Formal techniques for verification of complex real-time systems

Formal Techniques for Verification of Complex Real-Time Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. R.A. van Santen, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op dinsdag 8 oktober 2002 om 16.00 uur

door

Marc Constantijn Willem Geilen

geboren te Sittard

Dit proefschrift is goedgekeurd door de promotoren:

prof.ir. M.P.J. Stevens
en
prof.dr. J.C.M. Baeten

Copromotor:
dr.ir. J.P.M. Voeten

Druk: Universiteitsdrukkerij Technische Universiteit Eindhoven

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Geilen, Marc C.W.

Formal techniques for verification of complex real-time systems / by Marc C.W. Geilen. - Eindhoven: Technische Universiteit Eindhoven, 2002.
Proefschrift. - ISBN 90-386-1930-8
NUR 992
Trefw.: formele talen / real-time computers / temporele logica / discrete simulatie / modelcontrole / toestandsruimte-analyse.
Subject headings: formal verification / formal specification / real-time-systems / temporal logic / discrete event simulation / state-space methods.

Summary

Increasing complexity in real-time distributed systems calls for techniques to automate and support their design. In particular concurrent and communicating systems are hard to design correctly. This is especially important for embedded and safety-critical systems. Formal techniques are important aids for the construction of tools that support and automate parts of design and verification and reduce the amount of resources spent on it. Problems detected early in the design process can be eliminated without the excessive costs associated with faults that are dealt with at the end of a design. Although formal verification methods are designed to attack these types of problems, they often exhibit characteristics that clash with the nature of the initial design process. In practice in the early stages of the design, a popular and effective tool to study a design is simulation. It is flexible, easy to use and interactive. Its lack of formality however inhibits many automated design and verification steps. As a design evolves, the need for more thorough analyses emerges. The design and its requirements are more stable and in this phase the investments required for the application of (more) exhaustive formal verification techniques can be justified. To smoothen the design process it is necessary to allow a formal design specification and formal requirements to evolve together from initial specifications to the refined specifications they become later and to allow formal verification of the design to evolve simultaneously from lightweight, non-exhaustive techniques to thorough exhaustive analyses of the more mature design. The research in this thesis aims to provide formal techniques that apply in particular to the early stages of the design trajectory of complex concurrent real-time systems.

This thesis contributes a number of formal techniques that can be effectively used to study the behaviour and in particular the correct operation of distributed real-time systems. An abstract semantic model is fixed on which executable specification languages can be (and are) based, languages that can be characterised as concurrent, real-time, expressing structure and architecture as well as behaviour. The semantic model is adapted to expose a system's internal behaviour. For simulations one often models closed systems, meaning that both the actual system under design and the behaviour of its environment are modelled together. Traditional semantics in terms of externally observable behaviour alone is inadequate for such models. This model allows one to formally reason about the behaviour of individual components within a larger context.

A technique to execute (e.g. simulate) such executable specification languages is introduced, based directly on the formal semantics. This technique has been applied to the formal specification language POOSL, resulting in a tool called SHESim. It

supports interactive modelling and simulation of POOSL models and the graphical specification and visualisation of the system's architecture.
Furthermore, techniques are studied for the automatic verification of requirements formalised in linear temporal logic of these system models. A key element of the standard technique is the construction of finite state automata that can monitor correctness requirements specified by formulas in linear temporal logic. These automata are called tableau automata. A framework is introduced for describing different types of tableaux in a uniform manner. The use of these automata under verification or simulation conditions is studied resulting in techniques to recognise if finite executions satisfy or violate the properties. Interesting classes of properties are investigated that result in particularly efficient automata. Subsequently, the method is extended to real-time temporal logic, allowing for the expression of quantitative temporal requirements as well as qualitative ones. In particular, an on-the-fly tableau construction is presented for real-time temporal logic, enabling practical model-checking of real-time systems.

Samenvatting

De almaar toenemende complexiteit van gedistribueerde en tijdkritische systemen vraagt om technieken die het ontwerpen ervan ondersteunen en automatiseren. Parallelle en communicerende systemen zijn in het bijzonder moeilijk foutvrij te ontwerpen, terwijl dit vooral voor ingebedde systemen en systemen die de veiligheid moeten waarborgen, van groot belang is. Formele technieken zijn belangrijke hulpmiddelen voor het maken van gereedschappen die het ontwerp en de verificatie ervan ondersteunen en automatiseren en zo de daarvoor benodigde tijd verminderen. Fouten die vroegtijdig in het ontwerptraject ontdekt worden kunnen verholpen worden zonder de buitensporige kosten die gepaard gaan met fouten die aan het eind van het ontwerptraject gecorrigeerd moeten worden. Hoewel formele methoden ontwikkeld zijn om dit soort problemen aan te pakken, hebben ze vaak kenmerken die niet goed stroken met de initiële fase van het ontwerp. In de praktijk wordt in deze fase vaak gebruik gemaakt van simulaties om het systeem te bestuderen. Simulaties zijn erg flexibel, eenvoudig toe te passen en interactief. Gebrek aan exactheid van de gebruikte modellen maakt het echter vaak onmogelijk om ontwerp- of verificatiestappen te automatiseren. Naarmate een ontwerp evolueert, ontstaat er behoefte aan meer diepgaande analyses. Zowel het ontwerp als de collectie ontwerpeisen zijn stabieler geworden en in deze fase is het investeren van tijd in (meer) uitputtende verificatietechnieken te rechtvaardigen. Om het ontwerpproces geleidelijker te laten verlopen, is het nodig om een formele beschrijving van het ontwerp en van de ontwerpeisen tegelijkertijd te laten evolueren van initiële specificaties tot de gedetailleerde specificaties die ze later worden en om formele verificatietechnieken mee te laten ontwikkelen van lichtgewicht, niet uitputtende technieken tot grondige en eventueel uitputtende analyse van het uiteindelijke ontwerp. Het onderzoek in dit proefschrift is erop gericht om formele technieken te ontwikkelen die met name toepasbaar zijn in de vroege stadia van het ontwerpen van complexe parallelle en reactieve systemen.

Dit proefschrift introduceert enkele formele technieken die effectief gebruikt kunnen worden om het gedrag en in het bijzonder de foutvrije werking van dergelijke systemen te bestuderen. Een abstract semantisch model wordt beschreven, waarop executeerbare specificatietalen gebaseerd kunnen worden (en zijn) die gedrag, inclusief parallellisme en tijdsafhankelijk gedrag, structuur en architectuur kunnen uitdrukken. Dit semantische model is op een zodanige wijze uitgebreid, dat het interne gedrag van het systeem zichtbaar wordt gemaakt. Voor simulaties gebruikt men vaak een gesloten model. Dit betekent dat zowel het systeem dat ontworpen moet worden, als ook de omgeving waarbinnen het systeem moet gaan functioneren,

gemodelleerd worden. Traditionele semantieken die alleen extern observeerbaar gedrag beschrijven zijn daarvoor niet adequaat. Deze beschrijving maakt het mogelijk om te redeneren over het gedrag van een systeem in een specifieke context. Verder wordt er een techniek geïntroduceerd om dergelijke talen te executeren (bijv. om simulaties uit te voeren). Deze techniek is rechtstreeks gebaseerd op de semantiek van de taal en is toegepast op de formele specificatietaal POOSL in de vorm van een tool, die SHESim heet, waarmee POOSL modellen gemaakt en gesimuleerd kunnen worden en model en gedrag en architectuur grafisch gevisualiseerd kunnen worden. Daarnaast worden technieken bestudeerd voor de automatische verificatie van eigenschappen uitgedrukt in lineaire temporele logica. Een cruciaal onderdeel van een standaardtechniek voor verificatie is de constructie van eindige toestandsautomaten die corresponderen met formules van de logica en gebruikt kunnen worden om te controleren of gedragingen van het systeem aan deze eigenschap voldoen. Deze automaten worden wel tableau automaten genoemd. Dit proefschrift introduceert een kader waarbinnen verschillende soorten van tableaux op uniforme wijze behandeld kunnen worden. Er wordt bestudeerd hoe ze gebruikt kunnen worden tijdens simulaties. Dit resulteert in technieken om van eindige executies te bepalen of ze al dan niet aan de eigenschap voldoen. Ook worden bijzondere klassen van formules bekeken die leiden tot efficiënte automaten. Vervolgens worden deze technieken uitgebreid naar een temporele logica waarmee ook kwantitatieve tijdsaspecten van het gedrag uitgedrukt kunnen worden. In het bijzonder wordt er een tableau constructie gepresenteerd, die efficiënt genoeg is om praktische verificatie van deze logica mogelijk te maken.

Acknowledgements

The work described in this thesis was not and could not have been performed in isolation. It involved the help and support of many, to whom I am largely indebted. First of all I want to thank prof. Stevens for creating the opportunity for me to perform this work in his group together with an enthusiastic team of researchers, without too much interference from other duties and allowing me to continue to work in this environment after the end of my Ph.D contract.
Thanks also go out to prof. Baeten, prof. Feijs and prof. Larsen for reviewing a preliminary version of this thesis and all other members of the promotion committee for their valuable time and effort.
I am especially grateful to Jeroen Voeten for his guidance and inspirational support throughout the entire project, for interesting discussions and support in many other ways. Dennis Dams has taught me most of what I know about formal verification and supported me during a long period of research that at times seemed to fail to produce results, but eventually became chapter 8 of this thesis. Piet van der Putten has always been a driving force and inspiration within the Software / Hardware Engineering team and good at fuelling interesting discussions from a different perspective.
During these years, many roommates have been good company. Thanks for that, Ton, Maarten, Inder, Emil, Robin, Terry, Lucia, Daniel, Ron, Natalia, Zhangqin, Leo (thank you for proof reading and for providing the LaTeX macros for the arrows in this thesis), Frank for your work on and with the SHESim tool and finding many of its bugs (so they could be resolved) and Bart for extensive testing of the tool as well and also for keeping me company on many a train to or from Sittard.
I also owe gratitude to a number of students that have contributed directly or indirectly to this work: Maurice de Leijer, Nils van Tetrode, Georgina Lopez and Pieter Cuijpers.
Last but not least, I should thank my family for having faith during these years, that I was doing something useful, even though they did not have a clue what purpose this work could possibly have. I'm also grateful to the other members of the ICS group and many more people that are not mentioned explicitly.

Contents

Chapter 1

Introduction

As technological possibilities increase in a continually rapid pace, products that are being designed become more and more complex. Information technology permeates a wide variety of systems. The amount of software in consumer electronics increases with every new generation. Information technology is applied in transportation, commerce, industrial control systems, often in the form of embedded systems. In contrast with systems like a desktop PC, faulty behaviour in such systems can be very costly (for example if all products in the field have to be replaced) or can be a threat to the safety of people (for instance systems controlling a railway system or air traffic).

Such designs of increasing complexity have to be done by the same or a smaller number of designers in a shorter time to market. To be able to achieve this, one has to work at higher levels of abstraction and automate larger parts of the development process. Systems of concurrent and communicating components tend to exhibit unpredictable interactions that may lead to incorrect behaviours. If such errors stay undetected until late stages of the design trajectory or even until delivery of the actual products, the costs of solving such errors can be very large. The assessment of the correctness of a design is an inherently difficult problem, because of the complexity of the system and the vast number of possible scenarios that have to be explored.

This thesis deals with formal techniques that can be used to automate the analysis of complex distributed systems and in particular the analysis of the correctness of the system's behaviour. It is well known that distributed concurrent and reactive systems are error prone. A design has to be verified and tested to get rid of (as many as possible of) these errors before it is actually realised. The errors are often caused by the fact that the inherent non-determinism (originating from concurrency, the distributed nature or user interaction) makes the behaviour complex and hard to predict. The effect is that the number of ways the system can behave becomes very large and thus hard to validate that it will in any situation be as intended.

In the following section an introduction is given to the problem domain of this thesis; hierarchical, distributed, concurrent real-time hardware/software systems. The design issues and problems that play a role in these systems will be discussed as well as the particular design issues on which this thesis focusses. Section 1.1.2 discusses the existing approaches towards these design issues, their respective strong and weak points and the existing gap between formal/exhaustive and informal/ad-

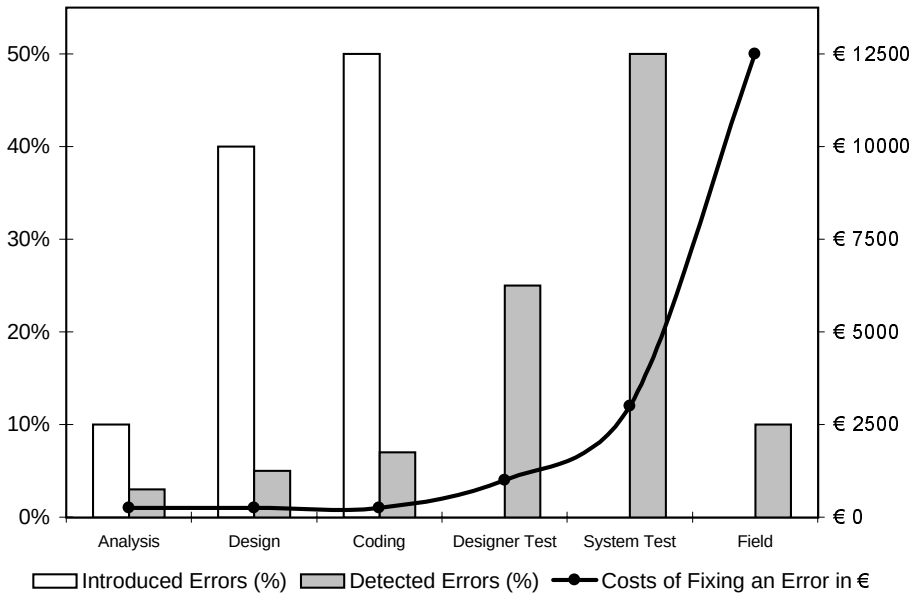


Figure 1.1: Introduction, detection and costs of errors in the design trajectory [130]

hoc methods. Section 1.2 states the objectives of this thesis and the issues that are addressed. Finally, section 1.3 explains the structure of the thesis.

1.1 Designing Distributed Systems

In the course of designing a system, errors are made and corrected as they are detected. Generally, the later an error is detected, the more costly it will be to solve it [130]. Figure 1.1 shows the results of a study of errors in software systems. It shows the percentages of errors introduced and detected in different phases of the design as well as the corresponding costs of fixing an error in that particular phase. The graph clearly shows that it is imperative to try to detect mistakes as early as possible. A well known illustration is the bug in the floating point division units of the Intel® Pentium® chip, which cost the company $500 million, as there were already a large quantity of products in the field that had to be replaced. Apart from the economical necessity to find errors, there are systems that are mission-critical or safety-critical in which the consequences of a system failure could be disastrous; for instance, a problem with the flight control system in the Ariane 5 rocket in June 1996 abruptly ended a $5 billion mission.

This thesis concentrates on the kind of systems that can be characterised as follows: real-time systems, distributed systems, embedded systems, multimedia systems or telecommunication networks; systems, performing many concurrent tasks in a real-time environment. Such systems often consist of multiple distributed concurrent components that frequently communicate with each other. They are often reactive, meaning that during operation they perpetually communicate with the en-

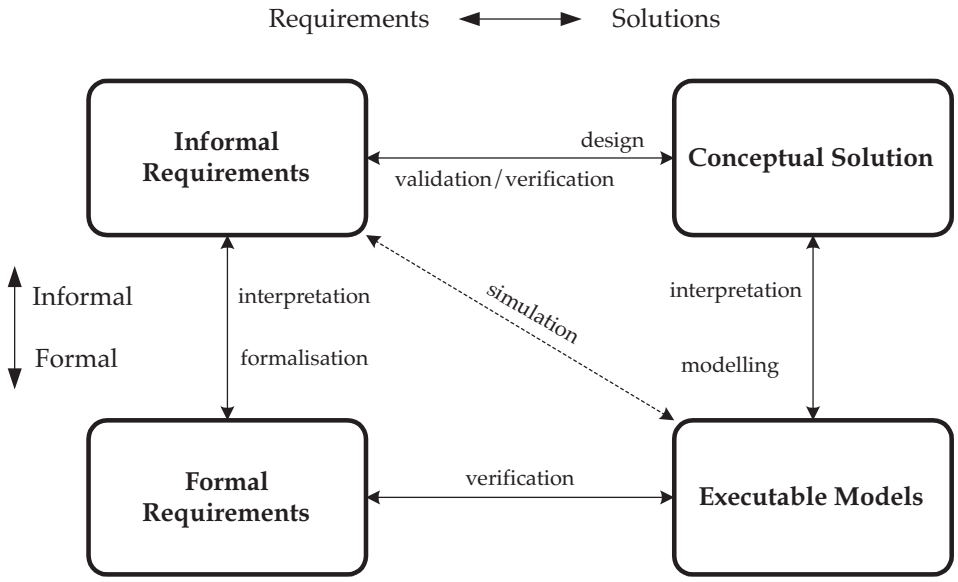


Figure 1.2: Requirements and designs

vironment, responding to stimuli from the environment or producing them. Many such systems combine both hardware and software components. As they interact with the physical world, timeliness and other timing aspects are very important. For dealing with complex heterogeneous systems, architecture and structure of the composite system are important.
Stimulated by technological advances, the complexity of these systems is growing rapidly. Yet, these complex systems must be designed in the same or even smaller amount of time with the same groups of developers. To achieve this, higher levels of abstraction are required and larger parts of the development process have to be automated.

1.1.1 Design Problems and Issues

To effectively design the aforementioned systems, many issues are important. Such issues include functional correctness, performance, testability, maintainability, domain analysis, requirements capturing, design space exploration, design for test and debug, project management, and so forth. There are many aspects that need consideration to constitute a good design methodology. This thesis concentrates in particular on the validation and verification of the qualitative or functional aspects, correctness of the system and in particular on the requirements regarding communication and concurrency and real-time aspects thereof.
The investigation of the functional correctness of a design is often designated as 'verification' or 'validation'. The terms are occasionally used to differentiate between different forms of this activity. However, they are not used consistently. Verification is frequently defined as an activity to answer the question "Are we making the

system right?", whereas validation answers the question "Are we making the right system?". Another use is to reserve the term verification for formal analysis and use validation in contrast for informal analysis. At other times they are used as synonyms. We shall use the terms in the following manner. By *verification* we mean the investigation of the truth of a statement about (a model of) a system under design. *Validation* is used to denote the more general investigation of the suitability of the conceptual solution for its purpose and goals. Furthermore we call a method *formal* if it is based on rigorous well-defined (mathematically defined) concepts. Formal verification thus means the investigation of a formally defined relationship between formalised representations of the system and its requirements. This also means that a formal verification does not directly apply to the system that is being designed, but only applies if the requirements have been captured correctly and the formalised model is an accurate representation of the actual system. Consequently, a statement that a system has been formally verified requires an explanation of what was formalised and what properties where verified [117].

A (somewhat simplified) overview of the validation / verification is shown in figure 1.2. A design typically starts with an informal description in some natural language of what the system is for and what criteria it is supposed to meet. Often, at the beginning there are already some ideas about the implementation of the system. The system might be an evolution, improvement or extension of an existing design. It may be known to use certain components. The collection of requirements will be incomplete. As the design progresses, the collection of requirements will be extended and the conceptual solution will obtain shape. Design decisions that are taken in this process have to be validated and the design has to be verified to satisfy all of its requirements.

To automate certain analyses of the design, a formal model of the design is made, for instance in the form of an executable specification. Such a formal model captures some part of the design that is to be analysed. The model may be very abstract, capturing precisely those aspects that are to be analysed, or may be very detailed, trying to capture as many of the design's aspects as possible. To automatically verify certain aspects of the design, next to the formal model of the design, a formalised statement of the requirements is necessary. Such requirement statements can be the absence of global deadlock or more complicated behavioural constraints. Examples of such formalised requirements are an executable model at a higher level of abstraction and a specification in temporal logic. Once it has been analysed whether the formal model satisfies the formalised requirements or not, the result can be interpreted for the actual design. If the model is accurate and the requirements are captured adequately, then the result should have a meaningful interpretation for the verification of the actual design. Sometimes the verification is partially automated. For example when an executable model is simulated and the produced simulation results are inspected manually to determine the interesting properties.

1.1.2 Current Practice, Techniques and Technologies

Many techniques exist for the verification of distributed and real-time systems, ranging from manual inspection of a design to formal verification methods. Different methods have their strong and weak points. Together however, they might comple-

ment each other and strong points of one method might compensate weaknesses of another.

Modelling behaviour of real-time concurrent systems

The description of a design may occur in the form of a (partially) textual or graphical description, a description using pseudo code, some sort of formal specification or anything in between. Textual specifications are readable (if well structured) and flexible. On the other hand they are often ambiguous and incomplete or inconsistent. Programming languages and in particular C/C++ [167] are often employed to describe a system's behaviour. Such specifications are executable and C is well known and understood among designers. It allows simulation of the design and the popularity of the language makes that integration with existing (commercial) tools is often feasible. Regretfully, C does not have support for aspects such as concurrency, communication, timing and hierarchy; these are often introduced via (proprietary) libraries. Another disadvantage is the lack of semantics. There is no formalised semantics and the existing standards describing the language leave the meaning of certain language constructs undefined.
Formalisms to construct mathematical models of systems include such things as Petri nets, process algebra, labelled transitions systems, finite state automata and Markov chains. All have their particular views on a system and focus on particular aspects. For instance, there are different abstractions to formalise concurrency, such as synchronous concurrency, asynchronous concurrency (interleaving) and partial orders (event structures). There are different abstractions of physical time and probabilistic aspects. Formalisms may capture structural and architectural aspects of a design or focus strictly on behavioural aspects. Most formal methods strive for simplicity, to allow for efficient analysis. They do not try to capture all aspects in a unified formalism, for such a formalism would not be easy to analyse.

State-space explosion problem

The major problem in the verification of distributed systems is the enormous number of their potential behaviours. Suppose we have a system consisting of M components, each having N different internal states. The state of the entire system can be represented by giving the local state of every component individually. The number of possible system states is N^M. Although the components are not completely independent and not all of these global states may be reachable, the actual number of reachable states does tend to show the same trend; the number of states grows exponentially with the number of system components and the size of its specification. This effect is called the *state-space explosion*. This means that complex systems may exhibit very large numbers of possible behaviours. Since verification entails checking that certain requirements hold under all possible circumstances, all behaviours should be explored. Although in specific cases the number of states to be explored can be reduced significantly or certain classes of behaviours can be examined all at once, the global trend is unavoidable.
The state-space explosion is the central problem in the verification of distributed systems. A designer is unable to foresee all potential interactions. Simulations do not cover the entire state-space and formal methods that do (try to) cover the entire space

battle with resource constraints such as computer memory or time required for the verification. Even though the formal methods are helped by Moore's Law, bringing exponentially increasing performance of computers used for verification, this is cancelled out by increase of the complexity of the systems that are being designed.

Analysis Techniques

There exist a number of techniques to analyse a given (model of a) design for certain correctness properties; such techniques include peer reviewing, simulation, testing of realisations or prototypes and formal verification techniques. In peer reviewing an (experienced) colleague who is not involved in the same design inspects the designed code and checks it manually for mistakes. Although this is an effective method to remove a number of mistakes, it needs to be followed by automated analysis to further examine the design. An (automated) analysis technique that is being used extensively and effectively in industry, is simulation. Both simulation and testing are usually non-exhaustive in the sense that not all possible behaviour is exercised and checked for conformance with the requirements. Exhaustive formal verification techniques have been and are being applied successfully in industrial projects in a number of cases (for example [17, 42, 50, 52]) and are being applied as an integral part of the design trajectory more often. They do intend to cover all behaviour of the system, but their application is not always easy. We will discuss some of the important tools for automated analysis.

Simulation Simulations are easy to use and can be largely automated. Validation of the simulation results however is often ad hoc and manual. It generally does not provide definite (positive) answers to the posed verification questions. It can expose erroneous behaviour, but the absence of bad behaviour cannot be guaranteed. Another problem with the simulation method is that in practice it is often not based on well-defined semantics. Interpretations of concurrency, communication primitives and real-time are often not made explicit, but are affected by the simulation model or the construction of the simulator; for instance the mechanism used by the scheduler of a discrete-event simulator to fire scheduled events.

A big advantage of the simulation method is that it is very scalable. It doesn't suffer from the state-space explosion problem in the sense that large models can still be simulated. Obviously if does suffer in terms of coverage and error-detecting capabilities. This makes it possible to use a single model (or a small set of models) for all simulation-type analyses. As the design or requirements change, it is easy to reevaluate the new model with respect to the new requirements. As errors often occur in unexpected ways, it is important to use detailed models and continue to monitor all of the requirements. Errors are often found while focussing on completely different properties [116]. The error finding capabilities are improved by automating the evaluation of simulation results with respect to (formalised) requirements. Models for simulation can be made more detailed and adequate and moreover they can be used, with appropriate visualisation, to communicate among designers or with customers.

Testing and prototyping Prototypes are often used for verification. A prototype is close to the actual realisation and is thus a highly adequate model. A prototype can be executed in real-life speed, which is often impossible for simulation models. However, the design must be almost ready before a prototype can be built and the process is time consuming and expensive. Models can be built and analysed quicker.

Formal methods Formal methods originate from a more ambitious and more fundamental than practical approach to system specification and design. It provides the vehicle to reason mathematically about the correctness of a design. This allows a mathematical proof of its correctness. As a side effect, the use of formal methods forces designers to make their ideas about a design or its requirements very precise and this uncovers many errors already.
Although formal methods to reason about and prove the correctness of designs have been around for a number of years, integration into industrial practice remains limited. This is caused by a number of issues that hinder their practical application. Some of these issues are the following.

- The state-space-explosion, the intrinsic hardness of the verification problem and the desire of most formal verification methods to produce a definite answer, make it a big effort to keep the state space within reasonable size.
- Formal methods have a reputation to be hard to use and to require the user to be knowledgeable both in advanced mathematics and also in the internal operation of particular tools.
- There is a lack of industrial strength tools. In particular there is a lack of tools that integrate in the entire design flow and tools that are used.

The two major types of verification techniques based on formal methods are the following.

- *Model-checking* is a procedure to check the correctness of a system by traversing the entire state space whilst looking for erroneous behaviour (expressed as the reachability of particular states or by formulas in temporal logic). A big advantage of model checking is that it is completely automatic. Model-checking has been applied successfully in the field of communication protocols and hardware verification; it is particularly suited for control intensive and less for data-intensive systems. If the check of a particular requirement fails, the model-checker tool provides a counter example which demonstrates this. The state-space explosion and the struggle against it often do require that a new (abstract) model of the system is made targeted towards the requirement that is being checked; such models are thus not optimised for readability but only to reduce state space. This might call for an in-depth study of the model and knowledge of the internal operation of the tools to achieve a successful verification [117]. When the design changes, the abstract model has to be adapted accordingly and the verification has to be repeated. Examples of (academic) model-checking tools are Spin[107], SMV[137], Mocha[14], UPPAAL[25] and Kronos[63]. Commercial tools are beginning to arise and tool vendors are starting to integrate model-checking techniques in their tool sets.

- In *deductive methods* such as theorem proving, correctness requirements are formalised as theorems in a mathematical theory which describes the system's behaviour. A requirement can be shown to hold by proving its corresponding theorem in this theory. Deductive methods are able to deal with infinite state and parameterised systems using inductive proofs. This makes them better suited for data-intensive systems. Theorem proving tools assist in the construction of such a proof or try to construct the proof autonomously. Generally however, such tools are not (yet) powerful enough to produce (large parts of) such proofs entirely autonomously. Usually the crucial steps of the proof have to be invented by the user and the theorem prover might be able to fill in the smaller steps. Another class of tools are proof checkers, tools that do not generate proofs, but rather check that a manually generated proof is correct. This requires formally trained people and a lot of manual effort to construct the proof in minuscule steps that can be checked by the tool. Thus, deductive methods are not automatic, require user interaction and mathematical expertise of the user as well as a deep understanding of the design itself [117]. Theorem provers are less suitable as debugging tools than model checkers; they do not provide much insight if a particular theorem cannot be proved. Also here the chances of success increase if one is experienced with the tools and familiar with their internal operation. This is illustrated by the fact that successful case studies in formal verification often involve the original tool builders or inventors of the underlying theory. Some popular theorem provers are HOL[94], Nqthm[39], PVS[152] and Isabelle[150].

