>
<td>46</td>
</tr>
<tr>
<td>Time stamp</td>
<td>41</td>
</tr>
<tr>
<td>Time step</td>
<td>42</td>
</tr>
<tr>
<td>Time valuation</td>
<td>100</td>
</tr>
<tr>
<td>Timed input–output automaton</td>
<td>42</td>
</tr>
<tr>
<td>Timed output automaton</td>
<td>43</td>
</tr>
<tr>
<td>Timer</td>
<td>100</td>
</tr>
<tr>
<td>Token</td>
<td>122</td>
</tr>
<tr>
<td>Toolkit for conceptual modelling</td>
<td>33</td>
</tr>
<tr>
<td>Travel office, see Internal travel office</td>
<td></td>
</tr>
<tr>
<td>UML, see Unified Modelling Language</td>
<td></td>
</tr>
<tr>
<td>Unified Modelling Language</td>
<td>32, 119</td>
</tr>
<tr>
<td>Unlabelled output automaton</td>
<td>39</td>
</tr>
<tr>
<td>Unreliable, but fair coin</td>
<td>67</td>
</tr>
<tr>
<td>Until condition</td>
<td>39, 45, 46</td>
</tr>
<tr>
<td>Zeno behaviour</td>
<td>43</td>
</tr>
</tbody>
</table>
SIKS Dissertatiereeks

1998

98–1 Johannes Fredericus Philippus van den Akker. *DEGAS: an active, temporal database of autonomous objects*. Universiteit van Amsterdam / CWI.


1999


2000


2000–10 Nicolaas Johannes Nes. *Image database management system design considerations, algorithms and architecture*. Universiteit van Amsterdam / CWI.


2001

2001–1 Silja Renooij. *Qualitative approaches to quantifying probabilistic networks*. Universiteit Utrecht.


2002


2003


CTIT Ph. D.-thesis Series

All theses have been defended at Universiteit Twente.

1994

1995
95–02 Aiko Pras. Network management architectures.
95–03 Gerhard Jan Heijenk. Connectionless communications using the asynchronous transfer mode.
95–04 Marten Jelle van Sinderen. On the design of application protocols.
95–05 Harm Henderikus Cornelis Kremer. Protocol implementation: bridging the gap between architecture and realization.
95–06 Xinli Hou. A network control architecture for advanced B-ISDN services.
95–07 Jan Herman Laarhuis. Multichannel interconnection in all-optical networks.

1996
96–08 Elske Heeren. Technology support for collaborative distance learning.
96–09 Joost-Pieter Katoen. Quantitative and qualitative extensions of event structures.

1997
97–12 Peter Leydekkers. Multimedia services in open distributed telecommunications environments.
97–13 Willem Theodoor Maria Kars. Process-algebraic transformation in context.
97–14 Willem Nico Borst. Construction of engineering ontologies for knowledge sharing and reuse.
97–16 Maurice van Keulen. Formal operation definition in object-oriented databases.
97–19 Frank Tempelman. Structured representations in case-based reasoning.
1998


98–18 Dirk Anton Cornelis Quartel. *Action relations: basic design concepts for behaviour modelling and refinement*.

98–21 Alexander Wilhelmus Heerink. *Ins and outs in refusal testing*. Eindhoven: IPA.

98–22 Jan Marcel Rijnders. *Optical signal processing: a novel approach to modifying digital information in the optical domain*.


1999

99–24 David Spelt. *Verification support for object database design*.


99–26 Arjen Paul de Vries. *Content and multimedia database management systems*.

99–27 Johannes Andreas Erik Oltmans. *A knowledge-based approach to robust parsing*.

99–28 Doutje Tjitske van Veen. *All-optical multiwavelength ring networks*.

2000

00–01 Pieter Tjerk de Boer. *Analysis and efficient simulation of queueing models of telecommunication systems*.


2001


01–33 Ciro de Barros Barbosa. *Frameworks for implementing protocols: a model based approach*.

01–34 Theodorus Cornelis Ruijs. *Towards effective model checking*.


01–38 Cléver Ricardo Guareis de Farias. *Architectural design of groupware systems: a component-based approach*.

2002


02–41 Henk Ernst Blok. *Database optimization aspects for information retrieval.* Enschede: Twente University Press.

02–42 Georgios Karagiannis. *Scalability and congestion control in broadband intelligent and mobile networks.* Enschede: Twente University Press.

02–43 Henderikus van Rein. *Extending model checking to object process validation.*


02–47 Marcos Rogério Salvador. *MAC protocols for optical packet-switched WDM rings.*

02–48 Milan Petković. *Content-based video retrieval supported by database technology.*

2003


03–49 Hartmut Peter Benz. *Casual multimedia process annotations, COMPA.*


03–51 Szilárd Kiss. *3D Character centred online editing modalities for VRML-based virtual environments.* Enschede: Neslia Paniculata.

03–52 José Laurindo Campos dos Santos. *A biodiversity information system in an open data/metadatabase architecture.*