The fight against the state-space explosion and the preciseness required for the formal methods make their application typically relatively time-consuming. This makes it hard to deal with aspects of a real-world design process, where specifications change and requirements are trade-offs. As timing or probability issues come into play the formal techniques become harder to use. The computational effort increases as well as the technical and mathematical complexity.

Current Status

Automated analysis of models of the design of a system is required to obtain insight into the correct operation of the design as early as possible in the design trajectory. In particular distributed reactive systems are difficult to design since unexpected interactions often cause subtle problems. Existing techniques are insufficient to deal with the verification problem in an effective and satisfactory way.
Existing simulation practices are flexible, easy to use and capable of dealing with large systems at a small level of detail. However, they are often too ad hoc and informal to support extensive automation. Formal methods are precise and give definite answers to the verification question. They are good at exposing subtle errors that tend to escape verification by means of simulation and testing. Yet, they are limited by the state-space explosion problem and expensive to use; in particular early in a design this is hard when design and requirements change frequently and reevaluation of all the requirements is required after such changes.
It follows that no one of them can replace the other and they should be glued together in a consistent framework. From the complementary nature of non-exhaustive and

exhaustive (formal) methods it follows that they can be used effectively together. Non-exhaustive methods are cheap, easy to use and flexible. This makes them very suitable in the initial stages of the design, to quickly evaluate design alternatives and obtain a quick impression about the correct behaviour of the system. As the design becomes more stable at a particular level of abstraction or a particular subsystem, and if some solid insight into the correct behaviour is desired to continue with a particular alternative, it is possible to exploit exhaustive formal methods to detect more subtle errors that are only exposed by very particular interactions and are likely to go undetected by simulations.
One area that needs to be addressed to alleviate the problems is a better integration of non-exhaustive and exhaustive formal methods and a formal basis for non-exhaustive analysis methods and simulation in particular. Especially the automated evaluation of simulation results, with respect to formalised requirements is required. It is important to establish the connection between the two by using (abstractions of) the same requirements in both worlds, and making the transition from one type of analysis to the other smaller.

Real-time Verification

Many embedded systems operate in an environment where providing a timely response is equally important as providing the correct type of response. For instance consumer electronics applications or an airplane's flight-control system. Formal methods that deal with timing aspects of a system have to take this into account. Many formal methods and tools abstract from the quantitative aspects of time to yield a simpler theory and more efficient tools. Model-checkers for timed systems are UPPAAL [25] and Kronos [63] (among others) and also Spin has its timed extensions [35]. Real-time model checking (in particular in a dense time domain) is computationally more expensive and therefore suffers even more from the state-space explosion problem, although the gap between untimed and timed verification algorithms is getting smaller with the advent of more clever techniques and improved data structures. Theorem provers can also be used, but the theories and proofs become more complicated.
In simulation, semantics for time is a problem. Many simulation tools do not have solid semantics for real-time built in and often a model of time is built by the designers on top of the simulator language. This is necessary for example if C is used. In particular in combination with concurrency, a practical real-time semantics is not trivial to make. From a computational point of view (simulation speed) the addition of time and other concepts such as probabilities are not that expensive.
To apply the temporal logic model-checking approach which is very popular and successful in the untimed case, practical methods for temporal logic model-checking in dense real-time still have to be developed.

1.1.3 Example

The kind of errors that may occur are illustrated by the events which occurred during NASA Jet Propulsion Laboratory's Mars Pathfinder mission in July 1997. Although the mission was successful and resulted in unique images of he surface of the planet Mars, there were some flaws in the operation. The mission delivered a small vehicle

called the *Sojourner Rover* to the surface of the planet. Controlled from earth the vehicle would drive around, collect data and images and transmit them to earth. The activities of the rover were controlled by a number of software tasks running on a real-time operating system. The software processes ran at different priority levels. A time-critical task was used to schedule the use of the system bus and a watchdog monitored the timely execution of this task. The software tasks shared certain resources and used a mutual exclusion mechanism to control access to this resource. The scenario which occurred was actually a known problem in this sort of systems and is called *priority inversion*. The shared resource was in possession of a low-priority task. Whilst occupying the shared resource, the task is interrupted by a medium-priority task used for the processing of the data that is being collected. As scheduled, the highest-priority time-critical task is started and requires the use of the shared resource. It cannot access the resource however, since it is in use by the low-priority task. The low-priority task, in turn, cannot be activated to complete its business with the shared resource because it is suspended by the medium-priority task. As a result this medium-priority task preempts the high-priority task (inverting their priorities) which is now unable to complete its duties in time. The watchdog notices this and initiates a total reset of the vehicle's software. The scientific activities of the rover are stopped until it is fed with new instructions the following day.

Errors often turn up in unexpected ways. Glenn Reeves of the Jet Propulsion Lab commented on the problem: "*Our before-launch testing was limited to the 'best case' high data rates and science activities. The fact that data rates from the surface were higher than anticipated . . . served to aggravate the problem. We did not expect nor test the 'better than we could have ever imagined' case*". It shows one of the difficulties of verification. The problems often occur in unanticipated ways. The model of the environment used during testing contained some implicit assumptions that turned out to be incorrect.

It is hard to say afterwards whether such a problem would have been discovered using exhaustive formal methods; the interactions regarding the mutual exclusion on the shared resource and the prioritised scheduling of the concurrent tasks on a single microprocessor must still be present in the model. It is easy to construct an abstract model that exposes this error very quickly, both by simulation and by model-checking. However, to come up with such a model, one first has to expect problems in this part of the system. It is important to (have the tools) be always alert to pick up the signals pointing to errors. The priority-inversion problem and the resulting system reset were actually observed during testing, but the problem was ignored and explained as a hardware glitch.

In this particular case the consequences of the error were not that drastic. The design of the system allowed uploading improved control software to the vehicle once the error was discovered and corrected. It was able to carry on its tasks without the system resets. The mission was overall successful and is mentioned by Wind River Systems, Inc®. as one of the success-stories of their VxWorks® real-time operating system (including the priority-inversion problem and their solutions that deal with such issues).

1.2 Objectives of this Thesis

Informal and ad hoc verification activities are insufficient to find hard bugs in distributed systems. Exhaustive formal methods are important tools for verification, but are (as yet) maladapted for certain parts of the design trajectory and expensive to use. It is worthwhile to introduce formal methods in simulation techniques and in evaluation of simulation results. In particular there is a lack of practical formal methods for the temporal logic approach to model-checking of real-time systems.
This thesis introduces (formal) techniques for validation and verification of system-level models of complex reactive real-time hardware / software systems. These techniques attempt to decrease the gap between formal system-level modelling and analysis, simulations of such systems and their formal verification. There are many kinds of systems, and equally many kinds of formalisms; some better suited for one type of systems and others better for another. In this thesis we will focus in particular on systems that match the following criteria: statically structured, hierarchical, complex, possibly data intensive, component based or object oriented. Examples of these are such systems as telecommunication protocols, networks and switches, industrial control systems and work-flow models as well as systems-on-a-chip.
Although an effective design framework requires attention for a lot of different aspects such as requirements capturing, analysis, tool support, project management and so forth, we limit the scope of this thesis to only a few of these aspects and in particular to techniques that form ingredients to the verification part of a coherent methodological framework for the design of real-time distributed systems. Since real-time is an important aspect of many systems, we focus on real-time systems and formal methods for real-time systems.
Although the techniques described in this thesis are generally applicable to the type of systems and methods described, their development has been centred around a language called POOSL (Parallel Object-Oriented Specification Language, [156]), designed to express formal executable specifications of real-time distributed communicating systems and the encompassing methodological framework SHE (Software / Hardware Engineering, [156]).
The main objectives of this thesis are the following.

- To obtain an execution model for expressive executable specification languages for complex real-time systems based on its formal semantics and an implementation in the form of a simulator for a particular instance of such formalisms: POOSL.

- A formal model for structured architectures of communicating components. To manage the complexity of large systems, many such systems are built from components that interact through precisely defined abstract interfaces to hide internal aspects from other components. Architecture design becomes crucial when dealing with large and complex systems.

- A framework for requirements specification for such systems. A way to write down requirements in a formalism such as temporal logic, respecting concepts like architecture and encapsulation of components. Such a framework enables the use of (real-time) temporal logic requirements specification and automatic verification.

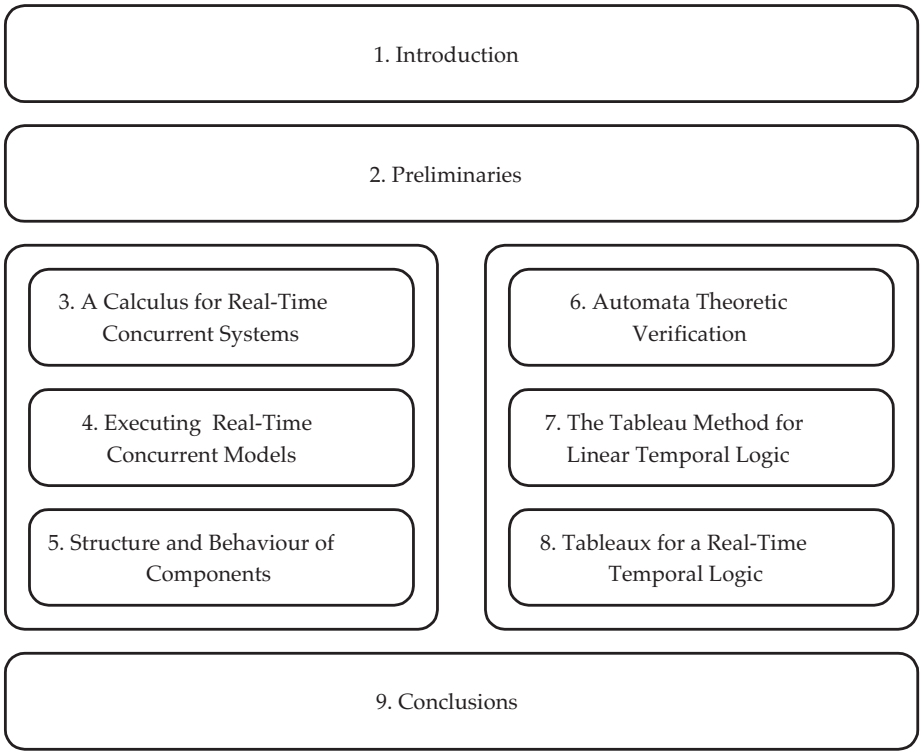


Figure 1.3: Overview of the thesis

- A methodology for the automatic verification of the designated systems. Dealing with the verification of formalised requirements during simulations as well as in model checking.
- A practical extension of these methods for automatic verification of (timed) systems using (real-time) temporal logic also both for simulations and model-checking.

1.3 Overview of the Thesis

This section gives an overview of the thesis. It explains the structure and gives a short description of the individual chapters.

1.3.1 Structure of the thesis

The overall structure of this thesis is depicted in figure 1.3. The thesis consists of two main parts. The parts are preceded by the introduction in chapter 1 (this chapter) and chapter 2 introducing the required mathematical preliminaries on which the results in this thesis are built. The first part consists of the chapters 3, 4 and 5 and

introduces a formal model for statically structured, concurrent, communicating real-time systems (chapter 3) and an execution mechanism for system descriptions based on that model (chapter 4). Chapter 5 deals with semantics and formal (temporal logic) requirements for statically structured and component-based systems.
The topic of the second part is the automatic verification of such models. Application of formal verification techniques in simulations will be discussed in chapter 6. Tableau constructions, an important ingredient of temporal logic verification, are the topic of chapter 7 and the tableau method is extended to real-time temporal logic properties in chapter 8. Preliminaries on temporal logic and the automata theoretic approach to model-checking are in chapter 2. At several places in this thesis, the developments are illustrated by an example protocol. The example protocol is introduced in chapter 2. Both parts mentioned above can be read independently.

1. Introduction This chapter introduces the problems addressed in this thesis, states its objectives and gives an overview of its structure.

2. Preliminaries The developments in this thesis build on existing mathematical constructs, such as labelled transitions systems, finite state automata, process calculi and temporal logic. Chapter 2 is a quick introduction to the ones used in this thesis. It also introduces the Automatic Protection Switching protocol that is used as an example throughout the thesis.

3. A Calculus for Real-Time Concurrent Systems The techniques introduced in this thesis are aimed at methods and languages that are expressive and mature and able to deal with complex designs, yet are based on a precise semantics describing its behaviour. It would be beyond the scope of this thesis to introduce a language of such complexity and its formal semantics. Instead, we shall introduce in chapter 3 the syntax and semantics of a calculus that will serve as an abstract model of such a language. We have to keep in mind in the remainder of the thesis that the methods are aimed at complex languages based on such a model. The calculus explicitly describes structure and architecture of a system, as well as (timed) behaviour.

4. Executing Real-time Concurrent Models Many kinds of automated analysis of models will require the execution of the model or the generation of its state-space. It should be possible to build the transition system that is defined by the semantics of the model. Chapter 4 describes a method for the execution of models described in the calculus. The methods employ *execution trees* to represent processes and to compute the possible transitions. In complex systems, these trees can be 'heavy' and the method allows for incremental updates of these data structures. The correctness of this execution method is demonstrated and an implementation in the form of a graphical tool set for modelling and simulation for the language POOSL [156] illustrates its use.

5. Structure and Behaviour of Components This chapter deals in a formal way with structure, hierarchy and encapsulation. For automatic verification of system

models, we need both a description of the system and a description of the requirements in a precise form. To allow the formalisation of properties of a system and the interpretation of these properties, the semantical model is extended to expose internal states and events. It is shown that the extended semantics does not change the behaviour. The extension allows the expression of static and dynamic properties of a system, both of external and of internal behaviour. A framework for an extended form of linear temporal logic is introduced to express properties of systems defined by the calculus, respecting structure and hierarchy of a system and the component / object-base philosophy of encapsulation and well-defined interfaces.

6. Automata Theoretic Verification A popular approach towards the automatic verification of linear temporal logic is discussed, the so-called automata theoretic approach. The application of these methods to non-exhaustive verification techniques and random state space exploration / simulation in particular are discussed. It is shown how the technique can be adapted for monitoring requirements during simulations and the class of properties for which this is possible is determined.

7. The Tableau Method for Linear Temporal Logic An important part of the automata theoretic verification is the conversion of formulas in linear temporal logic to finite state automata that recognise the behaviours that do not satisfy the formulas. This conversion is called the tableau method. An overview of the tableau method for the generation of finite state automata from temporal logic formulas is given. Such automata can be used to observe the conformance of a system to safety properties. The theoretical complexity of the conversion is high and practical approaches try to minimise the size of the automata resulting from the conversion. Different known tableau methods and optimisations are discussed. Classes of formulas are recognised for which efficient automata can be constructed.

8. The Tableau Method for a Real-time Temporal Logic Traditional linear temporal logic describes qualitative temporal properties of a system's behaviour. It relates only the order of events. Extensions of the logic exist that can also refer to qualitative temporal behaviour, allowing the distance in time between events to be measured. The practical tableau methods are extended to a class of real-time temporal logic formulas, allowing the verification of real-time temporal logic properties using timed automata. Special care has to be taken if these automata are to be used as observers during simulations, since the automata must be deterministic. Practical tableau constructions are given and the use of real-time tableaux as observers for simulations is discussed.

9. Conclusions The conclusions and possible future work are summarised in chapter 9.

Chapter 2

Preliminaries

Mathematical structures are employed to formalise and reason about the behaviour of concurrent systems. We use a number of such structures and possibly slight deviations of them that better fit our needs. This chapter serves as a quick introduction in the mathematical objects used in this thesis and the particular notations that are used. Section 2.1 starts with some general mathematical concepts and subsequently a number of structures are introduced that are frequently used to formalise concurrent and communicating systems. Labelled transition systems are introduced in section 2.2 as a formal abstraction of reactive systems. The notion of behaviour of a labelled transition system and corresponding concepts of equivalence between systems are discussed in section 2.3. Labelled transition systems are often specified by process calculi. An introduction is given in section 2.4. To analyse properties of reactive systems, these properties must be formalised. A particular framework to express such properties is temporal logic; it is the topic of section 2.5. Algorithms for the analysis of systems using temporal logic often build on finite state automata. These are introduced in section 2.6.

2.1 General mathematical concepts

We will first go through some basic mathematical concepts and introduce the notations we use for them in this thesis. More elaborate treatments can be found in many textbooks. This introduction was inspired by similar ones in [183] and [156].

2.1.1 Sets, Relations and Functions

Sets and their typical elements

A *set* X is a (finite or infinite) unordered collection of mathematical entities called *elements* of the set. $x \in X$ ($x \notin X$) is used to denote that the object x is (not) an element of X. The set without any elements is denoted as $\varnothing$. The number of elements of a set X is referred to as $|X|$ (we write $|X| = \infty$ if X has infinitely many elements).
A set can be denoted by enumerating its elements, such as $\{0, 1\}$ or $\{a, b, c, \ldots\}$ (an ellipsis is often used to indicate that the enumeration should be completed in a way

that is evident from the listed elements). Another way to specify a set is by a property P. The set $\{x \in X \mid P(x)\}$ (or just $\{x \mid P(x)\}$ [1]) denotes the set of all elements of X for which the property P holds. The set of odd natural numbers, for instance, is denoted by $\{n \in \mathbb{N} \mid n \textbf{ mod } 2 = 1\}$. Yet another way is to define a set by some explicit construction from the elements of another set; $\{e(x) \mid x \in X\}$ is the set of all elements obtained by substituting the elements of X in the expression $e(x)$, for example the set $\{2n \mid n \in \mathbb{N}\}$. A set X is said to be a *subset* of Y (denoted as $X \subseteq Y$) if every element of X is also an element of Y ($X \not\subseteq Y$ denotes that X is not a subset of Y). The set containing precisely the subsets of a set X is called the *powerset* of X: $2^X = \{Y \mid Y \subseteq X\}$. The *union* of the sets X and Y is denoted as $X \cup Y = \{x \mid x \in X \text{ or } x \in Y\}$ and their intersection as $X \cap Y = \{x \mid x \in X \text{ and } x \in Y\}$. The *difference* between sets X and Y is the set of all elements of X that are not elements of Y: $X \backslash Y = \{x \in X \mid x \notin Y\}$.

Some familiar sets are the set $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$ of natural numbers, the set $\mathbb{R}$ of real numbers ($\mathbb{R}^{\geq 0}$, the set of positive real numbers including 0), and the set $\mathbb{Q}$ of all rational numbers ($\mathbb{Q}^{\geq 0}$, the set of positive rational numbers and 0).

In this thesis, sets of all sorts are used frequently. One often needs to refer to elements of such sets. To avoid the necessity to specify the particular set whenever such an element is required, we make use of *typical elements* of sets. If we introduce a collection X and variable x as a typical element of X, then whenever x is used, it is implicitly understood that $x \in X$. Another way to say that x is a typical element of the set X is to say that x ranges over X. Moreover we implicitly assume that whenever it is stated that x ranges over X, then also the variables x_0, x_1, x' and so forth range over X. If X is a set of sets, then we denote their union by $\bigcup X = \{a \mid a \in x \text{ for some } x \in X\}$ and if X is non-empty, their intersection by $\bigcap X = \{a \mid a \in x \text{ for all } x \in X\}$.

For sets X and Y the product $X \times Y$ is the set of all ordered pairs of elements from X and Y respectively, $\{(x, y) \mid x \in X \text{ and } y \in Y\}$. In general one can define the set of ordered n-tuples $(x_1, x_2, \ldots, x_n)$ from a product of n sets $X_1 \times X_2 \times \ldots \times X_n$. If all X_i are identical and equal to say X, one can also write X^n. The element number i of the ordered n-tuple $\bar{x}$ is denoted by $\bar{x}(i)$.

Relations and Functions

A *binary relation* between two sets X and Y is a subset of $X \times Y$. If R is such a relation, we often write xRy instead of $(x, y) \in R$. If $R \subseteq X \times X$ we say that R is a binary relation *on* X. If R is a binary relation between X and Y and S is a binary relation between Y and Z then RS is a binary relation between X and Z, such that $xRSz$ if and only if there is some $y \in Y$ such that xRy and ySz. In the remainder we abbreviate the cumbersome phrase 'if and only if' as 'iff'.

If X is a set then the *identity relation* Id_X on X is the binary relation $\{(x, x) \mid x \in X\}$. If R is a binary relation on a set X then $R^{(n)}$ is the relation obtained from composing R with itself n times, taking $R^{(0)} = Id_X$ and $R^{(n+1)} = RR^{(n)}$ for all $n \geq 0$. The *reflexive transitive closure* R^* of a relation R on X is $\bigcup\{R^{(n)} \mid n \in \mathbb{N}\}$.

A *partial function* f from X to Y is a binary relation between X and Y such that for

[1] This is a simple but naive introduction to sets. The famous paradox by Bertrand Russel shows that a property does not always define a set; $\{x \mid x \notin x\}$ does not define a set. A detailed treatment of set theory is beyond the scope of this thesis; references can be found in [183].

every $x \in X$ there is at most one $y \in Y$ such that $(x, y) \in f$. If there is such a y, then $f(x)$ is said to be defined and we write $y = f(x)$. If there is no such y, then we say that $f(x)$ is undefined. To denote that f is a partial function from X to Y, we write $f : X \hookrightarrow Y$. A partial function f from X to Y is called a *total function* if it is defined for every $x \in X$. To express that f is a total function from X to Y, we write $f : X \to Y$. All functions we introduce are assumed to be total unless explicitly stated otherwise. $dom(f)$ is used to denote the *domain* of $f : X \hookrightarrow Y$, i.e. $\{x \in X \mid (x, y) \in f\}$. Y^X denotes the set of all total functions from X to Y.
If $f : X \hookrightarrow Y$ and $g : X \hookrightarrow Y$ then we use $g[f]$ to denote the function f modified by g, i.e. $(g[f])(x) = g(x)$ if $g(x)$ is defined and $(g[f])(x) = f(x)$ otherwise. If $f : X \hookrightarrow Y$ and $g : Y \hookrightarrow Z$ then we use the (customary) notation $g \circ f$ to denote the (relation) composition fg; thus $g \circ f(x) = g(f(x))$. Furthermore, we use the 'variant notation' for functions, $f\{y/x\}$. If $f : X \hookrightarrow Y$, $x \in X$ and $y \in Y$ then $f\{y/x\}(x) = y$ and for all $x' \in X$, $x' \neq x$, $f\{y/x\}(x') = f(x')$. When $x_1, x_2, \ldots$ are distinct, we use $f\{y_1/x_1, y_2/x_2, \ldots\}$ as shorthand for $f\{y_1/x_1\}\{y_2/x_2\}\ldots$.

2.1.2 Induction

An important technique to apply in proofs or definitions over infinite sets is *induction*.

Inductive Definitions

Many sets, functions, and other mathematical objects are defined inductively. The underlying principle of an inductive definition is a set of *inference rules* of the following form:

$$\frac{\textit{Premises}}{\textit{Conclusions}}$$

A rule prescribes the derivation of larger facts or objects from smaller ones. Rules without premises are called *axioms*. The set of rules inductively defines all conclusions that can be derived by a finite number of applications of the rules. Inductive definitions will be used frequently and in many forms.

Syntactic Sets and Grammars

We often employ sets of syntactic objects do denote mathematical structures. These syntactic sets are defined inductively by a set of rules indicating how to construct their elements. Such a set of rules is often called a *grammar*. The following rules for instance, define a set E of arithmetic expressions.

$$\frac{n \in \mathbb{N}}{n \in E} \quad \frac{e_1 \in E \quad e_2 \in E}{e_1 + e_2 \in E} \quad \frac{e_1 \in E \quad e_2 \in E}{e_1 - e_2 \in E} \quad \frac{e_1 \in E \quad e_2 \in E}{e_1 \times e_2 \in E}$$

Instead of using inference rules, a grammar is usually written in Backus-Naur Form (BNF). In BNF, the rules for E are written as follows.

$$e ::= n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 \quad (n \in \mathbb{N})$$

This definition states that an arithmetic expression (e) can be a number (n), the sum of two arithmetic expressions ($e_1 + e_2$), the difference of two arithmetic expressions ($e_1 - e_2$), or the product of two arithmetic expressions ($e_1 \times e_2$). e, e_1, e_2 and n are variables that range over syntactic sets and are called *(syntactic) metavariables*. In the example, n ranges over a (given) set of (syntactic representations of) numbers. e, e_1 and e_2 range over arithmetic expressions in E. Thus the grammar gives an inductive definition of the set E as the least set satisfying the formation rules. To show that a particular expression is in E, we can use a *derivation tree* in which rule instances are placed on top of each other, using the conclusion of one rule as one of the premises of another. For instance, the derivation tree

$$\dfrac{\dfrac{1 \in \mathbb{N}}{1 \in E} \quad \dfrac{\dfrac{2 \in \mathbb{N}}{2 \in E} \quad \dfrac{3 \in \mathbb{N}}{3 \in E}}{2 \times 3 \in E}}{1 + 2 \times 3 \in E}$$

shows that $1 + 2 \times 3 \in E$ (from the facts that $1 \in \mathbb{N}$, $2 \in \mathbb{N}$ and $3 \in \mathbb{N}$).

Two elements e_1 and e_2 of a syntactic set are called *syntactically equivalent* (denoted $e_1 = e_2$) if they have been constructed in the same way from the syntactic rules. ($e_1 \neq e_2$ denotes that e_1 and e_2 are not syntactically equivalent.) The presented grammar for E is called an *abstract grammar*, because it allows for the same expression (for instance $1 + 2 \times 3$) to be constructed in different ways ($(1 + 2) \times 3$ or $1 + (2 \times 3)$). Such a grammar would not be suitable for a practical implementation. The grammar would have to be extended to resolve such ambiguities; such an extended grammar is called a *concrete grammar*. For a theoretical description, abstract grammars suffice. If necessary, we use parentheses to express the intended parsing of an expression (as we did to express the different ways to parse $1 + 2 \times 3$). The parentheses are not part of the syntactic expressions themselves. If the expression e is an element of some syntactic set, we use $e[f/x]$ to denote the syntactic element obtained from e by replacing all occurrences of the subexpression x with f.

Inductive Proofs

In order to prove properties of objects that are defined inductively, one proceeds along the lines of the inference rules. Assuming that the desired property holds for the premises of a rule, it is shown to hold for the conclusions of the rule as well. If this can be shown for every inference rule, then the property holds for every object derivable.

Mathematical Induction

The elementary form of induction is called *mathematical induction* or *natural induction*. It is based upon the inductive definition of the natural numbers. An inductive proof of a property $P(n)$ parameterised by a natural number $n \in \mathbb{N}$, proceeds by proving the particular property $P(0)$ and proving that for every $n \geq 0$, $P(n+1)$ holds if $P(n)$ does.

Structural Induction

A particular instance of the concept of induction is called *structural induction*. The grammars introduced in this section define a syntactic set by structural induction. It consists of a number of formation rules describing how to build larger syntactic constructs out of smaller ones. If we want to prove a property of all elements of such a syntactic set, we can proceed by induction on the syntactical structure. Suppose that we want to prove that for every arithmetic expression that can be derived in the grammar of this section, the quantity of numbers n is one more than the number of operators ($+$, $-$ and $\times$). Then we can show that it is true for the basic formation rule (n, having one number and no operators) and that it is true for $e_1 + e_2$ ($e_1 - e_2$, $e_1 \times e_2$) if we know it is true for e_1 and e_2. From these observations it follows that the property holds for every arithmetic expression in the syntactic set E.

2.1.3 Infinite words and languages

Objects that play an important role in the formalisation of reactive systems and their behaviour, are finite or infinite sequences of elements (symbols) of some collection we refer to as an alphabet. Here we introduce some notation to be used with such sequences. An *alphabet* Σ is a set $\{\sigma_1, \sigma_2, \sigma_3, \ldots\}$ of *symbols*. A (countable) infinite sequence of symbols in Σ is considered to be an ordered tuple having an infinite (ω) number of elements. Hence the set of all such sequences is denoted as Σ^ω. Such a sequence is called an *(ω-)word*. A set of such (ω-)words is called an (ω-)*language*.
Let $\bar{w} = \sigma_0\sigma_1\sigma_2 \ldots \in \Sigma^\omega$. We use $\bar{w}(k)$ to denote σ_k. Moreover, we use $\bar{w}^k$ if we refer to the *suffix* of $\bar{w}$ from position $k \in \mathbb{N}$, i.e. to the sequence $\sigma_k\sigma_{k+1}\sigma_{k+2}\ldots$. If $\bar{u} \in \Sigma^*$ is a finite word over Σ, then $\bar{u} \cdot \bar{w}$ denotes the concatenation of $\bar{u}$ and $\bar{w}$. $\inf(\bar{w})$ is used to denote the set of symbols in Σ that occur infinitely often in an ω-word $\bar{w}$:

$$\inf(\bar{w}) = \{\sigma \in \Sigma \mid \forall_{k \geq 0} \exists_{m > k} \bar{w}(m) = \sigma\}.$$

2.1.4 Time

Our aim is to describe real-time systems. Therefore it is important to be able to describe time formally. Time can be measured discretely, such as the ticks of a digital clock (possibly represented by the natural numbers $\mathbb{N}$), or continuous such as time in the physical world (or at least the abstraction of physical time that we generally use); represented for instance by the positive real numbers, $\mathbb{R}^{\geq 0}$.

Time Domain

Without worrying about physical or philosophical questions, we naively assume that time is linear, it has a beginning $t = 0$, but extends infinitely into the future; a practical assumption about the nature of time when studying reactive systems.
Formally, a time domain is a commutative monoid $(T, +, 0)$ ($+$ is a commutative and associative operator on T and 0 is the identity of $+$) with the following properties [147]:

- $t + t' = t \Leftrightarrow t' = 0$;

- the relation $\leq$ defined as $t \leq t' \Leftrightarrow \exists_{t'' \in T} t + t'' = t'$ is a total order; for every $t, t' \in T$, $t \leq t'$ or $t' \leq t$.

Then the following properties follow from the previous ones:

- 0 is the least element of T;
- for any t, t', if $t \leq t'$ then the element t'' such that $t + t'' = t'$ is unique; it is denoted by $t' - t$.

$\mathbb{N}$, $\mathbb{Q}^{\geq 0}$ and $\mathbb{R}^{\geq 0}$ with their usual 0-elements, ordering and addition are proper time domains. We use T^+ to denote $T \backslash \{0\}$. Furthermore, a time domain is called discrete if every instant in time has one particular successor instant; $\forall_{t \in T} \exists_{t' \in T} (t < t' \wedge \forall_{t'' \in T} t < t'' \Rightarrow t' \leq t'')$. It is called dense if time has a more continuous nature, namely if for any two instants one can always find an instant in time in between; $\forall_{t,t' \in T} t < t' \Rightarrow \exists_{t'' \in T} t < t'' < t'$. Thus, $\mathbb{N}$ is a discrete time domain and both $\mathbb{Q}^{\geq 0}$ and $\mathbb{R}^{\geq 0}$ are dense.

Intervals

To define properties of real-time systems, one needs the concept of an interval of time. If the time domain is dense, one can discriminate between open and closed boundaries. An *interval* I is a convex subset of the time domain T. A finite interval with *lower bound* a and *upper bound* b is denoted as

- $[a, b] = \{t \in T \mid a \leq t \leq b\}$ if it is both left-closed and right-closed and $a \leq b$;
- $[a, b) = \{t \in T \mid a \leq t < b\}$ if it is left-closed and right-open and $a < b$;
- $(a, b] = \{t \in T \mid a < t \leq b\}$ if it is left-open and right-closed and $a < b$;
- $(a, b) = \{t \in T \mid a < t < b\}$ if it is both left-open and right-open and $a < b$.

If I is such an interval, then $l(I)$ denotes the lower bound a of I and $u(I)$ denotes the upper bound b. The *length* of interval I is $b - a$ and is referred to as $|I|$. An infinite interval with lower bound a and no upper bound is denoted as

- $[a, \infty) = \{t \in T \mid a \leq t\}$ if it is left-closed;
- $(a, \infty) = \{t \in T \mid a < t\}$ if it is left-open.

If I is such an infinite interval, then $l(I)$ denotes the lower bound a of I. $I - t$ denotes the interval $\{t' - t \mid t' \in I \text{ and } t' \geq t\}$.

2.1.5 Timed Words

The ω-words introduced earlier can only express discrete sequences of symbols. We can now extend words with (dense) timing information. An *interval sequence* $\overline{I} = I_0 I_1 I_2 \ldots$, is a *diverging* sequence of *consecutive* finite intervals, such that I_0 is left-closed and $l(I_0) = 0$. The interval sequence $\overline{I}$ is diverging if for any $t \geq 0$ there is an interval $\overline{I}(k)$ such that $t \in \overline{I}(k)$. Consecutive means that for any two successive

intervals $\overline{I}(k)$ and $\overline{I}(k+1)$, $u(\overline{I}(k)) = l(\overline{I}(k+1))$ and either $\overline{I}(k)$ is right-closed and $\overline{I}(k+1)$ is left-open or $\overline{I}(k)$ is right-open and $\overline{I}(k+1)$ is left-closed.
A *timed word* $\bar{v}$ over an alphabet Σ, is a pair $(\bar{w}, \overline{I})$ consisting of an ω-word $\bar{w}$ over Σ and an interval sequence $\overline{I}$. For any $t \in T$, we use $\bar{v}(t)$ to denote the symbol present at time t, this is $\bar{w}(k)$ if $t \in \overline{I}(k)$. Two timed words $\bar{v}_1$ and $\bar{v}_2$ are called *equivalent* ($\bar{v}_1 \equiv \bar{v}_2$) if for all $t \geq 0$, $\bar{v}_1(t) = \bar{v}_2(t)$. For $t \in T$,

- the suffix $\overline{I}^t$ of interval sequence $\overline{I}$ is the following sequence of intervals:

  $$(\overline{I}(k) - t) \ (\overline{I}(k+1) - t) \ (\overline{I}(k+2) - t) \ \ldots$$

  where k is such that $t \in \overline{I}(k)$;
- the suffix $\bar{v}^t$ of a timed word $\bar{v}$ is the timed word $(\bar{w}^k, \overline{I}^t)$ (k such that $t \in \overline{I}(k)$).

2.2 Labelled Transition Systems

To automatically analyse systems, we need an abstract (mathematical) model of them. A possible abstract view on the behaviour of a system is to regard it as an entity having some internal state and, depending on that state, is able to engage in transitions leading to other states. Such a transition might be autonomous or stimulated by the environment. It is assumed that the relevant behaviour of the system is exposed during such a state change. A mathematical structure that tries to capture this abstract view on a system is a labelled transition system.

2.2.1 Untimed Labelled Transition Systems

A labelled transition system (LTS) is a graph consisting of nodes (states) and labelled edges (transitions). The label represents the observable behaviour associated with the transition. The number of nodes of such a structure can be (uncountably) infinite. Formally, a labelled transition system is a triple $(S, L, \longrightarrow)$ consisting of a set S of states, a set L of labels and a labelled transition relation $\longrightarrow \subseteq S \times L \times S$. A transition $(s, \lambda, s') \in \longrightarrow$ denotes a possible transition from state s to state s' decorated with the label λ. For $\lambda \in L$, we write $\xrightarrow{\lambda}$ to denote the binary relation on S defined as

$$\xrightarrow{\lambda} = \{(s_1, s_2) \mid (s_1, \lambda, s_2) \in \longrightarrow\}.$$

Thus the transition from state s_1 to s_2 labelled λ is alternatively denoted as $s_1 \xrightarrow{\lambda} s_2$. Moreover we introduce the following notation. We write $s \xrightarrow{\lambda}$ if $s \xrightarrow{\lambda} s'$ for some $s' \in S$ and $s \not\xrightarrow{\lambda}$ if not $s \xrightarrow{\lambda}$.
At times it is convenient to discriminate certain transitions that are considered unobservable. Then, the LTS has a special label (or a special set of labels) that denotes (denote) internal unobservable transitions. This special label is usually denoted as τ.
Figure 2.1 shows a graphical representation of an example of a labelled transition system. The states are represented as circles and the transitions as arrows between states. The labels a, b, c, d and τ are the actions the system engages in during its transitions.

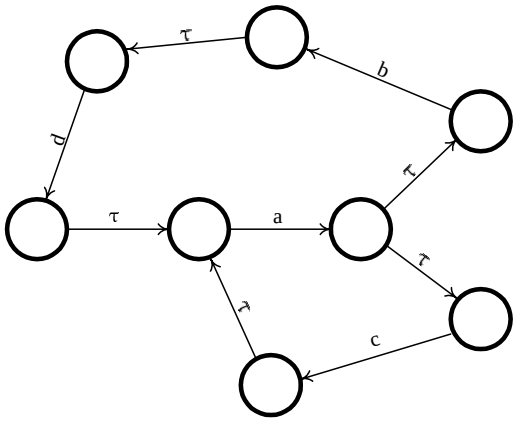


Figure 2.1: Example of a labelled transition system

A possible generalisation of the labelled transitions system is to assume that the system not only exposes some of its behaviour while making a transition from one state to another, but also whilst the system resides in a particular state. This would entail the extension of the LTS with a set of state labels and a mapping labelling every state with a particular label from that set.
The following properties of labelled transition systems are important.

- DEADLOCK. A state $s \in S$ in a labelled transition system $(S, L, \longrightarrow)$ is called a deadlock state if it has no outgoing transitions, i.e. if there is no $\lambda \in L$ such that $s \xrightarrow{\lambda}$. After reaching such a state, the system will never again engage in any transition.

- DIVERGENCE. A state $s_0 \in S$ in a labelled transition system $(S, L, \longrightarrow)$ is diverging if it may execute an infinite sequence of internal transitions. Formally, it diverges if there exists a sequence of states $s_1, s_2, s_3, \ldots$ such that for all $k \geq 0$, $s_k \xrightarrow{\tau} s_{k+1}$.

A trace $\theta = (\overline{s}, \overline{\lambda})$ describes an (infinite) path through an LTS. Formally, it consists of an infinite sequence $\overline{s}$ of states and an infinite sequence $\overline{\lambda}$ of labels, such that

$$\overline{s}(0) \xrightarrow{\overline{\lambda}(0)} \overline{s}(1) \xrightarrow{\overline{\lambda}(1)} \ldots.$$

Sometimes it is convenient to represent a trace as a sequence of pairs (s, λ) rather than a pair of sequences (for instance to write θ^k to denote a particular suffix of a trace). We will freely use (and mix) both notations.

2.2.2 Timed Labelled Transition Systems

The labelled transition system yields a very abstract representation of a system. Particularly it does not explicitly capture timing of the system's behaviour. Since we are especially interested in real-time systems, we make use of an extension of the LTS

that does incorporate timing information. A *timed labelled transition system* (TLTS) is a 5-tuple $\left(S, T, L, \xrightarrow[L]{}, \xrightarrow[T^+]{}\right)$. Here, S, L and $\xrightarrow[L]{}$ are as in untimed LTSs. T is a time domain and the time transition relation $\xrightarrow[T^+]{} \subseteq S \times T^+ \times S$ is used to represent passage of time.
In this model of a timed system, action transitions ($\xrightarrow[L]{}$) are assumed to be instantaneous (taking no time) and the passage of an amount t of time is represented by a transition $\xrightarrow[T^+]{t}$.
When reasoning about timed transition systems, the following properties of the transition relations are frequently used (see also [129, 147, 186]).

- TIME DETERMINISM. Time transitions are deterministic; no choices are made as time progresses; for every $s, s', s'' \in S$ and $t \in T$, if $s \xrightarrow[T^+]{t} s'$ and $s \xrightarrow[T^+]{t} s''$ then $s' = s''$.
- TIME ADDITIVITY. (Called time continuity in [186].) Time transitions can be split into smaller transitions, or combined to larger ones; for every $s, s' \in S$, and $t_1, t_2 \in T^+$, $s \xrightarrow[T^+]{t_1+t_2} s'$ iff there is some $s'' \in S$ such that $s \xrightarrow[T^+]{t_1} s'' \xrightarrow[T^+]{t_2} s'$.
- MAXIMAL PROGRESS. (Called action urgency in [147].) Maximal progress with respect to a set $L' \subseteq L$ of labels requires that transitions labelled with elements of L' are not postponed; if $s \xrightarrow[L]{\lambda}$ for some $s \in S$ and $\lambda \in L'$, then there is no $t \in T$ such that $s \xrightarrow[T^+]{t}$. Moreover if time is deterministic, then it is often also required that $s \not\xrightarrow[T^+]{t}$ if $s \xrightarrow[T^+]{t'} s' \xrightarrow[L]{\lambda}$ for some $0 < t' < t$ and $\lambda \in L'$. Note that in case of time additivity, this is automatically implied by the first requirement.
- PERSISTENCY. A TLTS is called persistent if the progress of time does not disable other transitions; for all $s_1, s_2 \in S, \lambda \in L$ and $t \in T$, if $s_1 \xrightarrow[L]{\lambda}$ and $s_1 \xrightarrow[T^+]{t} s_2$ then $s_2 \xrightarrow[L]{\lambda}$.

In the remainder we often leave out the subscript identifying the transition relation as being the action transition relation or the time transition relation. It will be clear from the label (being from L or T^+) which relations is intended. Hence, we shall write $s_1 \xrightarrow{\lambda} s_2$ and $s_1 \xrightarrow{t} s_2$ to indicate $s_1 \xrightarrow[L]{\lambda} s_2$ and $s_1 \xrightarrow[T^+]{t} s_2$ respectively. Moreover we use γ as a typical element of $L \cup T^+$.

A timed trace $\zeta = (\overline{s}, \overline{\gamma})$ describes an (infinite) path through an LTS. It consists of an infinite sequence $\overline{s}$ of states and an infinite sequence $\overline{\gamma}$ of labels or elements of the time domain, such that

$$\overline{s}(0) \xrightarrow{\overline{\gamma}(0)} \overline{s}(1) \xrightarrow{\overline{\gamma}(1)} \ldots.$$

A timed trace $(\overline{s}, \overline{\gamma})$ is called Zeno [2] if $\sum_{i \in \mathbb{N}} T(\gamma_i) < \infty$ ($T(\gamma)$ equals γ if $\gamma \in T$ and 0 otherwise). If this is the case, the total amount of time elapsed during the trace does

[2] Such a trace is called Zeno after the famous paradox by the greek philosopher Zeno of Elea about Achilles and the Tortoise in which he defends the belief that motion and change cannot exist.

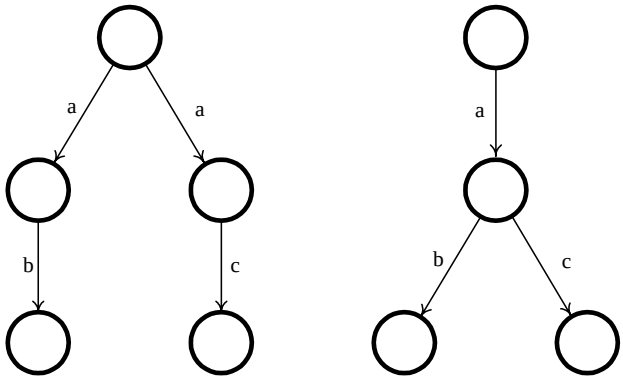


Figure 2.2: Two systems with similar behaviour

not progress beyond some point $t \in T$. If the time domain is discrete, then this can only occur if after some point, no more time transitions are taken. If the time domain is dense, then it might be the case that although infinitely often time transitions are made, the trace does still not progress beyond some point in time. Under certain circumstances, one would like to ignore Zeno traces as they represent 'unrealistic' behaviour.

2.3 Behaviour and Equivalences

Reactive systems operate in continuous interaction with their environment. Their behaviour cannot be described transformationally, such as in the case of traditional software systems which, when fed with some input data, compute for a while and then terminate delivering output results. Therefore, the behaviour of reactive systems is typically modelled as an infinitely long (never ending) interaction with the environment. There are many different models to formalise what the behaviour of a system is. What the best model is depends on the intended use.

Adopting the labelled transition system as the model of a reactive system, one might take as its behaviour, the set of all sequences of labels observed along any possible path through the transition system. For both systems in figure 2.2, this behaviour when started from the topmost state, is represented by the set $\{ab, ac\}$. Taking a slightly different point of view, one might argue that these systems should not be considered equivalent. If we imagine an observer interacting with the system, after observing a transition 'a', one of the systems is willing to engage in both a 'b' transition and a 'c' transition, whereas the other system is only willing to engage in either a 'b' transition or a 'c' transition. From this point of view, behaviour can be formalised as an infinite tree of possible transitions, starting from some initial state of the LTS and branching via its transitions to new states and so forth.

Sometimes one would like to consider certain transitions of a system to be internal to the system and unobservable from the outside. Systems that only differ in terms of internal transitions may then still be considered equivalent from the point of view of an external observer.

Since behaviours are generally infinite, there are some other questions about behaviours that can be raised. For example, assume that on an infinite path through a labelled transition system, a particular state is visited infinitely often and although this state has several outgoing transitions, the execution always follows the same transition to leave this state. Under certain circumstances, this behaviour can be regarded as 'unfair' with respect to the other outgoing transitions and therefore this behaviour is not a realistic one and should be neglected. Such extra conditions are called fairness conditions [79].
As there are multiple definitions of what a system's behaviour is, there are also many definitions of when to call two different systems equivalent. Taking the former, linear view, an obvious definition of equivalence is to call two systems equivalent when the sets of all sequences of observations along execution paths are identical. This is generally called *trace equivalence*.
An equivalence relation which pays more attention to the branching structure and which is based on the idea of an external observer interacting with the system, is called *observation equivalence*. Two systems are found equivalent if the observer cannot distinguish them by interacting with them. When we discuss the execution of specifications in chapter 4, we show that the executed behaviour is observation equivalent to the specification. It is discussed in more detail in the next section.
Besides equivalence relations on transition systems, one can also define pre-orders that express when a transition system is a good implementation of another, or more general or precise than the other.

2.3.1 Bisimulations

The formalisation of observation equivalence is established by means of a relation on states of the labelled transition system called a *bisimulation relation* [153, 140]. The idea behind bisimulation is to regard two states as equivalent if they can engage in the same observable transitions (transitions with identical labels) and after performing similar transitions end up in equivalent states again. If one chooses to think of every transition as observable, the relation is called a strong bisimulation. If one abstracts from internal transitions, it is called weak bisimulation or branching bisimulation if one abstracts from internal transitions but retains some of the internal branching structure.
Formally, a strong bisimulation relation on a labelled transition system is defined as follows.

2.3.1 DEFINITION *A symmetrical relation $\mathcal{S} \subseteq S \times S$ is a* strong bisimulation relation *on an LTS $(S, L, \longrightarrow)$ if for every $(s_1, s_2) \in \mathcal{S}$ and λ in L, if $s_1 \xrightarrow{\lambda} s_1'$ then there is some s_2' such that $s_2 \xrightarrow{\lambda} s_2'$ and $(s_1', s_2') \in \mathcal{S}$.* □

Two states s_1 and s_2 are said to be (strong bisimulation) equivalent ($s_1 \sim s_2$) if there is a strong bisimulation $\mathcal{S}$ such that $(s_1, s_2) \in \mathcal{S}$.
Instead of defining a bisimulation relation as a relation on some LTS, it can also be defined as a relation between different LTSs. Let $LTS_1 = (S_1, L_1, \xrightarrow[1]{})$ and $LTS_2 = (S_2, L_2, \xrightarrow[2]{})$ be two LTSs then $\mathcal{S} \subseteq S_1 \times S_2$ is a bisimulation between LTS_1 and LTS_2 if for every $(s_1, s_2) \in \mathcal{S}$ and $\lambda \in L_1 \cup L_2$,

- if $s_1 \xrightarrow{\lambda} s_1'$ then there is some s_2' such that $s_2 \xrightarrow{\lambda} s_2'$ and $(s_1', s_2') \in \mathcal{S}$;
- if $s_2 \xrightarrow{\lambda} s_2'$ then there is some s_1' such that $s_1 \xrightarrow{\lambda} s_1'$ and $(s_1', s_2') \in \mathcal{S}$.

Bisimulation captures a notion of equivalence that equates two systems if they cannot be distinguished by interacting with them. Thinking of internal, unobservable transitions, strong bisimulation can be regarded as too strong. To overcome this problem one can define equivalence in terms of a transition relation which abstracts from internal transitions. Define a τ-abstracted transition relation $\Longrightarrow$ as the smallest relation on S such that

- $s_1 \stackrel{\tau}{\Longrightarrow} s_2$ if $s_1 (\xrightarrow{\tau})^* s_2$;
- $s_1 \stackrel{\lambda}{\Longrightarrow} s_2$ if $\lambda \neq \tau$ and $s_1 (\xrightarrow{\tau})^* \xrightarrow{\lambda} (\xrightarrow{\tau})^* s_2$.

Weak bisimulation can now be defined in terms of the abstracted transition relation.

2.3.2 DEFINITION *A symmetrical relation $\mathcal{S} \subseteq S \times S$ is a* weak bisimulation relation *on an LTS $(S, L, \longrightarrow)$ if for every $(s_1, s_2) \in \mathcal{S}$ and λ in L, if $s_1 \stackrel{\lambda}{\Longrightarrow} s_1'$ then there is some s_2' such that $s_2 \stackrel{\lambda}{\Longrightarrow} s_2'$ and $(s_1', s_2') \in \mathcal{S}$.* □

Alternatively, weak bisimulation can be defined with the abstracted LTS, $(S, L, \Longrightarrow)$. Then (see [140]) $\mathcal{S}$ is a weak bisimulation on $(S, L, \longrightarrow)$ iff $\mathcal{S}$ is a strong bisimulation on $(S, L, \Longrightarrow)$. Two states s_1 and s_2 are called (weak bisimulation) equivalent ($s_1 \approx s_2$) if there is a weak bisimulation $\mathcal{S}$ such that $(s_1, s_2) \in \mathcal{S}$.

2.3.2 Timed Bisimulations

The notion of bisimulation can be extended to timed transition systems (see e.g. [186, 103, 21]). Guided by the same underlying ideas, two states are (strongly) equivalent if they possess the same time and action transitions, leading to equivalent states. Formally this is expressed as follows.

2.3.3 DEFINITION *A symmetrical relation $\mathcal{S} \subseteq S \times S$ is a* strong timed bisimulation relation *if for every $(s_1, s_2) \in \mathcal{S}$, λ in L and $t \in T^+$,*

- *if $s_1 \xrightarrow{\lambda} s_1'$ then there is some s_2' such that $s_2 \xrightarrow{\lambda} s_2'$ and $(s_1', s_2') \in \mathcal{S}$;*
- *if $s_1 \xrightarrow{t} s_1'$ then there is some s_2' such that $s_2 \xrightarrow{t} s_2'$ and $(s_1', s_2') \in \mathcal{S}$.* □

States s_1 and s_2 are (strong) timed bisimulation equivalent (written $s_1 \sim_T s_2$, or just $s_1 \sim s_2$ if it is obvious that the intended relation is a timed one) if there is some strong timed bisimulation relation $\mathcal{S}$ such that $(s_1, s_2) \in \mathcal{S}$.
Correspondingly one can define a weak bisimulation relation, by abstracting from internal actions [186]. A transition relation that abstracts from internal transitions should consider a sequence of time transitions and internal action transitions as one large time transition. The time that passes in this transition should equal the total amount of time passed in the sequence. Let the abstracted time transition relation $\stackrel{}{\Longrightarrow}_{T^+}$ be defined as follows. $\stackrel{}{\Longrightarrow}_{T^+}$ is the smallest relation on S such that $s_1 \stackrel{t}{\Longrightarrow}_{T^+} s_2$

if $s_1(\xrightarrow{\tau})^* \xrightarrow[T^+]{t_1} (\xrightarrow{\tau})^* \ldots (\xrightarrow{\tau})^* \xrightarrow[T^+]{t_n} (\xrightarrow{\tau})^* s_2$, where $t = t_1 + \ldots + t_n > 0$.
Note that time determinism is not necessarily preserved in the weak timed transition relation due to the possible non-determinism caused by the τ transitions.
Using the new transition relations, we can define weak timed bisimulation relations.

2.3.4 DEFINITION *A symmetrical relation $\mathcal{S} \subseteq S \times S$ is a* weak timed bisimulation relation *if for every $(s_1, s_2) \in \mathcal{S}$, λ in L and $t \in T^+$,*

- *if $s_1 \stackrel{\lambda}{\Longrightarrow} s_1'$ then there is some s_2' such that $s_2 \stackrel{\lambda}{\Longrightarrow} s_2'$ and $(s_1', s_2') \in \mathcal{S}$;*
- *if $s_1 \xRightarrow[T^+]{t} s_1'$ then there is some s_2' such that $s_2 \xRightarrow[T^+]{t} s_2'$ and $(s_1', s_2') \in \mathcal{S}$.* □

The relation $\stackrel{\lambda}{\Longrightarrow}$ is defined as in the untimed case. s_1 and s_2 are weak timed bisimulation equivalent ($s_1 \approx_T s_2$, or just $s_1 \approx s_2$) if there is some weak timed bisimulation relation $\mathcal{S}$ such that $(s_1, s_2) \in \mathcal{S}$.

2.4 Process Calculi

Behaviour of reactive systems can be formalised in terms of labelled transition systems. The specification of reactive systems by directly writing down their labelled transition systems however is impractical. Such an explicit description could become large and even infinite. The behaviour of the composition of two subsystems operating concurrently would have to be described as one monolithic transition system. There is a need for a more compositional approach to specify the behaviour of reactive systems. Such an approach is offered by *process calculi* such as CCS [140], CSP [104] and ACP [27] among others. Process calculi allow one to compositionally specify labelled transition systems. For instance, given two systems specified as process calculus expressions, one can easily express the behaviour resulting from taking both systems executing concurrently and possibly communicating together. Several flavours of process calculi exist. In this section we describe CCS ([140]) as a representant that resembles the calculus we use in the following chapters.
Concurrency is an important aspect of distributed systems which often consist of components operating concurrently and relatively independently. As the composed system is described as an LTS again, the independent individual transitions have to be combined to transitions of the total system. An abstraction that is often used is *interleaving concurrency*. It models independent concurrent actions by considering the occurrence of individual action in arbitrary order, but not at the same time. Only explicitly synchronising actions between components occur together.

2.4.1 Syntax

The set of process expressions is parameterised with a set $\mathcal{A}$ of *names* (ranged over by a, b, ...) and a set $\overline{\mathcal{A}} = \{\overline{\mathrm{a}} \mid \mathrm{a} \in \mathcal{A}\}$ of *conames* of $\mathcal{A}$. Let $\bar{}$ be an operator which yields for every name a its coname $\overline{\mathrm{a}}$ and for every coname $\overline{\mathrm{a}}$ the corresponding name ($\overline{\overline{\mathrm{a}}} = \mathrm{a}$). The names denote individual actions; names and their corresponding conames are compatible actions that can be executed together. The set $\mathcal{L}$ of *labels*

(ranged over by ℓ) is the set $\mathcal{A} \cup \overline{\mathcal{A}}$ of all names and conames. The *silent* or *internal* action is denoted as τ. Together they form the set of all *actions* $Act = \mathcal{L} \cup \{\tau\}$ (ranged over by α). Moreover, there exists a set $\mathcal{K}$ of *process constants* (sometimes called agent constants) ranged over by A. The set $\mathcal{P}$ of process expressions, ranged over by P, is produced by the following grammar ($f : Act \to Act$ and $L \subseteq \mathcal{A}$).

$$P \quad ::= \quad \mathbf{0} \quad | \quad \alpha \cdot P \quad | \quad P_1 + P_2 \quad | \quad P_1 || P_2 \quad | \quad P[f] \quad | \quad P \backslash L \quad | \quad A$$

0 has no interesting behaviour by itself. It is unable to perform any action and is used as a basic building block to construct more complex processes. $\alpha \cdot P$ is the process that is able to perform an action α, after doing so it will behave like process P. $P_1 + P_2$ is the process that behaves like process P_1 or as P_2. $P_1 || P_2$ is called the *parallel composition* of the processes P_1 and P_2. It represents the process whose behaviour is given by the concurrent operation of both processes. $P[f]$ is a process which behaves like the process P, but with the names of its actions relabelled according to the relabelling function f; f must be such that $\overline{f(\ell)} = f(\overline{\ell})$ for every $\ell \in \mathcal{L}$ and $f(\tau) = \tau$. $P \backslash L$ is the process that behaves just like P, but its actions are restricted to actions that are not included in the set L; this is called *hiding*. Finally, we have the behaviour of the process constant A; every $A \in \mathcal{K}$ is defined as a process expression P_A (in which A may occur again): $A \stackrel{\text{def}}{=} P_A$[3]. Then the behaviour of A equals the behaviour of P_A. The process constants allow the expression of infinite behaviour. For instance the process constant defined by $A \stackrel{\text{def}}{=} \mathrm{a} \cdot A$ can engage in an arbitrary number of 'a' actions. The interpretation of process expressions as labelled transitions systems is formalised in the next section.

2.4.2 Semantics

The semantics of a process expression is given in the form of a labelled transition system. In this transition system, the process expressions act as states and the transitions are labelled with actions from Act. Thus the semantics of the calculus is given by the LTS $(\mathcal{P}, Act, \longrightarrow)$. To complete the semantics of the calculus we need to define the labelled transition relation $\longrightarrow \subseteq \mathcal{P} \times Act \times \mathcal{P}$.

2.4.3 Structural Operational Semantics

The labelled transition relation is defined using a structural operational semantics (SOS) as introduced by Plotkin [154]. The transitions of a particular process expression are defined inductively along the structure of the process expression. The premises of the SOS rules consist of one or more statements about the presence of transitions of $\longrightarrow$ and / or other conditions. The conclusions state transitions that can be concluded from the premise. The statements contain syntactic metavariables. The actual rule instances are obtained from the rules by instantiation of the variables with

[3] Another interpretation of process constants is to view the behaviour of A as the solution to the set of equations $\{A \sim P_A \mid A \in \mathcal{K}\}$. Under certain conditions ([140]) the solution is unique or under different conditions, one can show the equations to have a unique 'smallest' solution that can be taken as the behaviour of A.

$$\frac{-}{\alpha \cdot P \xrightarrow{\alpha} P}\ Act \qquad \frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 + P_2 \xrightarrow{\alpha} P_1',\ P_2 + P_1 \xrightarrow{\alpha} P_1'}\ Sum$$

$$\frac{P_1 \xrightarrow{\alpha} P_1'}{P_1||P_2 \xrightarrow{\alpha} P_1'||P_2,\ P_2||P_1 \xrightarrow{\alpha} P_2||P_1'}\ Com_1 \qquad \frac{P_1 \xrightarrow{\ell} P_1',\ P_2 \xrightarrow{\overline{\ell}} P_2'}{P_1||P_2 \xrightarrow{\tau} P_1'||P_2'}\ Com_2$$

$$\frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}\ Rel \qquad \frac{P \xrightarrow{\alpha} P',\ \alpha \notin L, \overline{\alpha} \notin L}{P\backslash L \xrightarrow{\alpha} P'\backslash L}\ Res \qquad \frac{P \xrightarrow{\alpha} P',\ A \stackrel{\text{def}}{=} P}{A \xrightarrow{\alpha} P'}\ Con$$

Table 2.1: Semantics of CCS

processes and actions over which they range. For instance the rule

$$\frac{P_1 \xrightarrow{\alpha} P_1'}{P_1 + P_2 \xrightarrow{\alpha} P_1'}$$

states that whenever $P_1 \xrightarrow{\alpha} P_1'$ is a valid transition of some process expression of the form P_1 for some action α, then $P_1 + P_2 \xrightarrow{\alpha} P_1'$ is a transition of any process expression of the form $P_1 + P_2$. The axiom

$$\frac{-}{\alpha \cdot P \xrightarrow{\alpha} P}$$

states that the process $\alpha \cdot P$ can perform the action α and after that behave as P.
Given such a collection of axioms and rules, the transition relation is defined as the smallest relation satisfying all the axioms and rules. The complete collection of semantical rules of CCS are displayed in table 2.1.
Figure 2.3 shows the semantics of the process expression $(\mathrm{a} \cdot \mathrm{b} \cdot \mathbf{0} \ \ || \ \ \mathrm{c} \cdot \overline{\mathrm{b}} \cdot \mathbf{0})\backslash\{\mathrm{b}\}$. The actions 'a' and 'c' of the concurrent subprocesses are executed in any order and after that the subprocesses can only synchronise on their 'b' and '$\overline{\mathrm{b}}$' actions (individual 'b' or '$\overline{\mathrm{b}}$' actions are not possible because of the hiding of $\{\mathrm{b}\}$.
The transition labelled 'a' from the initial expression can be derived from the rules as demonstrated by the following derivation tree.

$$\cfrac{\cfrac{\cfrac{-}{\mathrm{a} \cdot \mathrm{b} \cdot \mathbf{0} \xrightarrow{\mathrm{a}} \mathrm{b} \cdot \mathbf{0}}\ Act}{(\mathrm{a} \cdot \mathrm{b} \cdot \mathbf{0} \ \ || \ \ \mathrm{c} \cdot \overline{\mathrm{b}} \cdot \mathbf{0}) \xrightarrow{\mathrm{a}} (\mathrm{b} \cdot \mathbf{0} \ \ || \ \ \mathrm{c} \cdot \overline{\mathrm{b}} \cdot \mathbf{0})}\ Com_1}{(\mathrm{a} \cdot \mathrm{b} \cdot \mathbf{0} \ \ || \ \ \mathrm{c} \cdot \overline{\mathrm{b}} \cdot \mathbf{0})\backslash\{\mathrm{b}\} \xrightarrow{\mathrm{a}} (\mathrm{b} \cdot \mathbf{0} \ \ || \ \ \mathrm{c} \cdot \overline{\mathrm{b}} \cdot \mathbf{0})\backslash\{\mathrm{b}\}}\ Res$$

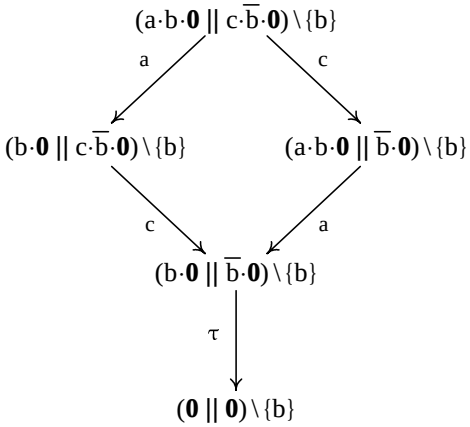


Figure 2.3: LTS induced by the process expression $(a \cdot b \cdot \mathbf{0} \quad || \quad c \cdot \overline{b} \cdot \mathbf{0}) \backslash \{b\}$

2.4.4 Real-time Process Calculi

Real-time extensions of the process calculi have been made to describe real-time systems in terms of timed labelled transition systems [18, 19, 49, 62, 96, 139, 147, 159, 186]. Thus the semantics of a real-time process calculus is a TLTS

$$\left(\mathcal{TP}, T, Act, \xrightarrow[Act]{}, \xrightarrow[T^+]{} \right).$$

Real-time process calculi include all operators that are used in the untimed calculi and give them a timed interpretation. Furthermore, new operators are introduced that quantitatively control a system's timed behaviour. An overview and comparison of possible timed operators can be found in [147]. What new operators are needed depends on the existing operators and the required expressiveness. We introduce here only a delay operator $\langle t \rangle P$, yielding a calculus similar to the one in [186], with the syntax

$$\begin{array}{rcl} P & ::= & \mathbf{0} \quad | \quad \alpha \cdot P \quad | \quad \langle t \rangle P \quad | \quad P_1 + P_2 \quad | \quad P_1 || P_2 \quad | \\ & & P[f] \quad | \quad P \backslash L \quad | \quad A \end{array}$$

The semantics of the real-time calculus (TCCS) is time-deterministic and exhibits maximal progress with respect to internal transitions. A process P is called *urgent* with respect to some set $U \subseteq Act$ of urgent actions (denoted as $Urgent^U(P)$) if $P \xrightarrow{u}$ for some $u \in U$. A process is urgent within time t (denoted as $Urgent^U(P,t)$) if it is urgent (now) or if there exist $t' < t$ and P' such that $P \xrightarrow{t'} P'$ and P' is urgent, i.e. $Urgent^U(P')$. This urgency predicate is comparable to similar ones in [3, 96, 186].
The urgency of actions and deterministic time give rise to a two-phase execution model. The definitions of the action and time transition relations is such that internal action transitions have precedence over time transitions. An execution of a system will take internal action transitions for as long as there are such transitions available. Only when there are no more internal transitions available, time transitions

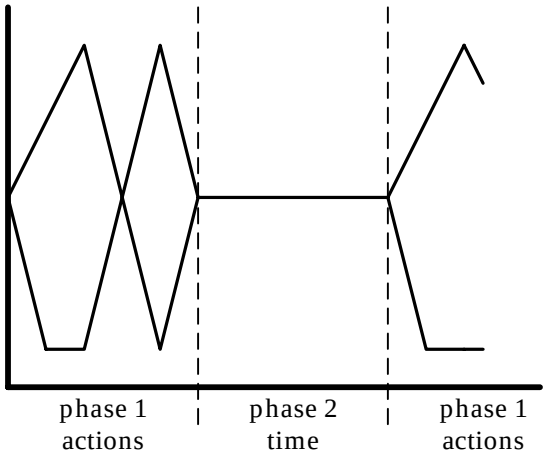


Figure 2.4: Two phases of the execution of timed systems (from [147])

become possible. Every time transition will be available up to the point where new internal transition become available. The effect is that the execution occurs in alternating 'phases' (this is illustrated in figure 2.4, [147]) starting with a phase in which internal actions are performed asynchronously and in which time does not progress. This phase is followed by a phase in which time passes synchronously for all processes. After new internal actions have become available, a new asynchronous phase starts, and so forth. In the asynchronous phase, possibly non-deterministic choices are made between available transitions. The synchronous phase on the other hand is entirely deterministic.

The timed interpretation of the existing and new operators is given by the rules in table 2.2. Together with the rules of the untimed calculus in table 2.1, they define the semantics of the calculus. Note that the rules represent just one timed interpretation, others are possible as well.

The first and third rule of the delay operator guarantee the time continuity of the transition system. The time behaviour of the $+$ operator is defined in such a way that the passage of time does not enforce a choice. Care has to be taken with the time transition rules of the parallel composition. The idea behind the rule is that if both subprocesses can let some amount of time pass, then their parallel composition is allowed to let the amount of time pass and the time passes synchronously in both subprocesses. To ensure that urgent actions take place as soon as possible, an extra requirement is added. The subprocesses of the parallel composite may become ready to synchronise only after some delay. This situation is dealt with by the urgency condition[4]. The hiding and relabelling operators operate on actions and have no influence on time.

[4]The parallel composition rule exploits a negative premise. There might be some concern regarding the existence of a (unique) solution of the transition relation defined by the rules [30]. That such a transition relation does indeed exist can be shown as in [174, 175] by the fact that the negative premise occurs in a rule of the time transition relation, but refers only negatively to action transitions whereas the action transitions are defined without referring to the timed transition relations. The existence of a transition system can be shown by a stratification separating action and time transition rules.

$$\frac{-}{\mathbf{0} \xrightarrow{t} \mathbf{0}}\ Ddl \qquad \frac{-}{\ell \cdot P \xrightarrow{t} \ell \cdot P}\ Act \qquad \frac{t' < t}{\langle t \rangle P \xrightarrow{t'} \langle t - t' \rangle P}\ Del_1$$

$$\frac{-}{\langle t \rangle P \xrightarrow{t} P}\ Del_2 \qquad \frac{P \xrightarrow{t} P'}{\langle t' \rangle P \xrightarrow{t+t'} P'}\ Del_3 \qquad \frac{P_1 \xrightarrow{t} P_1',\ P_2 \xrightarrow{t} P_2'}{P_1 + P_2 \xrightarrow{t} P_1' + P_2'}\ Sum$$

$$\frac{P_1 \xrightarrow{t} P_1',\ P_2 \xrightarrow{t} P_2',\ \neg Urgent^U(P_1||P_2, t)}{P_1||P_2 \xrightarrow{t} P_1'||P_2'}\ Com_1$$

$$\frac{P \xrightarrow{t} P'}{P[f] \xrightarrow{t} P'[f],\ P\backslash L \xrightarrow{t} P'\backslash L}\ Rel,Res \qquad \frac{P \xrightarrow{t} P',\ A \stackrel{\text{def}}{=} P}{A \xrightarrow{t} P'}\ Con$$

Table 2.2: Semantics of TCCS

2.5 Modal and Temporal Logic

We have seen formal models for distributed reactive systems, discussed their behaviour and equivalence relations and we have introduced process calculi as a formalism for the specification of reactive systems. Extensions of the introduced concepts into the real-time domain have been discussed. In order to verify these systems for properties they may possess, we need a way to effectively (formally) capture the desired or undesired behaviour. During transitions one can observe the labels of the executed transition. It is also conceivable that one would like to monitor certain properties that are observable while the system resides in a particular state. A formalism to express properties of a system's behaviour should not only be able to express statements about such observable features, but should also be able to relate them over time. For instance if an observation is allowed to occur, but only after certain other property was observed earlier.
Temporal logic is a popular framework to express such aspects of a system's behaviour. Several variants exist, depending on, for example, the view taken on the description of a system's behaviour.

2.5.1 Branching Time Temporal Logic and Linear Temporal Logic

One particular division among temporal logics is between the view of behaviour as a set of possible (linear) traces (linear time temporal logics) and considering the branching structure of an LTS an integral part of its behaviour (branching time temporal logics).
A distinction is often made between *open* and *closed* systems. An open system is a system whose behaviour is influenced by interaction with the outside world. In a closed system the exposed behaviour is determined entirely by the system itself

and is not influenced by the outside world. Although reactive systems are by their very nature open systems, formal models often include a model of the environment in which they are supposed to operate. Such a model of a reactive system together with its environment can be considered a closed system.
Temporal logics are interpreted over structures of observations (called Kripke structures[5]). In linear temporal logic (for instance, LTL[134]) a formula is interpreted over a sequence of observations. For a labelled transition system, this could be the sequence of observed labels along a path through the LTS. Branching time temporal logic formulas (such as CTL[72]) are interpreted over trees of observable events. At every point in the execution, all possible transitions are taken into account.
The observations used in temporal logic are usually boolean observations. Individual components of an observation are called *atomic propositions* and take a boolean value in every observation. Propositional connectives such as *not*, *and*, *or* and *implies* can typically be used to combine atomic propositions into larger propositional formulas. Several variants of operators can be added to relate propositions on observations over time.

2.5.2 Untimed Temporal Logic

In this section we will define (untimed) propositional linear temporal logic (LTL, [134]). This logic is interpreted over (linear) sequences of observations, called *state sequences*. A state sequence $\overline{\sigma}$ over a set *Prop* of atomic propositions, is an ω-word (see section 2.1.3 for notation) $\overline{\sigma} = \sigma_0\sigma_1\sigma_2\ldots \in \left(2^{Prop}\right)^{\omega}$ over the alphabet 2^{Prop} of *states*. We let p range over *Prop*. A state is an individual observation and the interpretation of a state σ as a subset of *Prop* is that the proposition p holds (is true) in state σ iff $p \in \sigma$.
The syntax of LTL (ranged over by φ, ψ) is defined by the following grammar.

$$\varphi ::= \text{true} \mid p \mid \neg\varphi' \mid \varphi_1 \vee \varphi_2 \mid \bigcirc \varphi' \mid \varphi_1 \mathsf{U} \varphi_2.$$

$Prop(\varphi)$ is used to refer to the set of atomic propositions occurring in formula φ. The semantics of LTL formulas is formally defined for a model (state sequence) $\overline{\sigma}$ and a formula φ by means of the satisfaction relation $\models$, as follows.

- $\overline{\sigma} \models \text{true}$ for every state sequence $\overline{\sigma}$;
- $\overline{\sigma} \models p$ iff $p \in \overline{\sigma}(0)$;
- $\overline{\sigma} \models \neg\varphi'$ iff not $\overline{\sigma} \models \varphi'$;
- $\overline{\sigma} \models \varphi_1 \vee \varphi_2$ iff $\overline{\sigma} \models \varphi_1$ or $\overline{\sigma} \models \varphi_2$;
- $\overline{\sigma} \models \bigcirc\varphi'$ iff $\overline{\sigma}^1 \models \varphi'$;
- $\overline{\sigma} \models \varphi_1 \mathsf{U} \varphi_2$ iff there is some $k \geq 0$, such that $\overline{\sigma}^k \models \varphi_2$ and for all $0 \leq m < k$, $\overline{\sigma}^m \models \varphi_1$.

[5] Kripke structures are themselves similar to labelled transition systems, although states are labelled rather than transitions.

$\overline{\sigma} \models \varphi$ denotes that the formula φ holds for the state sequence $\overline{\sigma}$. p evaluates to the value of the atomic proposition p in the first state. true, $\neg$, $\vee$ are traditional propositional connectives. The last two operators relate states over time. $\bigcirc\varphi$ states that the formula φ holds for the state sequence obtained by removing the first state from $\overline{\sigma}$. The operator is pronounced 'next'. $\varphi_1 \mathsf{U} \varphi_2$ holds if there is some moment n (now or in the future) at which φ_2 is true (holds for $\overline{\sigma}^n$) and for all moments before that φ_1 holds, i.e. for all $0 \leq k < n$, $\overline{\sigma}^k \models \varphi_1$.
Other operators can be defined in terms of these.

- false: false $\stackrel{\text{def}}{=} \neg$true;
- conjunction: $\varphi_1 \wedge \varphi_2 \stackrel{\text{def}}{=} \neg(\neg\varphi_1 \vee \neg\varphi_2)$;
- the 'eventually' operator: $\Diamond\varphi \stackrel{\text{def}}{=}$ true$\mathsf{U}\varphi$, stating that φ will at some point become true (sometimes written as $\mathsf{F}\varphi$);
- the 'always' operator, $\Box\varphi \stackrel{\text{def}}{=} \neg\Diamond\neg\varphi$, stating that φ is true at every moment (sometimes written as $\mathsf{G}\varphi$).
- the dual $\varphi_1 \mathsf{V} \varphi_2 \stackrel{\text{def}}{=} \neg(\neg\varphi_1 \mathsf{U} \neg\varphi_2)$ of the Until operator (called Unless or Release).

The *language* $\mathcal{L}(\varphi)$ associated with the LTL formula φ (w.r.t. a set *Prop* of propositions) is the set of all state sequences over *Prop* that satisfy φ,

$$\mathcal{L}(\varphi) = \left\{ \overline{\sigma} \in \left(2^{Prop}\right)^{\omega} \mid \overline{\sigma} \models \varphi \right\}.$$

The $\bigcirc$ operator allows the expression of properties that are sensitive to the number of states visited between two particular states. This is sometimes undesirable, since unobservable transitions and interleaving concurrency introduce repetitions of identical observations. This phenomenon is often referred to as stuttering [124]. If one leaves out the $\bigcirc$ operator, only properties can be expressed that are insensitive to (finite) stuttering. Moreover in the dense real-time extension of temporal logic, the $\bigcirc$ operator (in contrast with the other operators) does not have a natural interpretation.

2.5.1 EXAMPLE *A state sequence over the set* $\{p, q\}$ *of propositions is for instance*

$$\{p\}\{p\}\{p,q\}\varnothing\{q\}\ldots.$$

It satisfies the formula $p\mathsf{U}q$ *(*q *holds at the third instant, and* p *for the first and second), but not the formula* $\Box p$ *(*p *fails to hold at the fourth and fifth instant).*

When dealing with LTL formulas it is often convenient to have negations occurring only in front of atomic propositions. This is achieved by introducing duals for all operators (false for true, $\wedge$ for $\vee$ and V for U, $\bigcirc$ is its own dual) and removing the negation (except in front of atomic propositions). Such formulas are said to be in *positive normal form* or *negation normal form*.
The *syntax* of positive LTL is defined by the following grammar.

$$\varphi ::= \text{true} \mid \text{false} \mid p \mid \neg p \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi' \mid \varphi_1 \mathsf{U} \varphi_2 \mid \varphi_1 \mathsf{V} \varphi_2.$$

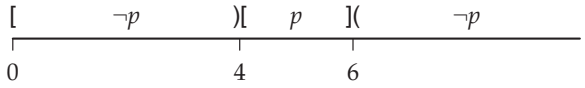


Figure 2.5: Example timed state sequence

2.5.3 Real-time Temporal Logic

Temporal logic can be used to express temporally related properties about the behaviour of a system. Such properties are only qualitative, it is not possible to express quantitative timing bounds on temporally related events. This is however crucial to express properties of real-time systems. The following section deals with the extension of linear temporal logic with quantitative timing constraints.

Real-time Models

Temporal logic formulas are interpreted over state sequences. For real-time temporal logic, the observations have to be extended with information about their timing. This is done by representing a sequence as a timed word over state observations, a timed state sequence.
A *timed state sequence* $\overline{\rho}$ over a set *Prop* of propositions is a timed word (see section 2.1.5) over the alphabet 2^{Prop}. Thus, it is a pair $(\overline{\sigma}, \overline{I})$ consisting of a state sequence $\overline{\sigma}$ and an interval sequence $\overline{I}$.
A timed state sequence describes the observable propositions during an execution including timing information. The propositions can only change a finite number of times in any finite time-interval due to the divergence property of the timed state sequence. A graphical representation of the timed state sequence $(\varnothing\{p\}\varnothing\ldots, [0,4)$ $[4,6](6\ldots)$ over $Prop = \{p\}$ is shown in figure 2.5.

Syntax and Semantics

We continue with the definition of the syntax and semantics of a particular real-time temporal logic, MITL([5]). The syntax of MITL is defined by the following grammar (where I is an interval).

$$\varphi ::= \text{true} \mid p \mid \neg\varphi' \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathsf{U}_I \varphi_2$$

Note that the $\bigcirc$ operator of LTL has no intuitive meaning in timed state sequences and has been removed. The Until operator is parameterised with an interval I, constraining the time of occurrence of φ_2 quantitatively to a moment in the interval.
Formally, the semantics of MITL formulas is defined as follows for a model (timed state sequence) $\overline{\rho}$.

- $\overline{\rho} \models \text{true}$ for every timed state sequence $\overline{\rho}$;
- $\overline{\rho} \models p$ iff $p \in \overline{\rho}(0)$;
- $\overline{\rho} \models \neg\varphi$ iff not $\overline{\rho} \models \varphi$;
- $\overline{\rho} \models \varphi_1 \vee \varphi_2$ iff $\overline{\rho} \models \varphi_1$ or $\overline{\rho} \models \varphi_2$;

- $\overline{\rho} \models \varphi_1 \mathsf{U}_I \varphi_2$ iff there is some $t \in I$, such that $\overline{\rho}^t \models \varphi_2$ and for all $0 \leq t' < t$, $\overline{\rho}^{t'} \models \varphi_1 \vee \varphi_2$.

The semantics of true, p, $\neg\varphi'$ and $\varphi_1 \vee \varphi_2$ are straightforward (note that $\overline{\rho}(0)$ denotes the observed state at time 0). The formula $\varphi_1 \mathsf{U}_I \varphi_2$ states that there is some instant in the interval I where φ_2 holds and up to the point where φ_2 holds, it is the case that φ_1 holds.
Note that in the semantics of $\varphi_1 \mathsf{U}_I \varphi_2$, we use $\overline{\rho}^{t'} \models \varphi_1 \vee \varphi_2$ in the latter part instead of $\overline{\rho}^{t'} \models \varphi_1$. This is used to avoid the anomaly that the formula is not true for the situation where φ_1 is true and φ_2 is false during a right-closed interval and φ_1 is false and φ_2 true in the next (and thus left-open) interval (see [9]). This solution however also has its drawback if the lower bound of I is greater than 0. This interpretation then allows that φ_1 temporarily does not hold before the lower bound of I is reached, as long as φ_2 holds in such periods. These problems do not occur in the logic we use in this thesis (we shall restrict the lower bound to 0).
Similar to the untimed case, other operators can be defined, indexed with an interval I, such as $\Diamond_I \varphi \stackrel{\text{def}}{=} \text{true} \mathsf{U}_I \varphi$ meaning there is some moment in I where φ holds and $\Box_I \varphi \stackrel{\text{def}}{=} \neg(\Diamond_I \neg\varphi)$ (φ holds at every moment in I).

2.5.2 EXAMPLE *The formula $\Diamond_{(5,10)} p$ holds for the timed state sequence of figure 2.5 (p holds for instance at $t = 6$). The formula $\Box_{[0,4]} \neg p$ does not.*

From the way timed state sequences are defined, it follows that on any finite interval of time, the valuation of atomic propositions can change only a finite number of times. It can be shown [5] that also the valuation of any MITL formula can change only a finite number of times on any finite interval, MITL is just one of many kinds of timed temporal logic, an overview can be found in [101].

2.6 Finite State Automata

Finite state automata are well known. The behaviour of reactive systems is viewed upon as a continuous interaction with the environment and is best captured as non-terminating behaviour. The classical finite state automaton that accepts finite words is adapted to accept infinite words. Such automata are called ω-automata ([43, 171]). These automata also move from state to state while accepting symbols of the input word. Since the input word never terminates, the notion of acceptance is modified. Several notions of acceptance exist. The most common one, called Büchi acceptance, states that a run of the automaton is accepting if it passes infinitely often through a state that is marked as accepting. Finite state automata are very similar to labelled transition systems. Although in automata often the edges are labelled instead of states, this difference is not significant. Acceptance conditions of automata can be compared to fairness constraints placed upon labelled transition systems.

2.6.1 ω-Automata

We define a fairly standard notion of ω-automaton. To avoid confusion with other entities, we call the states of the automata *locations* from now on. We label locations

rather than transitions and we label them with sets of symbols instead of with single symbols. An automaton having only single symbols in its locations can be easily obtained by splitting these locations into a single location for every symbol in the set. Similarly, it is easy to move the labelling to the transitions. Furthermore the acceptance conditions are slightly different and referred to in [89] as 'generalised Büchi' acceptance. This is a generalisation of Büchi acceptance, but equally expressive in the sense that an automaton with generalised Büchi acceptance can be converted into a normal Büchi automaton (at the cost of an increase in the number of locations by a factor of the number of acceptance sets) [51]. Formally, an ω-automaton is defined as follows.

2.6.1 DEFINITION *An* ω-automaton $A = (L, \Sigma, L_0, Q, E, F)$ *consists of*

- *a finite set* L *of locations;*
- *a finite alphabet* Σ*;*
- *a finite set* $L_0 \subseteq L$ *of initial locations;*
- *a mapping* $Q : L \to 2^\Sigma$ *labelling every location with a set of symbols from the alphabet;*
- *a set* $E \subseteq L \times L$ *of edges.* $(\ell, \ell') \in E$ *denotes an edge from location* ℓ *to location* ℓ'*;*
- *a set* $F \subseteq 2^L$ *of acceptance sets.* □

As an automaton accepts a word $\bar{w}$, it traverses the locations of the automaton while consuming symbols of $\bar{w}$. The sequence of locations that are visited is called a run. Next, we formally define what sequences of locations constitute a valid run and when a run matches a particular word.

2.6.2 DEFINITION *A* run $\overline{\ell} \in L^\omega$ *of an* ω*-automaton* $A = (L, \Sigma, L_0, Q, E, F)$ *is a sequence of locations, such that:*

- *(*CONSECUTION*) for all* $k \geq 0$*, there is an edge* $(\overline{\ell}(k), \overline{\ell}(k+1)) \in E$*;*
- *(*ACCEPTANCE*) for every set* $f \in F$*,* $\inf(\overline{\ell}) \cap f \neq \varnothing$ *(see section 2.1.3 for the definition of* $\inf(\cdot)$*).*

Given a word $\bar{w} \in \Sigma^\omega$*,* $\overline{\ell}$ *is a run for* $\bar{w}$ *on* A *if*

- *(*MATCHING*) for all* $k \geq 0$*,* $\bar{w}(k) \in Q(\overline{\ell}(k))$*.* □

A run $\overline{\ell}$ of A is called initial if $\overline{\ell}(0) \in L_0$. One says that A *accepts* the word $\bar{w}$ if there is an initial run $\overline{\ell}$ of A for $\bar{w}$. The following holds for every run $\overline{\ell}$.

2.6.3 LEMMA *if* $\overline{\ell}$ *is a run for* $\bar{w}$ *of* ω*-automaton* A*, then for any* $k \geq 0$*,* $\overline{\ell}^k$ *is a run for* $\bar{w}^k$ *of* A*.*

The *language* $\mathcal{L}(A)$ associated with the automaton A is the set of all ω-words that are accepted by A.

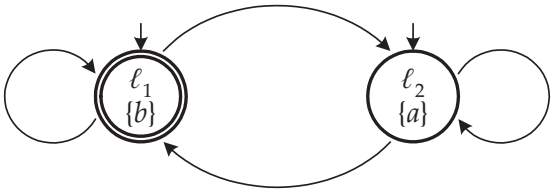


Figure 2.6: Example of a Büchi automaton

Drawing Conventions When drawing ω-automata, locations of an automaton are indicated by circles possibly containing a name or number for reference. Edges are represented by arrows between locations. The initial locations are indicated by a small arrow leading to the location, not starting from any location. The accepting locations are indicated by an extra line around it if there is only a single acceptance set. In the cases we need to visualise multiple acceptance sets, we use other means to visually discriminate locations in an ad hoc fashion. The labelling of locations with symbols is also written inside of the locations.

2.6.4 EXAMPLE *Figure 2.6 shows an example of a Büchi automaton that accepts all infinite words of $\{a,b\}^\omega$ that contain an infinite number of b's. Because of the acceptance condition, any run on the automaton must visit infinitely often, the location ℓ_1. Thus, any matching word has infinitely many b's.*

An automaton is called *deterministic*, if for every symbol $\sigma \in \Sigma$, there is at most one initial location l that matches it and for every location $\ell \in L$ and symbol $\sigma \in \Sigma$, there is at most one edge $(\ell, \ell') \in E$ such that ℓ' matches σ. The analysis of systems using finite state automata often involves operations on automata (see chapter 6) such as determination, the construction of a deterministic automaton accepting the same language, and complementation, the construction of an automaton accepting those words the original automaton does not accept. Determination of Büchi automata is not possible (ω-automata with certain other acceptance conditions can be made deterministic). Complementation is possible, but at the cost of an exponential blowup in size. Checking Büchi automata for emptiness of the associated language can be done in linear time [171].

2.6.2 Timed Automata

ω-Automata accept ω-words over some alphabet Σ. Such words do not contain information about time, other than the discrete order of occurrence of symbols. In section 2.1.5, we have seen the extension of ω-words to timed words. Timed automata ([7]) are finite state automata that accept timed words. To do this, the automata exploit timers that can be set to a particular value and whose values decrease as time passes. Conditions on timers can be used to enforce timing constraints.
We simplify and adapt our timed automata a little (from the standard automata of [7], in fact, from the interval automata of [5]) to suit the use in this thesis (in particular

in chapter 8)[6]. We start with some definitions. A *timer valuation (timer assignment)* ν over a set Θ of timers is a (total) mapping $\Theta \to \mathbb{R}$. We call the set of all timer valuations over Θ, $TVal(\Theta)$. For any $t \in \mathbb{R}$, we let $\nu - t$ denote the timer valuation that assigns the value $\nu(x) - t$ to any timer x in the domain Θ of ν. Note that timers may become negative. $\mathbf{0}^{\Theta}$ denotes the timer valuation that maps all timers in Θ to the value 0.
The values of particular timers can be changed by means of a timer setting operation. A *timer setting TS* over a set Θ of timers is a (partial) mapping $\Theta \hookrightarrow \mathbb{N}$. The set of timer settings over Θ is denoted by $TSet(\Theta)$. For any $\nu \in TVal(\Theta)$ and $TS \in TSet(\Theta)$, with $TS[\nu]$ we will refer to the timer valuation that maps a timer $x \in \Theta$ to $TS(x)$ if defined and to $\nu(x)$ otherwise.
A *timer condition* on a timer x is an expression of one the following forms $x > 0$, $x \geq 0$, $x < 0$ or $x \leq 0$. The collection of timer conditions on a set Θ of timers, $\{x > 0, x \geq 0, x < 0, x \leq 0 \mid x \in \Theta\}$ is called $TCond(\Theta)$. The *satisfaction of a timer condition* in $TCond(\Theta)$ by a timer valuation $\nu \in TVal(\Theta)$ is (straightforwardly) defined as follows.

- $\nu \models x > 0$ iff $\nu(x) > 0$;
- $\nu \models x < 0$ iff $\nu(x) < 0$;
- $\nu \models x \geq 0$ iff $\nu(x) \geq 0$;
- $\nu \models x \leq 0$ iff $\nu(x) \leq 0$.

The state of a timed automaton during the acceptance of a word now consists not only of the location it currently resides in, but also of the values of its timers. Such a state is called an *extended location* [8]. An *extended location s* of a timed automaton with locations L and timers Θ, is a pair (ℓ, ν) consisting of a location $\ell \in L$ and a timer valuation $\nu \in TVal(\Theta)$.

2.6.5 DEFINITION *A* timed automaton $A = (L, \Sigma, \Theta, L_0, Q, TC, E)$ *consists of*

- *a finite set L of locations;*
- *a finite alphabet Σ;*
- *a set Θ of timers;*
- *a finite set L_0 of initial extended locations $(\ell_0, \nu_0) \in L \times TVal(\Theta)$, where the timers are assigned by ν_0 integer values only;*
- *a mapping $Q : L \to 2^{\Sigma}$ labelling every location with a set of symbols from the alphabet;*
- *a mapping $TC : L \to 2^{TCond(\Theta)}$, labelling every location with a set of timer constraints over timers in Θ;*
- *a set $E \subseteq L \times TSet(\Theta) \times L$ of edges. $(\ell, TS, \ell') \in E$ denotes an edge from location ℓ to location ℓ', labelled with a timer set operation TS.* □

[6] Without proof, we conjecture that this restriction does not decrease the expressiveness of the automata other than the lack of acceptance conditions, i.e. our automata are equally expressive as interval automata of [5] without acceptance conditions. The main restriction lies in the fact that we can only compare timers to 0, though we can set it to an arbitrary (integer) value. The effect of comparing a single clock with different bounds can be mimicked by setting different timers to the corresponding initial values simultaneously and comparing the right timer with 0.

As a run describes the subsequent states of the automaton while accepting the input word, a run as used for ω-automata has to be extended with information about the timer values. It should describe at any moment what the location of the automaton is and what its timer values are. This can be fully described by the sequences of locations, the intervals during which the automaton resides in each location, and the timer values at the beginning of each such interval.

2.6.6 DEFINITION *A* (timed) run $\bar{r}$ *of a timed automaton* $A = (L, \Sigma, \Theta, L_0, Q, TC, E)$ *is a triple* $(\overline{\ell}, \overline{I}, \overline{\nu})$ *consisting of a sequence of locations, an interval sequence, and a sequence of timer valuations, such that:*

- *(*CONSECUTION*) for all* $k \geq 0$*, there is an edge* $(\overline{\ell}(k), TS_k, \overline{\ell}(k+1)) \in E$ *such that* $\overline{\nu}(k+1) = TS_k[\overline{\nu}(k) - |\overline{I}(k)|]$*;*
- *(*TIMING*) for all* $k \geq 0$*,* $t \in \overline{I}(k)$ *and* $\chi \in TC(\overline{\ell}(k))$*, we have* $\overline{\nu}(k) - (t - l(\overline{I}(k))) \models \chi$*.*

We write $\bar{r}(t)$ *to denote* $\overline{\ell}(k)$ *if* $t \in \overline{I}(k)$*. A run of A is initial if* $(\bar{r}(0), \overline{\nu}(0)) \in L_0$*. Given a timed word* $\bar{\upsilon}$*,* $\bar{r}$ *is a run for* $\bar{\upsilon}$ *of A if*

- *(*MATCHING*) for all* $t \geq 0$*,* $\bar{\upsilon}(t) \in Q(\bar{r}(t))$*.* □

The requirement on the timer valuation ν_{k+1} reflects the operational interpretation of a run. If ν_k captures the timer values at the beginning of the k'th interval, then $\nu_k - |\overline{I}(k)|$ represents their values at the end of the interval (if the interval is right-open, then this time valuation is not actually reached). After the timer setting TS_k, we obtain the timer values at the beginning of the next interval. Similarly, $\overline{\nu}(k) - (t - l(\overline{I}(k)))$ represents the timer values at instant t.
The timed automaton A accepts the timed word $\bar{\upsilon}$ iff there is an initial run $\bar{r}$ of A for $\bar{\upsilon}$. Note that in our timed automata, timers decrease rather than increase and they are set to a positive value instead of reset to 0 as in [7]. Timers are only compared to zero. We have furthermore left out the acceptance conditions for simplicity. They can be added in the standard way. It follows from 2.6.6 that timed automata cannot discriminate between equivalent timed words.

2.6.7 DEFINITION *The* suffixes *of sequences of timer valuations and timed runs are defined as follows.*

- *For any* $k \in \mathbb{N}$ *and* $d \in \mathbb{R}^{\geq 0}$*, the suffix* $\overline{\nu}^{k,d}$ *of a sequence* $\overline{\nu} = \nu_0\nu_1\nu_2\ldots$ *of timer valuations is defined as the sequence* $(\nu_k - d)\,\nu_{k+1}\nu_{k+2}\ldots$*.*
- *For a timed run* $\bar{r} = (\overline{\ell}, \overline{I}, \overline{\nu})$ *and any* $t \in \mathbb{R}^{\geq 0}$*, the suffix* $\bar{r}^t$ *is defined as* $(\overline{\ell}^k, \overline{I}^t, \overline{\nu}^{k,d})$ *where* k *is such that* $t \in \overline{I}(k)$ *and* $d = t - l(\overline{I}(k))$*.* □

If $\bar{r}$ is a run for the timed state sequence $\overline{\rho}$ on A, then for every $t \geq 0$, $\bar{r}^t$ is a run for $\overline{\rho}^t$ on A. The *language* $\mathcal{L}(A)$ associated with the timed automaton A is the set of all timed words that are accepted by A.

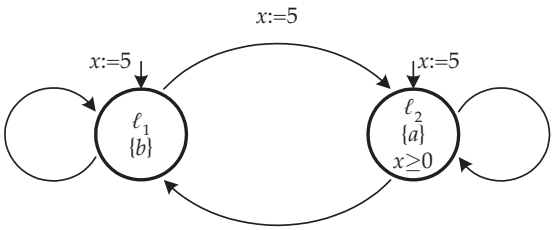


Figure 2.7: Example of a timed automaton

Drawing Conventions The locations, edges and symbol labelling are visualised as it is done for ω-automata. Initial extended locations are represented by small arrows with assignments to the timers of the automaton. The timer conditions are written inside locations and timer settings associated with edges are written in the form of assignments along the arrow representing the edge. Our timed automata will not have acceptance conditions.

2.6.8 EXAMPLE *Figure 2.7 shows a timed automaton that accepts timed words over $\{a, b\}$ such that a symbol a is never observed for more than 5 units of time uninterruptedly. The use of timer x guarantees that a run of the automaton does not reside for more than 5 units of time in location ℓ_2 without visiting ℓ_1 in between. x is set to 5 upon entering ℓ_2 and is required to remain at least 0 for as long as the run remains in ℓ_2. Thus a matching run cannot show the symbol a for more than 5 units of time without showing symbol b in between.*

As with untimed automata, the use of timed automata for analysing systems involves checking some of its properties. Checking the emptiness of the language it accepts is very important (see chapter 6) and is PSPACE complete. Universality of the language, language inclusion and language equivalence between automata are undecidable and timed automata are not closed under complement [8].

2.7 Automatic Protection Switching Protocol

As a practical case study for the techniques described in this thesis, we have modelled and analysed a protocol designed to provide automatic protection against failure of communication lines between high-level telecommunication nodes. The project was performed in cooperation with Lucent Technologies in Huizen, the Netherlands, part of Bell Laboratories [132]. The protocol is named APS (Automatic Protection Switching). SDH (Synchronous Digital Hierarchy), is a protocol used to transport large quantities of traffic between high-level switches in a telecommunication network through optical fibres. APS is used to protect linear (point to point) SDH network sections against the failure or degradation of one of the communication lines by automatically rerouting the traffic onto an extra, redundant line (see figure 2.8). This happens automatically, without the intervention of the network management system, governed by APS. As there are two sides of the communication line, a

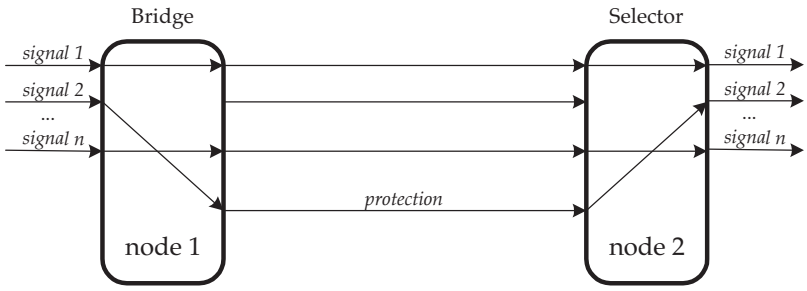


Figure 2.8: Rerouting through protection line

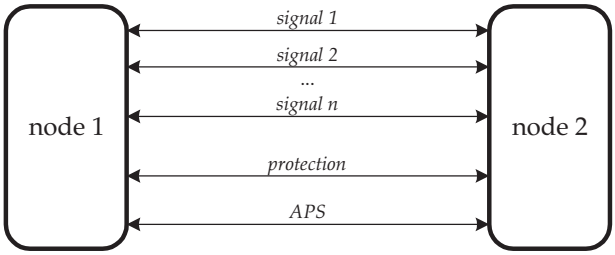


Figure 2.9: APS Nodes

protocol is necessary to coordinate the use of the protection line. The APS protocol is specified in ITU-T Recommendation G.841[110] and (at that time) a formalisation of the protocol was under study within the context of ETSI specification ETS 300 417 [75][7].

On one side of the connection, there is a bridge that determines which signal is placed on the protection line and on the other side there is a selector that connects the redundant line to the appropriate channel if necessary. The two nodes exchange messages via a (reliable) APS channel, informing each other about their status (see figure 2.9). The decision what channel to be switched onto the protection line is based upon information received about the quality of the lines (signals may be normal, degraded, or failing completely). Moreover external configuration commands can be given by the network management. The highest priority request is granted (signal failure has priority over signal degradation, signals with a lower number have priority over signal with a higher number). Network management commands have even higher priority and can be used (among other things) to force a switch of a particular signal to the protection channel or to forbid a particular signal to be switched to the protection channel.

Nodes on both sides generate requests based upon this information and communicate the request to the other node. The node with the highest request keeps transmitting the request and the other node starts transmitting an acknowledgement that

[7]Recommendation G.841 is a description of the protocol in plain english text. The ETSI specification is an attempt to make this more precise by giving a reference model in terms of a structure of communicating processes and pseudo-code; not a model with a mathematically defined semantics.

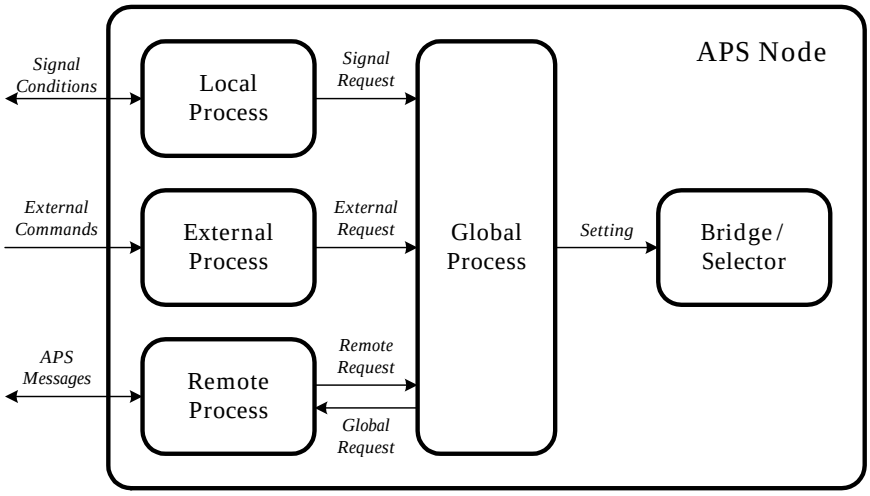


Figure 2.10: Structure of an APS node

it operated in accordance with the request of the opposite node.

There are several modes and architectures in which the APS protocol may operate. The protocol can be unidirectional or bidirectional, there may be one protection line for one signal line or one protection line for several signal lines. The protocol may automatically revert to the original situation after a signal condition returns to normal or it may leave the signal on the protection line until some other signal requires it. The protection line, if unused for protection, may be used to carry extra traffic.

A node is specified as a number of concurrent processes whose behaviour is given in pseudo-code in the ETSI specification. The important processes in the specification are the local process, the remote process, the external process, the global process and bridge and selector control (see figure 2.10). The local process determines the local request with highest priority. The remote process monitors the APS messages of the other node to determine the request and bridge / selector settings of the other node. The external process takes commands from the network management and turns them into corresponding requests. The global request combines local, remote and external request to determine the appropriate local settings of bridge / selector and request to send back to the opposite node.

The following scenario describes the possible behaviour. Assume that at the beginning, no signal is switched to the protection line and signal conditions are normal. Both nodes issue the lowest priority request NR, indicating that the situation is normal. Now assume node 1 detects that signal 1 is failing. Then it issues a request SF-1 (Signal Failure on line 1). This is detected by the opposite node, which issues a request RR (Reverse Request) indicating that it acknowledges that the request of the other node is of higher priority than its own (still NR) and sets its selector accordingly. Some time later, the signal quality may return to normal and the request SF-1 returns to NR. The numbers behind the request in the scenario indicate the local bridge and selector setting.

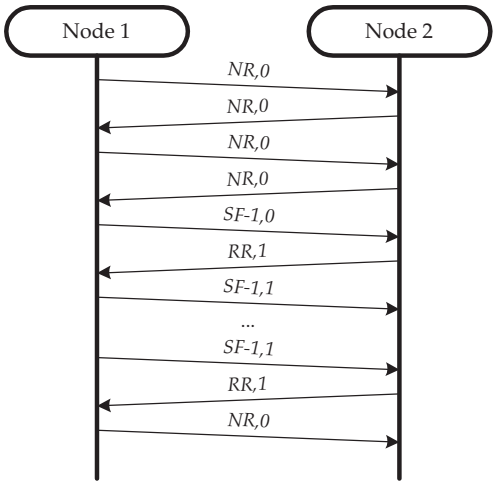


Figure 2.11: Scenario of APS messages

The model presented in this thesis is a simplified model of the protocol and only supports the following requests (in order of priority, low to high): RR, NR, SD-n (degradation of signal n), SF-n (failure of signal n), FSw-n (forced switch of signal n to the protection line). Only the following commands from the network management to the protocol are modelled: FSw-n, CLR (clear current command), LOP-n (lockout of protection of signal n, deny switch of signal n to the protection line) CLRLOP-n (clear the lockout of signal n).

Chapter 3

A Calculus for Real-Time Concurrent Systems

In this chapter a formal model for real-time concurrent systems is introduced in the form of a process calculus. Although many of the techniques that are introduced are aimed at (and have been applied to, [87, 156]) complex systems, including communication and concurrency, real-time and complex data, the calculus we present here will abstract from data; it would be impossible and unnecessary for the purposes of this thesis to introduce a calculus of this complexity. Since we focus our attention on static structures of collaborating entities, the calculus describes a fixed structure of interconnected components. The semantics of the calculus is given in the form of a structured operational semantics, including a notion of time. Moreover, the semantics is cast in a more direct, operational form that is better suited for the implementation described in the next chapter.

This work started in the context of the formal modelling and specification language POOSL ([156]). The basic concepts of POOSL for communication and concurrency are inspired by CCS. The CCS-like calculus introduced here is in fact an abstract version of the language POOSL. The results obtained in this thesis have been applied to POOSL itself and are applicable to other languages or calculi in this style as well. The focus on static structure and architecture originates also from POOSL and the accompanying analysis and design framework SHE([156]).

Section 3.1 introduces the syntax of the calculus and section 3.2 its semantics. We focus the notion of termination in the semantics in section 3.3. A small example illustrates the calculus in section 3.4. Since the calculus does not include data, but POOSL does, we pay some attention to an operational framework for data semantics in section 3.5. A model of the automatic protection switching protocol in the calculus is presented in section 3.6. Section 3.7 gives an example of a part of the actual POOSL model of the APS protocol and discusses its relation with the presented calculus. Sections 3.8 and 3.9 end the chapter with related work and conclusions respectively.

3.1 Syntax of the Calculus

A process of the calculus will describe a static structure of interacting components. The syntax of process expressions is split into two parts, the dynamic processes and the static composition of such dynamic processes. In [140], Milner distinguishes between static and dynamic combinators. The static combinators are the ones that remain in the process expression after a transition. The dynamic combinators may disappear after the transition. In CCS, static and dynamic operators can be mingled freely. We separate static and dynamic combinators into two layers (similar to the 'networks of regular processes' in [129] and the 'simple agents' of [149]). This way, the static operators are truly static in the sense that they cannot be removed because they occur in a subprocess of a dynamic operator.

3.1.1 Dynamic Processes

The set $\mathcal{DP}$ of dynamic processes, ranged over by S, is the set produced by the following grammar, ($\alpha \in Act$ and $t \in T^+$):

$$S ::= \alpha \quad | \quad \langle t \rangle \quad | \quad S_1; S_2 \quad | \quad S_1 + S_2 \quad | \quad S_1 \mid S_2 \quad | \quad A$$

together with the process $\surd$.
α is the process which can perform the action α after which it terminates (sometimes it is written as (α) to be able to distinguish between the *action* α and the *process* (α)). $\langle t \rangle$ is the process that can delay for an amount t of time and after that terminate. It cannot perform any actions. $S_1; S_2$ is the sequential composition of S_1 and S_2. Its behaviour is similar to that of S_1 until it terminates. After that its behaviour is that of S_2. The choice process $S_1 + S_2$ behaves as either S_1 or S_2 depending on a (non-deterministic) choice, made at the time at which either of the processes performs some action. $S_1 \mid S_2$ represents the concurrent execution of S_1 and S_2. Concurrency is modelled as the arbitrary interleaving of their respective actions. It is a pure interleaving, i.e. the processes do not synchronise. This is in contrast with the parallel composition operator that is introduced in the following section. A is a process constant. The process constants are defined by a set of (recursive) definitions $\{A \stackrel{\text{def}}{=} S_A \mid A \in \mathcal{KD}\}$, where $S_A \neq \surd$ and $\mathcal{KD}$ is a set of process constants. It is furthermore assumed that all recursion in the constant definitions is guarded, i.e. if a constant A is directly or indirectly defined in terms of itself, then there must be an action α or delay $\langle t \rangle$ in between; the definition $A \stackrel{\text{def}}{=} A + a$ for instance, is not allowed[1].
The $\surd$ process is used as an aid for representing a terminated process. The only (dynamic) process expression in which it occurs is the terminated process $\surd$ itself. It cannot engage in any action, but can let arbitrary amounts of time pass.

3.1.1 EXAMPLE *We give a definition of a variant of Milner's jobshop ([140]). The dynamic components of the jobshop (making use of sequential composition rather*

[1] When interpreted as equations, unguarded definitions may have multiple solutions. Moreover, together with sequential composition and the use of data expression to parameterise a process constant, this leads to processes that are very hard to implement.

than CCS's prefix operator) are the following constants.

$$
\begin{array}{rcl}
\mathit{Hammer} & \stackrel{\text{def}}{=} & \mathrm{geth};\ \mathrm{puth};\ \mathit{Hammer} \\
\mathit{Mallet} & \stackrel{\text{def}}{=} & \mathrm{getm};\ \mathrm{putm};\ \mathit{Mallet} \\
\mathit{UseHammer} & \stackrel{\text{def}}{=} & \overline{\mathrm{geth}};\ \langle 2\rangle;\ \overline{\mathrm{puth}} \\
\mathit{UseMallet} & \stackrel{\text{def}}{=} & \overline{\mathrm{getm}};\ \langle 1\rangle;\ \overline{\mathrm{putm}} \\
\mathit{UseTool} & \stackrel{\text{def}}{=} & \mathit{UseHammer} + \mathit{UseMallet} \\
\mathit{Jobber} & \stackrel{\text{def}}{=} & \left(\begin{array}{l} \mathrm{inEasy};\ \overline{\mathrm{outEasy}} \\ + \\ \mathrm{inNormal};\ \mathit{UseTool};\ \overline{\mathrm{outNormal}} \\ + \\ \mathrm{inHard};\ \mathit{UseHammer};\ \overline{\mathrm{outHard}} \end{array}\right);\ \mathit{Jobber}
\end{array}
$$

The Hammer and Mallet processes model the behaviour of tools that are used in the shop. The geth *and* getm *action represent the event that a tool is picked up by someone to be used. Before it can be used again, it must be put back, represented by the* puth *and* putm *actions. The process constants UseHammer, UseMallet and UseTool are used to define the behaviour of a jobber. UseHammer models the use of the hammer (picking it up, using it for 2 units of time and putting it back). Similarly, UseMallet models the use of the mallet and UseTool models the use of either of the tools. Finally, Jobber models the behaviour of a jobber. It may take any of three kinds of jobs, easy ones, normal ones or hard ones. Easy jobs can be done without a tool, normal jobs with any of the tools and hard jobs only with the hammer. The finishing of a job is represented by the* $\overline{\mathrm{outEasy}}$, $\overline{\mathrm{outNormal}}$ *and* $\overline{\mathrm{outHard}}$ *actions.*

3.1.2 Static Structure of Processes

The static part of the syntax describes the static structure in which a collection of dynamic processes are connected. This structure remains intact as the processes execute. Static processes are not created dynamically and the structure does not change. The syntax defining the set $\mathcal{SP}$ of static processes (ranged over by R) is as follows. The static operators are identical to the corresponding operators of CCS [140].

$$R \quad ::= \quad S \quad | \quad R_1||R_2 \quad | \quad R[f] \quad | \quad R\backslash L \quad | \quad AS$$

A static process can be the instantiation of a dynamic process S. Static processes can be composed in parallel as $R_1||R_2$. This operator lets R_1 and R_2 operate interleaved concurrent and lets the processes synchronise on complementary actions. $R[f]$ is the process R, but with its actions relabelled according to the relabelling function f. The hiding operator restricts the actions of a process, $R\backslash L$ behaves as R, but the hiding operator prevents actions in L from occurring. Furthermore, there is a set $\mathcal{KS}$ of static process constants AS, defined by a set of *non-recursive* definitions $\{AS \stackrel{\text{def}}{=} R_{AS} \mid AS \in \mathcal{KS}\}$, where $\mathcal{KS}$ is a set of process constants. This guarantees that any static process has a finite static structure built from dynamic processes.

3.1.2 EXAMPLE *The jobshop can be described by the static process expression*

$$\mathit{Jobshop} \stackrel{\text{def}}{=} (\mathit{Jobber}||\mathit{Jobber}||\mathit{Hammer}||\mathit{Mallet}) \setminus \{\mathrm{geth}, \mathrm{puth}, \mathrm{getm}, \mathrm{putm}\}.$$

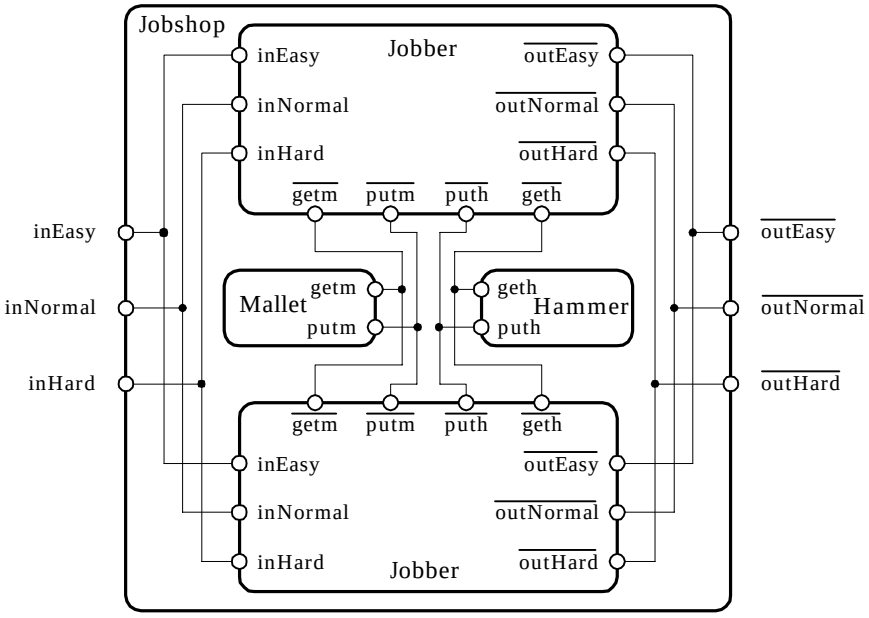


Figure 3.1: Example of a statically structured system: Milner's jobshop

The structure of this static process is graphically illustrated in figure 3.1. A component's sort of actions is indicated by 'ports' at its border. Lines between the ports ('channels') indicate the possible synchronisations between components; actions that have not been restricted are available for synchronisation at the ports of the jobshop itself. The geth, puth, getm *and* putm *actions are restricted to prevent entities from outside the jobshop to use the tools. Due to the static nature of the static processes, this picture is appropriate for any derivative expression of Jobshop.*

3.2 Semantics of the Calculus

This section defines the semantics of processes in terms of a timed labelled transition system. The labelled transition system will have transitions with two different kinds of labels. On the one hand there are action transitions, describing the change in a process which is the result of performing some action, and on the other hand there are time transitions describing the effect the passage of time has on a process.
The semantics of the calculus is given as the quintuple

$$StatCalc := \left(\mathcal{SP}, Act, T, \xrightarrow[Act]{}, \xrightarrow[T^+]{}\right)$$

consisting of the set of static processes $\mathcal{SP}$ that will serve as the states of the labelled transition system; the set of actions Act, the time domain T and two labelled transition relations: $\xrightarrow[Act]{} \subseteq \mathcal{SP} \times Act \times \mathcal{SP}$, a set of action transitions labelled with the actions of Act and a time-labelled transition relation $\xrightarrow[T^+]{} \subseteq \mathcal{SP} \times T^+ \times \mathcal{SP}$.

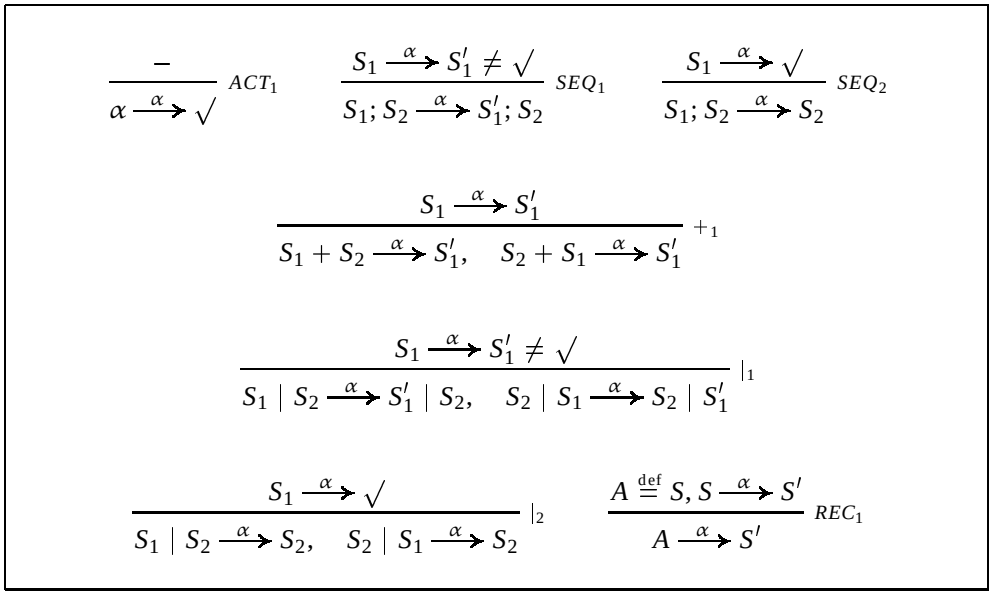


Table 3.1: Semantics of the action transitions of dynamic processes

The semantical axioms and rules that define the action transition relations of dynamic processes are given in table 3.1. Each rule is given a name for reference that is written beside it. The axiom ACT_1 states that the *process* α can perform the *action* α and after that terminate (α cannot perform any other actions). Sequential composition is defined by rules SEQ_1 and SEQ_2 stating that the possible actions of the composite are 'inherited' from the possible actions of the first process, corresponding to the idea that the sequential composition starts with executing the first part. In case the result of the action is the termination of the first process (SEQ_2), the subsequent behaviour of the composite is identical to the behaviour of the second process. If S_1 did not terminate ($S_1 \neq \surd$, SEQ_1), its execution is continued. The choice operator $+$ can perform any action that can be performed by one of its subprocesses. Once such an action has been performed, a choice has been made and the subprocess from which the action originates is further executed. This is expressed by rule $+_1$. The interleaving of two processes (rules $|_1$ and $|_2$) can also perform any action that can be performed by one of its subprocesses. In contrast with the choice operator however, both subprocesses remain active and continue to interleave their actions. Upon termination of one of the processes ($|_2$), the terminating process is removed and the other process continues its execution. Finally the rule REC_1 claims that the agent A defined as R can perform the same actions as R and has the same remaining behaviour.

The rules for the time transition relations of dynamic processes are shown in table 3.2. The axiom ACT_2 says that synchronisation actions (all actions other than τ) are allowed to let an arbitrary amount of time pass, waiting for a synchronisation to become available. If an internal action τ can be performed, one would like it to be performed immediately. The delay rules DEL_1 and DEL_2 describe the time behaviour of the delay process. Note that the delay process $\langle t \rangle$ is not mentioned in the

$$\frac{-}{\ell \xrightarrow{t} \ell}\ ACT_2 \qquad \frac{0 < t' < t}{\langle t \rangle \xrightarrow{t'} \langle t - t' \rangle}\ DEL_1 \qquad \frac{-}{\langle t \rangle \xrightarrow{t} \surd}\ DEL_2$$

$$\frac{S_1 \xrightarrow{t} S_1' \neq \surd}{S_1; S_2 \xrightarrow{t} S_1'; S_2}\ SEQ_3 \qquad \frac{S_1 \xrightarrow{t} \surd}{S_1; S_2 \xrightarrow{t} S_2}\ SEQ_4$$

$$\frac{S_1 \xrightarrow{t} S_1' \neq \surd,\ S_2 \xrightarrow{t} S_2' \neq \surd}{S_1 + S_2 \xrightarrow{t} S_1' + S_2'}\ +_2 \qquad \frac{S_1 \xrightarrow{t} \surd,\ S_2 \xrightarrow{t} S_2'}{S_1 + S_2 \xrightarrow{t} \surd, \quad S_2 + S_1 \xrightarrow{t} \surd}\ +_3$$

$$\frac{S_1 \xrightarrow{t} S_1' \neq \surd,\ S_2 \xrightarrow{t} S_2' \neq \surd}{S_1 \mid S_2 \xrightarrow{t} S_1' \mid S_2'}\ |_3 \qquad \frac{S_1 \xrightarrow{t} \surd,\ S_2 \xrightarrow{t} S_2'}{S_1 \mid S_2 \xrightarrow{t} S_2', \quad S_2 \mid S_1 \xrightarrow{t} S_2'}\ |_4$$

$$\frac{A \stackrel{\text{def}}{=} S,\ S \xrightarrow{t} S'}{A \xrightarrow{t} S'}\ REC_2 \qquad \frac{S \xrightarrow{t_1} S',\ S' \xrightarrow{t_2} S''}{S \xrightarrow{t_1+t_2} S''}\ CNT$$

Table 3.2: Semantics of the time transitions of dynamic processes

action rules and thus it cannot perform any actions. $\langle t \rangle$ can make time transitions up to t. The resulting process corresponds to the remaining delay (DEL_1) or terminates after precisely t units of time (DEL_2). The sequential composition rules SEQ_3 and SEQ_4 are identical to rules for the action transitions and state that the composite inherits its behaviour from the first component. The choice operator can let time pass only if both of the subprocesses can[2]. Termination of one of the subprocesses (this must have been without ever performing an action) makes the choice terminate[3]. The interleaving rules ($|_3$ and $|_4$) state that time passes synchronously in concurrent processes. Rule $|_3$ applies if neither of the subprocesses terminates after the delay; otherwise, rule $|_4$ applies and the terminated subprocess is removed from the interleaving. When the subprocesses terminate together, both conclusions of $|_4$ apply and state that the interleaving process terminates. The rule REC_2 is again the same as the corresponding rule for the action transitions. The final rule *CNT* makes (together with *ACT* and DEL_1) that the time transitions are closed in the sense that time transitions can be arbitrarily combined or refined into larger or smaller time steps[4].

The remaining part of the semantics, the action and time rules for the static processes, are given in table 3.3. The axiom *TRM* states that once a local process has terminated, it can delay indefinitely. To be able to combine time and action transitions in the rule *CMP*, we introduce γ as a typical element of $Act \cup T^+$. In the parallel composition, the subprocesses interleave their actions or synchronise on complementary actions as described by rules PAR_1 and PAR_2 respectively. Time passes synchronously in the static parallel processes (PAR_3, the urgency condition will be discussed in section 3.2.1). The rules for the relabelling and hiding operators are straightforward. One can derive for example using the rules ACT_1 and SEQ_2 that $a;b \xrightarrow{a} b$, as demonstrated by the following derivation tree:

$$\dfrac{\dfrac{}{a \xrightarrow{a} \surd}\,ACT_1}{a;b \xrightarrow{a} b}\,SEQ_2$$

From rules ACT_2, DEL_1, SEQ_4, $+_3$ and $|_4$ it follows that $a \mid (b + \langle 2 \rangle) \xrightarrow{2} a$ (assuming that $2 \in T$), as demonstrated by the following derivation:

$$\dfrac{\dfrac{}{a \xrightarrow{2} a}\,ACT_2 \qquad \dfrac{\dfrac{}{b \xrightarrow{2} b}\,ACT_2 \qquad \dfrac{}{\langle 2 \rangle \xrightarrow{2} \surd}\,DEL_2}{b + \langle 2 \rangle \xrightarrow{2} \surd}\,+_3}{a \mid (b + \langle 2 \rangle) \xrightarrow{2} a}\,|_4$$

[2]Thus it is a strong choice operator [147], like the one in TCCS [186], as opposed to the weak choice of ACP_ρ [21], where the inability to idle of one of the constituents may lead to discarding of that alternative.

[3]An alternative would have been to allow the other constituent to continue its behaviour. This interpretation was chosen because it enabled the expression of time-out and watchdog behaviour in combination with other primitives in the language POOSL.

[4]Time-closure could have been achieved by introducing a rule particularly for the sequential composition as it is the only operator that does not preserve time-closure by its standard rules.

$$\frac{-}{\surd \xrightarrow{t} \surd}\ TRM \qquad \frac{R_1 \xrightarrow{\alpha} R_1'}{R_1||R_2 \xrightarrow{\alpha} R_1'||R_2,\quad R_2||R_1 \xrightarrow{\alpha} R_2||R_1'}\ PAR_1$$

$$\frac{R_1 \xrightarrow{\ell} R_1', R_2 \xrightarrow{\overline{\ell}} R_2'}{R_1||R_2 \xrightarrow{\tau} R_1'||R_2'}\ PAR_2$$

$$\frac{R_1 \xrightarrow{t} R_1', R_2 \xrightarrow{t} R_2', \neg Urgent^{\{\tau\}}(R_1||R_2, t)}{R_1||R_2 \xrightarrow{t} R_1'||R_2'}\ PAR_3 \qquad \frac{R \xrightarrow{\alpha} R'}{R[f] \xrightarrow{f(\alpha)} R'[f]}\ REL_1$$

$$\frac{R \xrightarrow{t} R'}{R[f] \xrightarrow{t} R'[f]}\ REL_2 \qquad \frac{R \xrightarrow{\alpha} R', \alpha \notin L, \overline{\alpha} \notin L}{R\backslash L \xrightarrow{\alpha} R'\backslash L}\ HID_1 \qquad \frac{R \xrightarrow{t} R'}{R\backslash L \xrightarrow{t} R'\backslash L}\ HID_2$$

$$\frac{AS \stackrel{\text{def}}{=} R, R \xrightarrow{\gamma} R'}{AS \xrightarrow{\gamma} R'}\ CMP$$

Table 3.3: Semantics of the static processes

3.2.1 Maximal Progress

The rules of the semantics do not allow a process that can perform a τ action to delay. It is desirable to force processes to advance as soon as possible, to guarantee maximal progress ([186, 147]). The semantical rules do not allow the process τ to delay and thus neither any process containing τ as a subprocess that is ready to execute. However, an internal action in the form of a synchronisation may be available (or become available after a time transition). The individual synchronising actions are allowed to delay arbitrary amounts of time waiting for a synchronisation partner to become available. It is the rule PAR_3 that allows concurrent entities to wait synchronously, but only if and as long as they cannot synchronise. Note that time transitions in the calculus are deterministic and therefore if $R \xrightarrow{t} R'$ then R' is uniquely determined by R and t (this can be proved by structural induction on R). Thus it is completely determined by R and t, whether or not R will be urgent after time t.
To achieve maximal progress in the calculus, the rule PAR_3 has an urgency condition similar to corresponding rules of the timed extension of CCS discussed in chapter 2 taking τ as the only urgent action[5]. Thus the predicate $Urgent^{\{\tau\}}(R, t)$ holds iff $R \xrightarrow{\tau}$ or $R \xrightarrow{t'} R' \xrightarrow{\tau}$ for some static process R' and $t' < t$. The condition is modelled after a similar condition introduced in TCCS[186].

3.3 Termination rules

The semantics of the sequential composition operator requires the notion of successfully terminated behaviour. We have modelled this by introducing the special process named $\surd$, denoting the successfully terminated process (as in the calculi of [21], [96], [103] and [156]). (Another way to model termination is the use of a unary predicate that tells us if a process expression has terminated[100].) Consequently, many rules have to make a distinction between the case of a (successfully) terminating component and a non-terminating one. Examples are the rules $|_1$ and $|_2$. Essentially they describe the same behaviour, but they differ in the resulting process depending on the termination of the subprocess.
Working towards a version of the semantics that is more straightforward to implement, we can give an alternative and clearer definition of the semantics of the dynamic processes.
We define an extended set $\mathcal{DP}^e$ of dynamic processes with some additional processes. $\mathcal{DP}^e$ contains all processes of $\mathcal{DP}$, allowing also $t = 0$ in the process $\langle t \rangle$ and allowing the process $\surd$ as a subprocess. Thus $\mathcal{DP}^e$ (ranged over by S^e) is the syntactic set generated by the following grammar ($\alpha \in Act$ and $t \in T$):

$$S^e ::= \alpha \quad | \quad \langle t \rangle \quad | \quad S_1^e ; S_2^e \quad | \quad S_1^e + S_2^e \quad | \quad S_1^e \mid S_2^e \quad | \quad A \quad | \quad \surd$$

Moreover, we define an extended set $\mathcal{SP}^e$ of static processes, ranged over by R^e, which is identical to $\mathcal{SP}$, except for the fact that processes S^e from $\mathcal{DP}^e$ are used

[5] As discussed in chapter 2, the presence of the (negative) urgency condition does not hinder the definition of the transition relation. By the absence of recursion in the static part, the existence of a transition relation satisfying the rules can be shown using a stratification on the size of the static process expressions ([30, 174]).

R^e	$\surd(R^e)$	
α	α	
$\langle t \rangle$	$\langle t \rangle$ $\surd$	if $t > 0$ if $t = 0$
$S_1^e ; S_2^e$	$\surd(S_1^e); S_2^e$ S_2^e	if $\surd(S_1^e) \neq \surd$ if $\surd(S_1^e) = \surd$
$S_1^e + S_2^e$	$\surd$ $\surd(S_1^e) + \surd(S_2^e)$	if $\surd(S_1^e) = \surd$ or $\surd(S_2^e) = \surd$ otherwise
$S_1^e \mid S_2^e$	$\surd(S_2^e)$ $\surd(S_1^e)$ $\surd(S_1^e) \mid \surd(S_2^e)$	if $\surd(S_1^e) = \surd$ if $\surd(S_2^e) = \surd$ otherwise
A	A	
$\surd$	$\surd$	
$R_1^e \Vert R_2^e$	$\surd(R_1^e) \Vert \surd(R_2^e)$	
$R^e[f]$	$\surd(R^e)[f]$	
$R \backslash L$	$\surd(R^e) \backslash L$	

Table 3.4: The termination function

rather than from $\mathcal{DP}$. Now we can deal with terminated subprocesses by using a function $\surd : \mathcal{SP}^e \rightarrow \mathcal{SP}$ that 'cleans up' these terminated subprocesses. This function is defined in table 3.4. For example if the first statement of a sequential composition has terminated, it is discarded ($\surd(\surd; S)$ yields S), or if a subprocess in an interleaving has terminated it is discarded. Note that terminated processes are not removed in the static process operators, in order to retain the static structure of the process, even if a subprocess has terminated.
Using the function $\surd$, we can combine for example rules SEQ_1 and SEQ_2 into the single rule:

$$\frac{S_1 \xrightarrow{\alpha} S_1'}{S_1; S_2 \xrightarrow{\alpha} \surd(S_1'; S_2)} \; SEQ_{1,2}$$

The same is true for the + rules, the interleaving rules and corresponding rules of the time transition relations.
The alternative definition of the semantics of dynamic processes using the termination function is given in table 3.5. Identical rules for action and time transitions have also been combined into a single rule. The semantical rules of the static processes remain unchanged. Notice that although the termination function is only applied in the conclusion of those rules where it is necessary, its definition ensures that it can be applied to the conclusion of *every* rule of both the dynamic and the static processes, without changing the semantics. If $R \in \mathcal{SP}$ then $\surd(R) = R$, in other words, $\surd$ is a projection onto $\mathcal{SP}$. This fact will be used later when the execution of processes is discussed. Notice how this definition of the rules makes them more operational in the sense that application of the rules does require one to predict or determine a priori if the process terminates or not. It demonstrates more clearly that the effect of termination of a subprocess can be applied independently after the transition itself has taken place.

$$\frac{-}{\alpha \xrightarrow{\alpha} \surd}\ ACT_1 \qquad \frac{-}{\ell \xrightarrow{t} \ell}\ ACT_2 \qquad \frac{0 < t' \leq t}{\langle t \rangle \xrightarrow{t'} \surd(\langle t - t' \rangle)}\ DEL$$

$$\frac{S_1 \xrightarrow{\gamma} S_1'}{S_1; S_2 \xrightarrow{\gamma} \surd(S_1'; S_2)}\ SEQ$$

$$\frac{S_1 \xrightarrow{\alpha} S_1'}{S_1 + S_2 \xrightarrow{\alpha} S_1', \quad S_2 + S_1 \xrightarrow{\alpha} S_1'}\ +_1 \qquad \frac{S_1 \xrightarrow{t} S_1', S_2 \xrightarrow{t} S_2'}{S_1 + S_2 \xrightarrow{t} \surd(S_1' + S_2')}\ +_2$$

$$\frac{S_1 \xrightarrow{\alpha} S_1'}{S_1 \mid S_2 \xrightarrow{\alpha} \surd(S_1' \mid S_2), \quad S_2 \mid S_1 \xrightarrow{\alpha} \surd(S_2 \mid S_1')}\ |_1$$

$$\frac{S_1 \xrightarrow{t} S_1', S_2 \xrightarrow{t} S_2'}{S_1 \mid S_2 \xrightarrow{t} \surd(S_1' \mid S_2')}\ |_2 \qquad \frac{A \stackrel{\text{def}}{=} S, S \xrightarrow{\gamma} S'}{A \xrightarrow{\gamma} S'}\ REC$$

$$\frac{S \xrightarrow{t_1} S', S' \xrightarrow{t_2} S''}{S \xrightarrow{t_1 + t_2} S''}\ CNT$$

Table 3.5: Semantics of the action and time transitions of dynamic processes

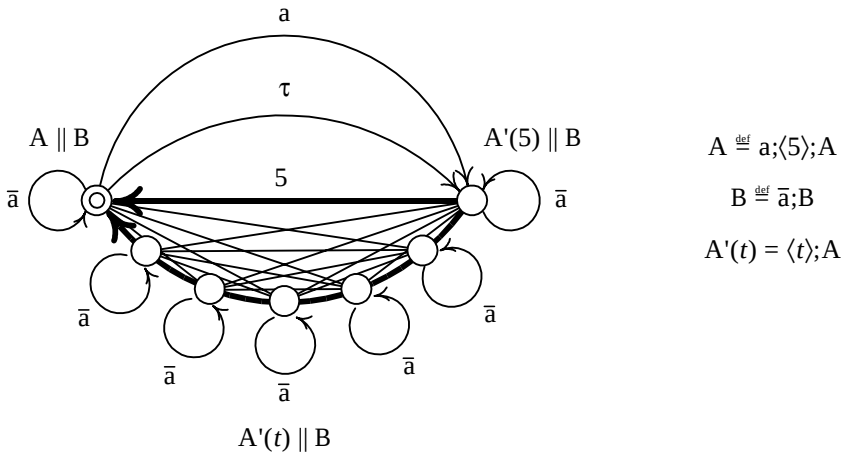


Figure 3.2: Example of a labelled transition system defined by the semantics

3.4 Example

As an example we will look at the timed labelled transition system defined by the process $A||B$, where $A \stackrel{\text{def}}{=} \text{a}; \langle 5 \rangle ; A$ and $B \stackrel{\text{def}}{=} \overline{\text{a}}; B$. The corresponding labelled transition system is shown in figure 3.2. From the initial process (indicated by the extra circle inside the state) there are three possible transitions, one labelled a, one labelled $\overline{\text{a}}$ and their synchronisation τ. The transition labelled $\overline{\text{a}}$ leads back to the initial process (after performing the action $\overline{\text{a}}$, the process B equals B again). Both transitions labelled a and τ (originating from a synchronisation between A and B) lead to the process $(\langle 5 \rangle ; A)||B$. From this process there is a time transition to any process of the form $(\langle t \rangle ; A)||B$ for any $t \in T$ and $0 < t < 5$. Moreover from any state $(\langle t_1 \rangle ; A)||B$ there is a transition to any other state $(\langle t_2 \rangle ; A)||B$ if $0 < t_1 < t_2 < 5$. If T is dense, then this is a continuum of processes and transitions. After a number of time transitions, totalling an amount of 5 units of time, the process returns to the state $A||B$.
The semantics of the jobber process is shown in figure 3.3. The thick circles denote states that are not urgent; i.e. for any $t \in T^+$, there is a transition from the state to itself labelled t; these transitions have been left out of the drawing for clarity. The Jobber process for instance is allowed to let any amount of time pass, waiting for a job to arrive. Moreover, the thick transitions are time transitions and represent a continuum of states and transitions as in the previous example.

3.5 Data

A formal language that is used to model complex distributed systems should not only be able to express system structure, communication and concurrency (as can be done using the calculus as it was introduced), but it should also be capable of expressing data values.
As it would be beyond the scope of this thesis to introduce a calculus for such a language, we limit ourselves here to an indication of how this can be incorporated.

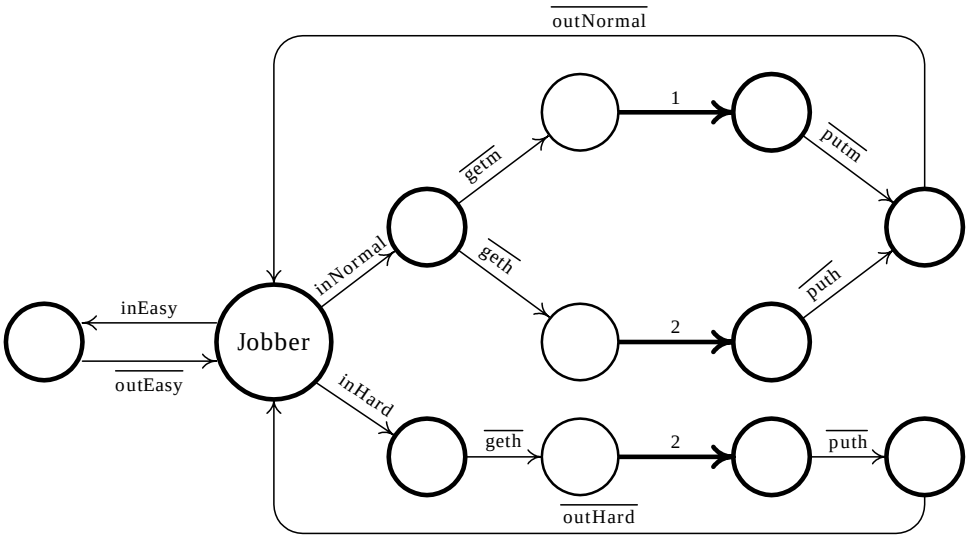


Figure 3.3: Labelled transition system of a jobber

Instead of writing (in the jobshop example),

$$\begin{array}{l} \quad \mathrm{inEasy};\ldots \\ + \\ \quad \mathrm{inNormal};\ldots \\ + \\ \quad \mathrm{inHard};\ldots \end{array}$$

one would prefer to write: $\mathrm{in}(\mathrm{job})$; $\mathit{Process}(\mathrm{job})$; $\overline{\mathrm{out}}(\mathrm{job})$. Allowing to write down the sequence of receiving a job, processing it and delivering it only once, using the type of job as a parameter.

3.5.1 The Value Passing Calculus

In [140], Milner introduces the value-passing calculus in which in addition to the basic CCS processes as described in section 2.4, synchronisation primitives and process constants can be parameterised. For instance, the process expression $\mathrm{a}?(x);P(x)$ denotes the process that can synchronise on port 'a' with an action of the form $\mathrm{a}!v$ carrying some value v from some value domain. Upon receiving a value, it is substituted in every free occurrence of x in the parameterised process expression $P(x)$.
It can be shown (see [140, 109]) that this calculus can be reduced to basic CCS. The input action $\mathrm{a}?x$ can be split into individual actions $\mathrm{a}?v$ for every possible value $v \in \mathit{VAL}$ to be received in x. Every parameterised process $P(x)$ can be split into separate processes $P(v)$ for every $v \in \mathit{VAL}$. Theoretically, this reduction is attractive, since it allows one to reason about a more complex model (the value passing calculus) in terms of a much simpler one (basic CCS). This advantage comes at the cost of the fact that the number of possible transitions of a process may become as large as the

number of distinct values in the domain *VAL*, which will often be infinite. This is a big disadvantage if we try to implement such a calculus.

Another disadvantage of the value passing calculus is that the scope of the 'variable' x is limited to the subexpression $P(x)$. This prohibits the modelling of data that is shared between concurrent entities.

3.5.2 Data Environments

In order to be able to deal with shared data environments, one can introduce a variable environment in which a process executes (see e.g., POOSL[156] or MTCCS[172]).

If we have a set *Id* of variable identifiers and a value domain *VAL*, then a variable environment is a partial mapping $Id \hookrightarrow VAL$. The state of a process is then described by a pair $(P, \mathcal{E})$, consisting of a process expression (parameterised with variables $x \in Id$ and a variable environment $\mathcal{E}$ which provides the values of these variables.

Now the semantical rule for interleaving for instance, may take the following shape.

$$\frac{(P_1, \mathcal{E}) \xrightarrow{\alpha} (P_1', \mathcal{E}')}{(P_1 \mid P_2, \mathcal{E}) \xrightarrow{\alpha} (P_1' \mid P_2, \mathcal{E}')}$$

When process P_1 makes a transition to P_1' (possibly) modifying the data environment from $\mathcal{E}$ to $\mathcal{E}'$, then it is easy to see how this also effects the data environment of P_2.

The notion of a variable environment can and should be refined for modelling of complex systems to allow for variables to have different scopes. In POOSL for instance, the static processes cannot share data, so they have no need for a common variable environment. Within dynamic processes, however, data can be shared. They do have a global data environment (the variables of which are called *instance variables*) and moreover it is possible to employ local variable environments, the scope of which is limited to some subexpression of the process.

3.5.3 Operational Framework for Data Communication

The traditional way to give semantics to processes including data is by reduction to a basic calculus without data. We have seen that this may quickly lead to an infinite number of transitions. To arrive at an implementation for such a calculus, this is unacceptable. In [54] and [109], alternative semantics for Milner's value passing calculus are introduced, which overcome this problem. The problem is in the receiving party. The sending party knows what element(s) of the value domain it wants to send as thus produces only a finite number of actions. The receiving party on the other hand does not known what value will be received and it must therefore offer a synchronisation action for every $v \in VAL$. The solution proposed in both [54] and [109] is simply to let the receiving party produce an action indicating only the name of the port and the parameter which is to be substituted by the value that is passed when the synchronisation takes place. The semantical rules for action prefix

and synchronisation are then as follows:

$$\mathrm{a?}(x) \cdot P(x) \xrightarrow{\mathrm{a?}(x)} P(x)$$

$$\mathrm{a!}(v) \cdot P \xrightarrow{\mathrm{a!}(v)} P$$

$$\frac{P_1 \xrightarrow{\mathrm{a?}(x)} P_1',\ P_2 \xrightarrow{\mathrm{a!}(v)} P_2'}{P_1 || P_2 \xrightarrow{\tau} P_1' || \left(P_2'[v/x]\right)}$$

Some technicalities are involved to prevent the situation that multiple subprocesses have an unbound variable x although only one of them actually performed the communication and should have its variable replaced by the value v. In [109], this is achieved by renaming of variables to guarantee that the same variable is used only once. In [54] an alternative approach is taken, together with the action, the 'address' of the receiving subprocess is recorded. '$\mathrm{a}(x)@\bullet\lceil\rceil$', for instance, is an action label, denoting a synchronisation on port 'a' receiving a value to variable 'x' of the right ($\lceil$) subprocess of the left ($\rceil$) subprocess of the process performing this action. We will use a similar approach (for a different purpose) in the next chapter. So, at the cost of a slightly more complicated semantics, one achieves an interpretation which is implementable.

3.5.4 Data Semantics

We have seen how data operations can be connected to the processes of the calculus. To arrive at a complete calculus, one has to define what data objects or values can be used, what operations they support, etcetera. In POOSL for instance, this is realised in a computation semantics which supports data in an object-oriented fashion, similar to objects in Smalltalk[93] or Java[95].

3.6 Automatic Protection Switching Protocol (2)

We now give a model of the protection switching protocol of section 2.7 in the calculus introduced in this chapter. It consists of two APS nodes modelling the protocol behaviour at the ends of the connection lines. The protocol nodes are influenced by an environment model that changes the conditions of the signal lines and the network management that provides the protocol with external commands.

We have a fixed architecture consisting of the environment, network management and two APS nodes. The APS nodes themselves are built from smaller components. The APS node is composed of a local process, dealing with signal conditions and so-called 'lock-out of protection' of signals, an external process, dealing with commands from the network management, a process dealing with sending and receiving APS messages, and a global process that coordinates the node's status.

The behaviour of the dynamic processes of the APS model can be defined by the

following set of definitions of dynamic process constants.

$$\{LP \stackrel{\text{def}}{=} (\overline{sgnCond} + \overline{lop}); LPCompReq,$$
$$LPCompReq \stackrel{\text{def}}{=} \langle 1 \rangle ; (\tau + (\tau; locReq)); LP,$$

$$EP \stackrel{\text{def}}{=} \overline{extComm}; ((\tau; lop) + (\tau; extReq)); EP,$$

$$GP \stackrel{\text{def}}{=} (\overline{locReq} + \overline{extReq} + \overline{rmtReq}); GPCompReq,$$
$$GPCompReq \stackrel{\text{def}}{=} \langle 3 \rangle ; (\tau + (\tau; glbReq)); GP,$$

$$APS \stackrel{\text{def}}{=} APSGetGlbReq \mid (APSSend + APSReceive),$$
$$APSGetGlbReq \stackrel{\text{def}}{=} \overline{glbReq}; \langle 2 \rangle ; \tau; APSGetGlbReq,$$
$$APSSend \stackrel{\text{def}}{=} sndAPS; \langle 5 \rangle ; APSReceive,$$
$$APSReceive \stackrel{\text{def}}{=} \overline{recAPS}; APSCompReq,$$
$$APSCompReq \stackrel{\text{def}}{=} \langle 1 \rangle ; (\tau + (\tau; rmtReq)); APSSend,$$

$$NM \stackrel{\text{def}}{=} ((\tau; extComm1) + (\tau; extComm2) + \tau); \langle 1 \rangle ; NM,$$

$$EV \stackrel{\text{def}}{=} ((\tau; sgnCond1) + (\tau; sgnCond2) + \tau); \langle 2 \rangle ; EV\}.$$

The local process (*LP*) can receive messages indicating a change in signal conditions (*sgnCond*) or 'lock-out of protection' settings (*lop*). After such a message, a new local request is determined, possibly resulting in a message *locReq* after some delay, signalling a change in local request to the global process. The external process (*EP*) awaits messages *extComm* from the network management. When an external command is received, action is taken in the form of a lock-out of protection message to the local process or a message to the global process conveying a new external request. The global process (*GP*) responds to changes in local, external or remote requests and possibly communicates a new global request to the APS messages process. This process (*APS*) perpetually exchanges APS control messages with a similar process at the opposite side of the connection. The reception of a control message may result in an update of the remote request being sent to the global process. Concurrently, the process executes the behaviour defined by the constant *APSGetGlbReq*, which receives updates of the global request that determine the outgoing APS control messages. The *NM* and *EV* processes define the context in which the protocol operates. *NM* provides the protocol nodes with external commands. At the same time, *EV* influences the signal conditions, triggering the protocol to take appropriate actions.

The model includes static processes incorporating these dynamic behaviours which form the basic building blocks of the system's architecture.

$$LocalProcess \stackrel{\text{def}}{=} LP, \qquad ExternalProcess \stackrel{\text{def}}{=} EP,$$
$$GlobalProcess \stackrel{\text{def}}{=} GP, \qquad APSMessages \stackrel{\text{def}}{=} APS,$$
$$NetworkManagement \stackrel{\text{def}}{=} NM, \qquad Environment \stackrel{\text{def}}{=} EV.$$

From these static processes, one can create larger static structures, such as an entire APS node.

$$APSNode \stackrel{\text{def}}{=} (LocalProcess||ExternalProcess||GlobalProcess||APSMessages)\setminus \{lop, locReq, extReq, glbReq, rmtReq\}.$$

This specifies that the static *APSNode* consists of the parallel composition of the four mentioned static processes synchronously communicating with each other on complementary messages (such as *locReq* and $\overline{locReq}$). Communication with entities outside of the process is restricted to messages other than *lop*, *locReq*, *extReq*, *rmtReq*, *glbReq*, and their complementary messages.
The behaviours of the two APS nodes are identical; they are connected with each other and the rest of the model by an appropriate relabelling:

$$APSNode1 \stackrel{\text{def}}{=} APSNode[extComm1/extComm, aps1/sndAPS, aps2/recAPS, sgnCond1/sgnCond]$$

$$APSNode2 \stackrel{\text{def}}{=} APSNode[extComm2/extComm, aps2/sndAPS, aps1/recAPS, sgnCond2/sgnCond]$$

The entire protocol model is defined by the two nodes in their environment.

$$APSProtocol \stackrel{\text{def}}{=} (NetworkManagement||APSNode1||APSNode2||Environment)\setminus \{extComm1, extComm2, aps1, aps2, sgnCond1, sgnCond2\}.$$

3.7 POOSL models

It has been stated before that the calculus introduced in this chapter is an abstract representation of the language POOSL. The form of calculus expressions is obviously not suitable for the specification of large and complex systems. The architecture of a system is more conveniently represented graphically, as demonstrated in figure 3.4, showing the static architecture of a POOSL model of the APS protocol, consisting of the two APS nodes, a network manager and an environment model, influencing the condition of the signal lines. The dynamic processes that define the behaviour of the individual building blocks are defined by a textual specification language similar to traditional imperative programming languages. The POOSL specification of the processes of the APS protocol can be found in appendix A. An excerpt of the behaviour of the local process is shown in figure 3.5.
Line 1 defines a *method* called `init` without parameters (empty braces). Methods correspond to dynamic process constants of the calculus. Line 2 declares local variables for this method, stating their name and type. We recognise on lines 3-12 a `sel...or...or...les` statement. This is the + operator of our calculus (although not binary). The first alternative (lines 4 and 5) starts with a synchronisation action `sgnCond?changeCondition(n,condition)`, a message reception (indicated by `?`) on channel `sgnCond` with a message named `changeCondition` and two arguments that are to be stored in the variables `n` and `condition`. After that (line 5), an internal action is performed that stores the received information about the

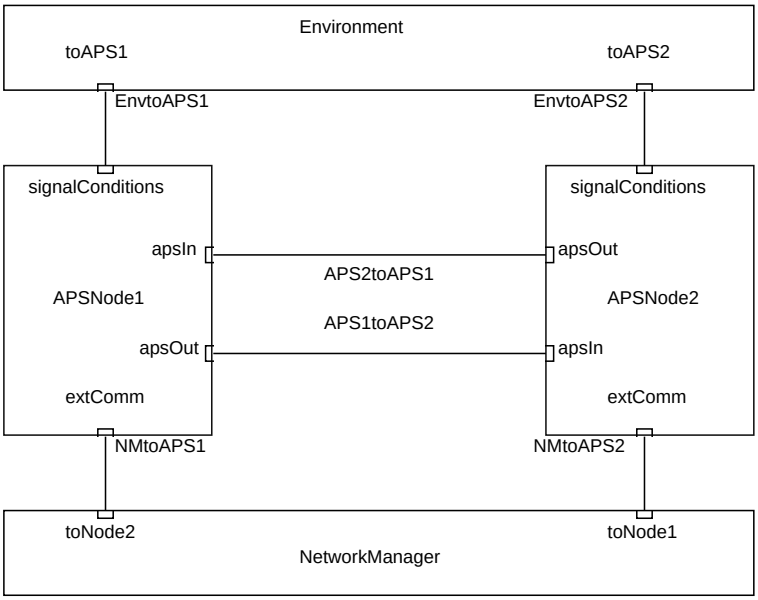


Figure 3.4: Architecture specification in POOSL

`condition` of signal number `n` in a table `signalConditions`. The second alternative (lines 7 and 8) is selected when a synchronisation takes place on channel `lop` receiving a message `lockout` with one argument stored in `n`. This indicates that signal number `n` is to be locked out of protection and this information is stored in the table `lockedout`. The third alternative on lines 10 and 11 is similar to the second, dealing with a message that clears the lockout of a signal. After the selection statement, another method (`updateLocalRequest`) is called, i.e. the behaviour associated with the dynamic constant is invoked. Finally, on line 14, the method itself is invoked again, resulting in the infinite execution of the behaviour.
The formal semantics of the complete POOSL language is given in [156, 80] (the original (untimed) language in [156] and the extension with time in [80]). Other, more complex examples of the application of POOSL to model real-life systems are found in [156, 88, 66, 132, 170, 182].

3.8 Related Work

Process Calculi

Process calculi are widely used for the formal study of concurrent systems and their behaviour. They are also used to give formal semantics to concurrent models and specification or programming languages, such as LOTOS [71] or POOL[16]. Among the most popular ones are CCS [140], CSP [104] and ACP [22]. Their semantics is usually given by transition rules in a structural operational semantics and / or algebraically by means of a set of axioms. An overview of process calculi and process algebra can be found in many textbooks, such as [28].

```
normalOperation()()                                          1
  | n:  Integer; condition:  SignalCondition |               2
  sel                                                        3
    sgnCond?changeCondition(n,condition);                    4
    signalConditions put(n, condition);                      5
  or                                                         6
    lop?lockout(n);                                          7
    lockedout put(n, true);                                  8
  or                                                         9
    lop?clearlockout(n);                                    10
    lockedout put(n, false);                                11
  les;                                                      12
  updateLocalRequest()();                                   13
  normalOperation()().                                      14
```

Figure 3.5: Dynamic process specification in POOSL

Our calculus serves as an abstract place holder for the formal specification language POOSL. An important aspect of POOSL is the ability to deal with complex data. POOSL incorporates dynamic data objects like the objects of Smalltalk or Java. Data is incorporated in the semantics in a style similar to POOL[16] or MTCCS[172], allowing data to be shared between concurrent processes. Other efforts to obtain an operational characterisation of processes with data are found in [140, 54, 109]. These do not allow the description of processes that share data, but they can be reduced to a basic calculus without data.

Another aspect of POOSL is timing, using a dense time domain. POOSL's timing model is included in the calculus used in this thesis. Many extensions to process calculi have been designed to add quantitative (discrete or dense) timing to processes, [18, 20, 64, 31, 49, 62, 96, 98, 139, 147, 159, 161, 186] to name some. An overview of decisions to be made when designing a timed process calculus can be found in [147]. In our calculus, time is additive, deterministic and our processes display maximal progress. Maximal progress is desirable if we want to interpret the system as an executable model. The model should not be allowed to let arbitrary amounts of time go by when progress can be made. Maximal progress is usually not achieved without some extra trouble in the form of negative premises or extra transition relations. In TPCCS of [98] for instance, a discrete time model is used, first a 'virtual' time transition relation (not taking urgent actions into account) is defined and then based upon that the 'real' time transitions are defined allowing a process to 'tick' (let a discrete amount of time go by) only if it cannot perform an action. Similarly, but without the extra relation, directly using a condition on the parallel composition (so also in a discrete time domain using individual tick actions), RtCCS[161] achieves maximal progress. In other calculi such as [143], actions cannot be made urgent. Normal prefixed actions cannot wait. Actions can be made delayable with an explicit operator, but then they are never urgent. Another option is to add urgency as a separate operator. This is done in U-LOTOS [32]. Its definition also requires the use of negative premises. Although it is mathematically attractive to separate the notions of communication and urgency into different operators, it is not practical in a specification language, since it requires the use of more language primitives.

Specification Languages for Complex Concurrent Systems

Formal models are good to specify behaviour precisely and unambiguously and for studying the very nature of concurrent systems. To be used for the actual specification of large and complex systems, a lot more is necessary, such as a practical syntax, methodological guidance and tool support. Languages such as UML[176], Statecharts[99], SDL[151], and also POOSL are designed to be able to deal with practical specifications and designs. To this end, they provide for instance suitable structuring concepts that are purely syntactical and uninteresting from a formal point of view, but make life easier for a practical designer.

3.9 Conclusions

We have presented the syntax and semantics of a calculus to model distributed concurrent systems. The calculus particularly focusses on a static structure of communicating components. The calculus is similar to (real-time extensions of) Milner's CCS. In particular it is used for its resemblance to the (more complex) formal specification language POOSL. Attention has been paid to the concept of maximal progress and the way it is achieved by the calculus.
The concept of termination plays an important role in the calculus and we have given an alternative presentation of the transition rules, using the function $\surd$ which removes the necessity to discriminate between terminating and non-terminating transitions in many of the semantical rules. Moreover, it gives the rules a more operational flavour, which is useful for the execution of processes, described in the next chapter. An example of a process and its corresponding transition system have been given to clarify the material of this chapter. We have also briefly discussed some aspects of adding data to a calculus like this one and given an example of a POOSL model. Although interesting and valuable in its own right, we have not included an equational or axiomatic treatment of our calculus, since it is beyond the scope and focus of this thesis. In the following chapter we turn toward the effective execution of processes of the calculus. This provides a basis for the formal simulation and verification of its models.

Chapter 4

Executing Real-Time Concurrent Models

For many types of automatic analyses on the models described in the calculus of the previous chapter, it is required to generate the transition system that is induced by the process description and the semantics of the calculus. For instance, to perform a random simulation through the transition system one can generate all possible transitions from a given state, pick one of these transitions and move to the destination state of the transition. Model-checking and reachability analysis of processes also require the construction of the transition system. In this chapter a method is described to generate the available transitions of processes of the calculus and to effectively compute the destination process of a chosen transition.

In section 4.1, 'requests' are introduced. They are used to identify the available transitions of a process. Section 4.2 shows how, given a process expression and such a request, the destination of the transition corresponding to this request is constructed. The ingredients for an effective implementation of such a scheme are discussed in section 4.3. The suggested implementation can be characterised by a new labelled transition system, built upon requests. The fact that this transition system is in some sense a faithful implementation of the calculus is made precise and is demonstrated in section 4.5. Some preliminary definitions are given in section 4.4 to characterise this implementation relationship.

The techniques introduced in this chapter have been applied to construct simulation tools for the formal specification language POOSL. POOSL is an industrial strength specification language of a much higher complexity than the abstract calculus of this thesis. The constructed tools are not merely a prototype implementation, they are efficient and user-friendly enough to have been applied to a number of practical projects involving POOSL specifications. Section 4.6 briefly describes these tools as well as some additional issues arising in an implementation of the full language POOSL as opposed to the abstract calculus. Related work is discussed in section 4.7 and section 4.8 summarises conclusions.

4.1 Requests

The calculus defined in the previous chapter can be executed by collecting the possible transitions of the process compositionally from its syntactic structure. These possible transitions are further decorated to make them uniquely identifiable. Such labelled transitions will be referred to as 'requests'. From a process and a request, the process that is related to it by this request, can be computed.
We first define the set AR of *action requests* as decorated labels of the transition relations and a corresponding function ACT that removes the decoration and yields the original action. Normally the labels of the transitions of a process expose the externally visible behaviour only. The decoration adds to this information about which subprocess actually produced the action. Comparable approaches are taken in [54], [37] and [180] for different purposes.

4.1.1 DEFINITION *The set AR of* action requests *and the* reduction function ACT *which yields a request's corresponding action, are simultaneously defined inductively by the following rules.*

- *Undecorated actions are also requests. $Act \subseteq AR$ and if $r \in Act$ then $ACT(r) = r$:*
$$\frac{\alpha \in Act}{\alpha \in AR} \qquad \frac{\alpha \in Act}{ACT(\alpha) = \alpha}.$$
- *If r is an action request, then so are $L \cdot r$ and $R \cdot r$. They are used for binary combinators such as choice ($+$) and parallel composition to designate the origin of the request as either its left or as its right subprocess. $ACT(L \cdot r) = ACT(R \cdot r) = ACT(r)$.*
$$\frac{r \in AR}{L \cdot r \in AR, R \cdot r \in AR} \qquad \frac{ACT(r) = \alpha}{ACT(L \cdot r) = \alpha, ACT(R \cdot r) = \alpha}.$$
- *If r_1 and r_2 are complementary action requests, then their synchronisation is represented by the action request $\tau \cdot (r_1, r_2)$, the synchronisation yields a τ-request and the individual requests from which it originates, are remembered. $ACT(\tau \cdot (r_1, r_2)) = \tau$.*
$$\frac{r_1 \in AR, r_2 \in AR, ACT(r_1) = \overline{ACT(r_2)}}{\tau \cdot (r_1, r_2) \in AR} \qquad \frac{-}{ACT(\tau \cdot (r_1, r_2)) = \tau}.$$
- *If $\ell \in \mathcal{L}$ and r is an action request then $\ell \cdot r$ is an action request and $ACT(\ell \cdot r) = \ell$. This type of action request is used when an action is renamed by a relabelling operator. The new label is placed in front of the request and the name before the relabelling can be deduced from the rest of the request.*
$$\frac{\ell \in \mathcal{L}, r \in AR}{\ell \cdot r \in AR} \qquad \frac{-}{ACT(\ell \cdot r) = \ell}.$$

□

S	$AReq$	$TReq$
ℓ	$\{\ell\}$	T^+
τ	$\{\tau\}$	$\varnothing$
$\langle t \rangle$	$\varnothing$	$(0,t]$
$S_1; S_2$	$AReq(S_1)$	$TReq(S_1)$
$S_1 + S_2$	$L \cdot AReq(S_1) \cup R \cdot AReq(S_2)$[a]	$TReq(S_1) \cap TReq(S_2)$
$S_1 \mid S_2$	$L \cdot AReq(S_1) \cup R \cdot AReq(S_2)$	$TReq(S_1) \cap TReq(S_2)$
A ($A \stackrel{\text{def}}{=} S$)	$AReq(S)$	$TReq(S)$
$\surd$	$\varnothing$	T^+

[a] $L \cdot AReq = \{L \cdot r \mid r \in AReq\}, R \cdot AReq = \{R \cdot r \mid r \in AReq\}$

Table 4.1: Computing the requests of a dynamic process

R	$AReq$	$TReq$
S	$AReq(S)$	$TReq(S)$
$R_1 \| R_2$	$L \cdot AReq(R_1) \cup R \cdot AReq(R_2) \cup Comm(R_1, R_2)$[a]	$TReq(R_1) \cap TReq(R_2) \cap Urg(R_1, R_2)$[b]
$R[f]$	$Relabel\,(R, f)$[c]	$TReq(R)$
$R \backslash L$	$\{r \in AReq(R) \mid ACT(r) \notin L\}$	$TReq(R)$

[a] $Comm(R_1, R_2) = \{\tau \cdot (r_1, r_2) \mid r_1 \in AReq(R_1), r_2 \in AReq(R_2), ACT(r_1) = \overline{ACT(r_2)}\}$

[b] $Urg(R_1, R_2) = \begin{cases} T^+ & \text{if } Comm(R_1, R_2) = \varnothing \\ \varnothing & \text{otherwise} \end{cases}$

[c] $Relabel(R, f) = \{f(ACT(r)) \cdot r \mid r \in AReq(R)\}$

Table 4.2: Computing the requests of a static process

Time requests are used to represent the ability of a process to perform a time transition. As time requests we use labels of the time transition relation and they need no further decoration. Thus *time requests* are positive elements of the time domain T. Let the set Req of all (action and time) requests equal $AR \cup T^+$ and let r be a typical element of Req.

The action and time requests of a process are given by the functions $AReq : \mathcal{SP} \to AR$ and $TReq : \mathcal{SP} \to T^+$ as defined in tables 4.1 and 4.2 for dynamic and static processes respectively. The process ℓ for instance produces a single request, ℓ. Conform to the semantical rule $+_1$, the process $S_1 + S_2$ draws its requests from both the requests of S_1 and S_2, adding the appropriate decoration to be able to distinguish between them. The process $\langle t \rangle$ has in accordance with the rule *DEL* all time requests in $(0,t]$. The time requests of $S_1 \mid S_2$ are precisely the time requests occurring in both S_1 and S_2, corresponding to the rule $|_2$. Note that the sequential composite $S_1; S_2$ derives its time requests from its first composite S_1 only. Therefore, there will not always be a time request corresponding to every possible time transition of the process. Note also that the restriction to guarded recursion in the process constants ensures that the functions are well-defined.

The requests associated with synchronisation process ℓ are the action request of the possible synchronisation transition ℓ itself, and every possible time request, since the synchronisation is not urgent by itself. There is only one request of the τ pro-

cess, since it represents an urgent internal action. The delaying process $\langle t \rangle$ has no action requests but is willing to delay for any amount of time up to and including t. The binary operators $+$ and $|$ have a request for every request of any of their subprocesses. These requests are decorated to indicate the origin of the request. The terminated process does not engage in any actions, but allows arbitrary amounts of time to pass.
For the parallel composition of two static processes, the set of action requests corresponds to the actions originating from both subprocesses and all possible synchronisations between complementary actions of the subprocesses. Looking at the time requests of the parallel composition we observe that there are only time requests if both subprocesses allow time to pass and the parallel composition is not urgent. One only needs to check whether the combined processes are urgent *now*. If they are not urgent now, and $TReq(R_1) \cap TReq(R_2) = (0, t]$ then there is no $t' < t$ such that $R_1||R_2 \xrightarrow{t'} R_1'||R_2'$ and $R_1'||R_2'$ is urgent. It can be proved (by structural induction) that the action requests do not change after such a delay. To do this we first show that no process may terminate by this kind of time transition.

4.1.2 LEMMA *Let $R \in \mathcal{SP}$, $R \neq \surd$, $t \in TReq(R)$ and $0 < t' < t$, then for all $R' \in \mathcal{SP}$ such that $R \xrightarrow{t'} R'$, $R' \neq \surd$.*

PROOF Straightforward, by transition induction. □

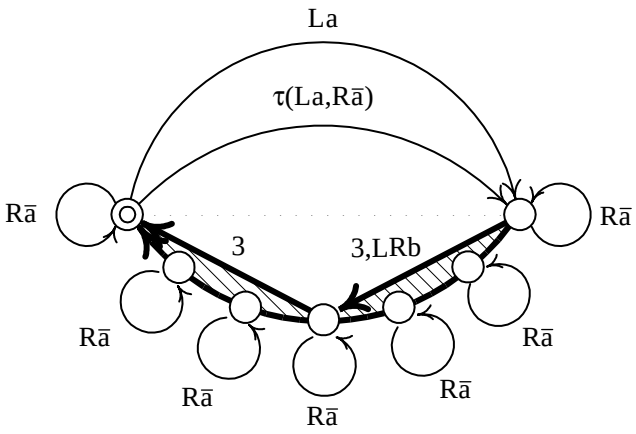
This fact is now used to show that a delay smaller than the maximum time request does not change the action requests of a process.

4.1.3 LEMMA *Let $R \in \mathcal{SP}$, $t \in TReq(R)$ and $0 < t' < t$, then for all $R' \in \mathcal{SP}$ such that $R \xrightarrow{t'} R'$, $AReq(R') = AReq(R)$.*

PROOF By transition induction. We show only the cases $R = S_1; S_2$ and $R = S_1 + S_2$, the other cases are similar. (Remember that a dynamic process S is also a static process.)

- If $R = S_1; S_2$, then $t \in TReq(S_1)$. If $R \xrightarrow{t'} R'$ then (by lemma 4.1.2 and the rules SEQ_3 and SEQ_4) there is some S_1' such that $S_1 \xrightarrow{t'} S_1'$ and $R' = S_1'; S_2$, by induction, $AReq(R') = AReq(S_1') = AReq(S_1) = AReq(R)$.
- If $R = S_1 + S_2$, then $t \in TReq(S_1) \cap TReq(S_2)$. If $R \xrightarrow{t'} R'$ then (by lemma 4.1.2 and the rules $+_2$ and $+_3$) there exist S_1' and S_2' such that $S_1 \xrightarrow{t'} S_1'$, $S_2 \xrightarrow{t'} S_2'$ and $R' = S_1' + S_2'$, by induction, $AReq(S_1') = AReq(S_1)$ and $AReq(S_2') = AReq(S_2)$ from this it follows immediately that $AReq(R') = AReq(R)$. □

This property allows one to determine the length of a time transition during which no processes will become urgent. Figure 4.1 shows an example where the transitions corresponding to the requests are shown. (In the following sections it is made precise how one can interpret a process and its requests as a labelled transition system.) The right hatched area represents a continuum of time transitions in the usual way and transitions labelled 'LRb', all leading to the process $(\langle 3 \rangle; A)||B$. The left hatched area is a continuum of time transitions only.
Figure 4.2 shows a labelled transition system defined by the semantics of some process S_1 and a corresponding labelled transition system defined by the requests. Note



A‖B

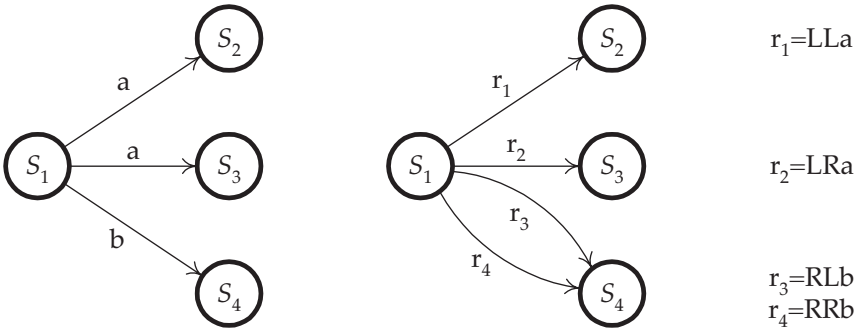
where

A $\stackrel{\text{def}}{=}$ a;(⟨3⟩+b);⟨3⟩;A

B $\stackrel{\text{def}}{=}$ $\overline{a}$;B

Figure 4.1: Time transitions without the CNT rule

S_1=(a;S_2+a;S_3)+(b;S_4+b;S_4)



r_1=LLa

r_2=LRa

r_3=RLb

r_4=RRb

Figure 4.2: Processes, transitions and action requests

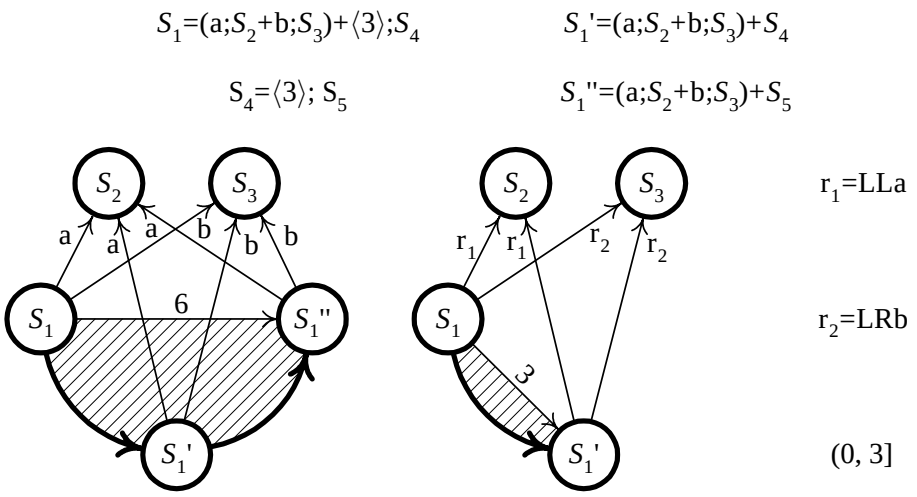


Figure 4.3: Processes, transitions and requests with time

how every transition starting from a process has a unique label. Moreover, even though there is only one transition labelled 'b' in the labelled transition system on the left, there are two requests, r_3 and r_4, such that $ACT(r_3) = ACT(r_4) = \mathrm{b}$, corresponding to the two ways to derive the existence of the transition.
Figure 4.3 shows the labelled transitions system and the corresponding transition system of requests, of a process with a delay operator. The process S_4 can delay for 3 units of time. This makes that process S_1 has time transitions for every t with $0 < t \leq 6$. The corresponding transition system of requests on the right, however, has only requests for time transitions up to 3, i.e. up to the termination of the delay operator. Hence, for the generation of the requests of S_1, one does not need to inspect S_4.

4.1.1 Properties of the requests of a process

We now list a few properties of the requests that illustrate the correspondence with the transitions of a process.
Every action request of R corresponds to some transition of R.

4.1.4 LEMMA *For every $R \in \mathcal{SP}$, $r \in AReq(R)$ there is some $R' \in \mathcal{SP}$ such that $R \xrightarrow{ACT(r)} R'$.*

A more general lemma is proved in section 4.5 (lemma 4.5.2). The converse of this lemma states that for every action transition of R, there is a corresponding request.

4.1.5 LEMMA *If $R \xrightarrow{\alpha} R'$ then there is some $r \in AReq(R)$ such that $ACT(r) = \alpha$.*

Similar properties can be stated about time transitions and requests. The following lemmas are also made more precise in section 4.5, in the form of lemmas 4.5.4 and 4.5.7.

4.1.6 LEMMA *For every $R \in \mathcal{SP}$, $t \in TReq(R)$ there is some $R' \in \mathcal{SP}$ such that $R \xrightarrow{t} R'$.*

W.r.t. the converse of the previous property, note that it is not true that every time transition of R can be matched by a time request of R. But if the process has some time transition, then there is at least some time request.

4.1.7 LEMMA *For every $R \in \mathcal{SP}$, if $R \xrightarrow{t} R'$ for some $t \in T^+$ then $TReq(R) \neq \varnothing$.*

We repeat the lemma of the previous section, stating that any time transition made by the process smaller than the maximum time request of the process cannot change the set of its action requests.

4.1.3 LEMMA *Let $R \in \mathcal{SP}$, $t \in TReq(R)$ and $0 < t' < t$, then for all $R' \in R$ such that $R \xrightarrow{t'} R'$, $AReq(R') = AReq(R)$.*

As a consequence of this lemma, the urgency status of the process cannot change.

4.1.8 COROLLARY *Let $R \in \mathcal{SP}$, $t \in TReq(R)$ and $0 \leq t' < t$, then $Urgent^{\{\tau\}}(R, t')$ iff $Urgent^{\{\tau\}}(R)$.*

This is important, because it allows one to move forward for any amount of time in $TReq(R)$ and still guarantee maximal progress, knowing that no τ-transition has become available in the mean time.

One can also easily show (by structural induction on R) that the set of time requests is always of a particular shape.

4.1.9 LEMMA *For every $R \in \mathcal{SP}$, either $TReq(R) = T^+$ or there is some $t \in T$ such that $TReq(R) = (0, t]$.*

PROOF Shown by structural induction. The basic processes ℓ, τ and $\langle t \rangle$ produce sets of time requests of such shape and the shape is preserved by finite unions and intersections of such intervals, as performed by *TReq*. □

The time requests respect urgency, the set of time requests of a process is empty if and only if it can perform a τ action.

4.1.10 LEMMA *For every $R \in \mathcal{SP}$, $TReq(R) = \varnothing$ iff there is some $r \in AReq(R)$ such that $ACT(r) = \tau$.*

PROOF This is also proved by structural induction, where the most interesting rule is the one for parallel composition $R_1 || R_2$, which intersects the requests of R_1 and R_2 with $Urg(R_1, R_2)$ which disallows all time transitions if R_1 and R_2 can communicate. □

Finally, we state a fact about the action requests that is important if we want to use them for an implementation of a process specified in the calculus: the number of action requests of a process is always finite.

4.1.11 LEMMA *For every $R \in \mathcal{SP}$, $|AReq(R)| < \infty$.*

PROOF This is shown by structural induction. The basic processes produce a finite number of action requests. These finite sets of action requests are combined by the operators in different ways to form new finite sets of requests. In particular, the fact that recursion is guarded makes that this combination is performed only a finite number of times, resulting in a finite total number of action requests. □

R	r	$Grant(R,r)$
α	α	$\surd$
	t	α
$\langle t \rangle$	t'	$\langle t - t' \rangle$
$S_1; S_2$	r	$Grant(S_1, r); S_2$
$S_1 + S_2$	$L \cdot r'$	$Grant(S_1, r')$
	$R \cdot r'$	$Grant(S_2, r')$
	t	$Grant(S_1, t) + Grant(S_2, t)$
$S_1 \mid S_2$	$L \cdot r'$	$Grant(S_1, r') \mid S_2$
	$R \cdot r'$	$S_1 \mid Grant(S_2, r')$
	t	$Grant(S_1, t) \mid Grant(S_2, t)$
A ($A \stackrel{\text{def}}{=} S$)	r	$Grant(S, r)$
$\surd$	t	$\surd$
$R_1 \Vert R_2$	$L \cdot r'$	$Grant(R_1, r') \Vert R_2$
	$R \cdot r'$	$R_1 \Vert Grant(R_2, r')$
	$\tau(r_1, r_2)$	$Grant(R_1, r_1) \Vert Grant(R_2, r_2)$
	t	$Grant(R_1, t) \Vert Grant(R_2, t)$
$R[f]$	$\alpha \cdot r'$	$Grant(R, r')[f]$
$R \backslash L$	r	$Grant(R, r) \backslash L$

Table 4.3: Computing the process after executing a request

4.2 Computing the new process

Every request of a process uniquely determines a derivation of one of the possible transitions of a process expression. In this section it is shown how the resulting process of a particular transition can be determined from the process and a request. The function $Grant : \mathcal{SP} \times Req \hookrightarrow \mathcal{SP}^e$ shown in table 4.3 defines the effect of executing a particular request. *Grant* is a partial function, but it is defined for every (R, r) such that $r \in AReq(R) \cup TReq(R)$. If the granted request is an action request, it is being passed down to the subprocess from which it originates. This is possible due to the decoration that was added to the request. When a delay request is granted, it is distributed over the tree corresponding to the synchronous nature of time passage. If an action request is granted to the action process α, then it can only be the action request α and the effect is the termination of the process. The granting of a time request, corresponding to a time transition has no effect on the process α. Action requests will never be granted to the process $\langle t \rangle$. If a time request t' is granted, then $0 < t' \leq t$ and the result is $\langle t - t' \rangle$. Note that if $t' = t$ then the result is $\langle 0 \rangle$, which is not in $\mathcal{SP}$, but in $\mathcal{SP}^e$. In case of sequential composition $S_1; S_2$, any request originates from S_1 and consequently the effect of granting the request to S_1 is computed to determine the effect of granting the request to the sequential composite. Choice and interleaving operators direct action requests to the correct subprocess, retrieved from the decoration, and distribute time requests along both subprocesses. Granting a request to A, defined as S, is identical to granting the request to S. $\surd$ will never have an action request granted and time requests do not have any effect on it. The static processes deal with granting requests in a similar way.
The result of the grant function is a process in $\mathcal{SP}^e$; i.e. it might contain $\surd$ or $\langle 0 \rangle$

as a subprocess. After granting a request, we have to take care of these terminated subprocesses. This is exactly what the $\surd$ function of the previous chapter does. A function $\mathit{Step} : \mathcal{SP} \times \mathit{Req} \hookrightarrow \mathcal{SP}$ can now be defined as the composition of the *Grant* and the $\surd$ function.

$$\mathit{Step} = \surd \circ \mathit{Grant}$$

The application of this function represents a particular transition of the process for any of its requests and computes the resulting process expression. In section 4.5 it will be shown that requests and the *Step* function correspond to the original semantics.

4.3 Implementation

In the previous sections, requests have been defined, based upon the semantics of the calculus. Their relationship to the semantics has been shown and they will serve as a good basis for the implementation that is discussed in this section. In the new transition system based upon requests, every available transition is uniquely identified by a request. The function *Step* introduced in the previous section demonstrates how, given a process and a request, the resulting process can be effectively constructed.
In order to arrive at an implementation of processes expressed in the calculus, there are a few issues that remain to be addressed. We shall describe how one can effectively (finitely) represent processes and their requests and perform the *Step* function when a request has been selected. Finally, the semantics of the calculus specifies intrinsic non-deterministic behaviour. An implementation should define a way to resolve all non-deterministic choice in the model. Such a mechanism is usually called a scheduler. This topic is discussed in section 4.3.4.

4.3.1 Trees representing process expressions

To execute the behaviour of a process we need to have a (finite) representation of the process. Since the semantics of processes as well as requests and other functions are naturally defined along the lines of the syntactic structure of the process, a suitable representation is a tree which corresponds to the syntactic structure of the process expression.
Requests are generated by the leaves of the tree (synchronisations, delays and internal actions) and collected or manipulated by the different operators to form their own requests. To compute their requests, process constants occurring in the expression are replaced by their defining process expressions. This yields a finite tree since recursion is always guarded with a sequential composition operator and the sequential composition draws its requests from its first process only. When a transition is made the *Step* function is performed on the tree to obtain a new tree corresponding to the destination of the transition. In practice this can be a new data structure representing the tree that corresponds to the destination of the transition. It might also be a modification of the data structure of the source of the transition, transforming it into a representation of the target of the transition. The former is useful, for instance, for explicitly constructing the labelled transition system corresponding to the process. If the trees are large and in particular if they contain large amounts of

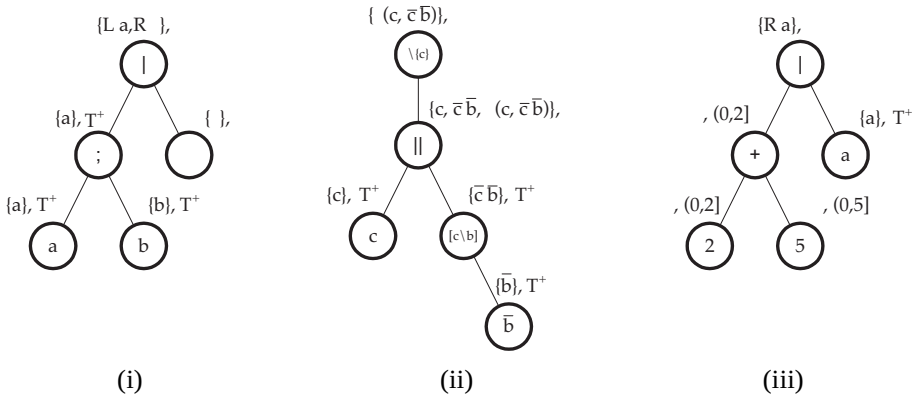


Figure 4.4: Compositionally generating the requests of the processes. (i) $(a;b) \mid \tau$ (ii) $(c||(\overline{b}[c/b]))\backslash\{c\}$ and (iii) $(\langle 2\rangle + \langle 5\rangle) \mid a$.

data as well, then it is far more efficient to modify the tree than to create a new one by replicating its (modified) structure as well as all of its data. This is possible for instance in forward simulations, where the history of the path through the transition system is not important. In particular, we know that the static operators remain the same after any transition.

4.3.2 Generating requests

The requests of a process are computed in the tree from bottom to top. First, the leaves compute their sets of requests and the different operators generate their requests based upon the requests of their subprocesses according to the definitions of *AReq* and *TReq* in tables 4.1 and 4.2. In figure 4.4 a few examples are shown. The process expression is represented as a tree and at every node, the sets of action requests and time requests are displayed. In example (i), the synchronisation node labelled 'a' is willing to engage in a synchronisation 'a' and is prepared to let an arbitrary amount of time pass while waiting for a synchronisation partner. The sequential composition node above it inherits all requests from its first subprocess (according to the semantical rules SEQ_1 and SEQ_2). As all requests are known to originate from the first subprocess, no further decoration is required. The leaf node labelled τ is requesting an internal action transition and urgency requires that there are no time requests. Finally, the interleaving operator inherits all action requests, adding information to remember which subprocess they belong to. The time requests of the interleaving are only those time requests that occur in every subprocess and in this case there are none, since its right subprocess cannot let any time pass before performing the τ action.

In example (ii) the same is shown for the process $(c||(\overline{b}[c/b]))\backslash\{c\}$. Here we see how the relabelling operator modifies the name of the request and remembers the old name for the moment when the request is granted. The parallel composition operator combines requests like the interleaving operator, but additionally combines requests into pairwise synchronisations (remembering the original requests that have

been used to build it). Furthermore, time requests are not generated because of the available synchronisation, which is urgent. Finally, the node corresponding to the hiding operator only passes the τ request, all requests corresponding to actions 'c' or '$\overline{c}$' are restricted. The third example shows how the delay operator generates a bounded set of time requests. The $+$ and $|$ operators combine the time requests by collecting those that are present in both of their subprocesses, reflecting the synchronous nature of the passing of time.
The correspondence between the sets of requests and the rules for generating the transition relations, can be illustrated by the derivation of the time transition labelled 2 corresponding to the request '2' in figure 4.4 (iii).

$$\dfrac{\dfrac{\dfrac{0 < 2 \leq 2}{\langle 2\rangle \xrightarrow{2} \surd}\,DEL \qquad \dfrac{0 < 2 \leq 5}{\langle 5\rangle \xrightarrow{2} \langle 3\rangle}\,DEL}{\langle 2\rangle + \langle 5\rangle \xrightarrow{2} \surd}\,+_2 \qquad \dfrac{-}{\mathrm{a} \xrightarrow{2} \mathrm{a}}\,ACT_2}{(\langle 2\rangle + \langle 5\rangle) \mid \mathrm{a} \xrightarrow{2} \mathrm{a}}\,|_2$$

In the derivation, one can recognise an upside-down version of the tree. The rules correspond to the operator nodes of the tree and the requests are generated according to the rules of the operator.

Representing time requests

The time domain is usually infinite and possibly even uncountable. This makes it impossible to represent every possible time request individually. Fortunately, the set of time requests is always a subset of the time domain of the form $(0, t]$ for some $t \in T$, or it is T^+. Therefore it is possible to represent the set of time requests using the maximum of the requests, or ∞ if there is no maximum. This form is also maintained when the time transitions of a composite process are determined from its subprocesses. As seen in the previous chapter, the set of action requests of a process is always finite since recursion is restricted to guarded processes.

4.3.3 Granting requests

Once all the requests of a process expression have been generated, it is possible to perform a step in the computation by selecting a request and applying the function *Grant*. Again, the compositional nature of the grant function suggests how granting a request of a process is achieved by granting the corresponding request to the subprocess or subprocesses from which it originated.

Granting an action request

In figure 4.5, it is shown how the request $L \cdot \mathrm{a}$ is granted to the process $(\mathrm{a}; \mathrm{b}) \mid \tau$ (example (i) of section 4.3.2). The interleaving operator recognises it as a request of its left subprocess and computes the *Grant* of the request 'a' of the left process. The sequential composition grants the request to its first subprocess. The synchronisation 'a' terminates as a result of the granted request.

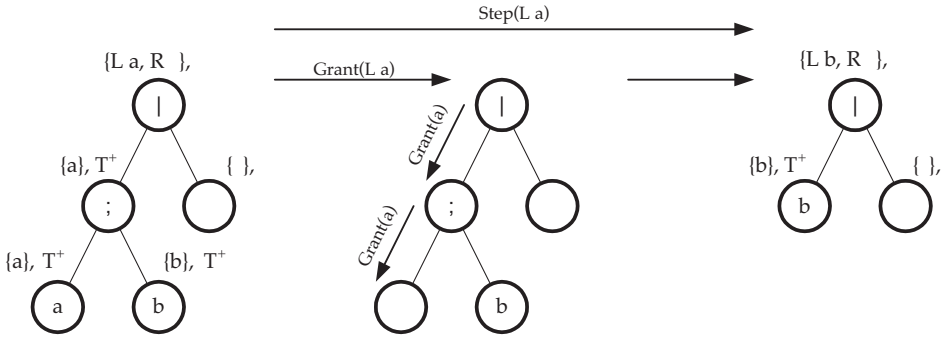


Figure 4.5: Granting a request of a process and computing the resulting process

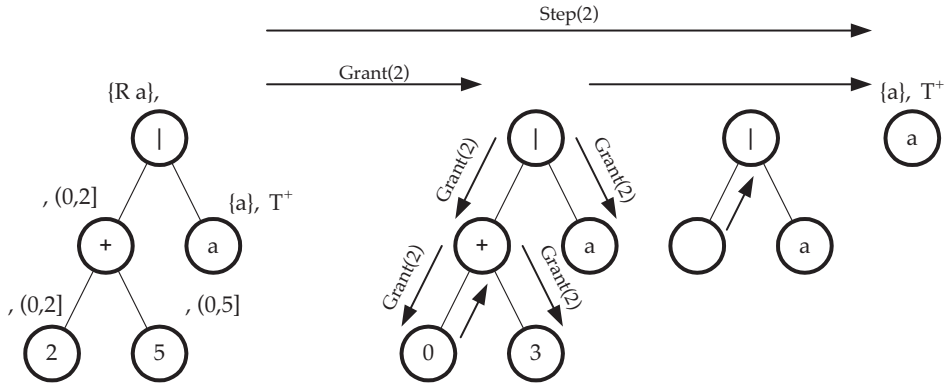


Figure 4.6: Granting a delay request of a process and computing the resulting process

To complete the transition, the $\surd$ function is applied. This can be viewed as a bottom-up procedure starting from the leaves of the tree. It can be started from the leaves that were affected by the granting of the request only. In the example, the now terminated synchronisation makes that the sequential composition is replaced by its right subprocess. After this update, the transition is complete. The new requests can be computed in preparation of the next step. Note that not all of the requests need to be recomputed. Subprocesses that were not affected by the transition, do not change their requests.

Granting a time request

When a time request is granted to a process, a similar thing happens. By the synchronous nature of the time transitions however, the time requests are distributed along all subprocesses except for the sequential composition operator, where it is passed only to the first subprocess. In figure 4.6, a time request is granted to the process of figure 4.4 (iii). The time request is distributed over the process and both delay operators decrease their remaining delay by 2, the amount of time that has been granted. The left of the delay operators has decreased to 0. In the termination

phase, the expired delay leads to the termination of the + operator and finally to the removal of the interleaving operator leaving only the synchronisation 'a' to be executed.

4.3.4 Schedulers

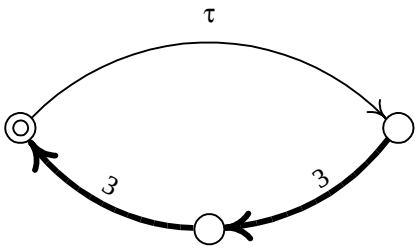
Once we can generate the transition system by computing the sets of requests and are able to determine the process resulting from the transition corresponding to a request, it is possible to 'execute' the behaviour described by a process. In order to execute a process, a *scheduler* has to be attached. At every state, the process may have a number of available transitions. A scheduler is an entity which resolves the non-deterministic choice between the available transitions according to some predefined mechanism.
Although the semantics does not make any explicit assumptions about what constraints a scheduler should satisfy, it is often assumed that a scheduler is at least in some sense fair (see [79]) with respect to all actions, transitions or (sub)processes. What is considered to be an unfair treatment can be defined in many different ways. It is often required that a transition that is infinitely often available during an execution must at some point be taken ('strong fairness'). A weaker version of fairness states that a transition that is continually enabled must at some point be taken. In the latter case one refers to local transitions of one of the system's components, rather than the global transitions of the transition system. Such definitions only pertain to infinitely long traces. In case of a simulation, a simple way to guarantee strong fairness is to apply a random selection at every step, attributing a positive probability to all enabled requests. This results in a zero probability of generating an unfair execution.
The combination of the system and an appropriate scheduler produces a (timed) trace, a sequence of processes and transitions.

$$R_1 \xrightarrow{\gamma_1} R_2 \xrightarrow{\gamma_2} R_3 \xrightarrow{\gamma_3} \dots$$

Recall that the γ_i are either actions or delays.

4.3.5 Open and closed systems

Processes can be executed as a closed system, assuming that there are no external processes they can synchronise with. This makes that only internal transitions are enabled. In this interpretation, only requests corresponding to internal actions can be executed. Alternatively one can interpret the process as an open system, assuming the possibility of external processes synchronising on externally visible actions. In that case, any request of a process can be selected. In case of abstract actions this can easily be done. If messages contain data, however, it is often hard to use this interpretation because not every possible data value corresponds to a meaningful message. As a consequence it is common in the analysis of complex systems to model not only the system that is being designed, but also the environment in which it operates. Then, the system together with its environment is interpreted as a closed system.



A||B

where

A $\stackrel{\text{def}}{=}$ a;(⟨3⟩+b);⟨3⟩;A

B $\stackrel{\text{def}}{=}$ ā;B

Figure 4.7: Closed interpretation of a labelled transition system

If the labelled transition system of figure 4.1 is interpreted as a closed system, only the internal transitions and time transitions remain, giving rise to the transition system displayed in figure 4.7. The processes A and B synchronise once every 6 units of time. The thick arrow indicates the continuum of states that are passed during the time transitions. Because of the determinism of time transitions, this continuum can in the closed interpretation be replaced by a single transition without considering any of the intermediate states.

4.4 Equivalences

To compare processes in the calculus with the implementation described in this chapter, we use bisimulation equivalence as introduced in section 2.3. We show that the implementation is 'almost bisimilar' to the calculus. In order to make this relation precise, we need some extra definitions that are introduced here.

4.4.1 Time-Closure

The calculus has an explicit rule *CNT* which makes the transition system closed with respect to time transitions in the sense stated by this rule, namely if $R_1 \xrightarrow{t_1} R_2$ and $R_2 \xrightarrow{t_2} R_3$ are transitions, then $R_1 \xrightarrow{t_1+t_2} R_3$ is also a transition. If a transition system is not closed with respect to time transitions, we can define its corresponding *time-closure* as follows. The time-closure of the timed transition relation $\xrightarrow[T^+]{}$ of a TLTS $(S, T, L, \xrightarrow[L]{}, \xrightarrow[T^+]{})$, denoted $\xrightarrow[T^+]{}^{\bullet}$, is the smallest relation $\stackrel{}{\Longrightarrow}_{T^+} \subseteq S \times T^+ \times S$ such that

- $\xrightarrow[T^+]{} \subseteq \stackrel{}{\Longrightarrow}_{T^+}$ and
- if $s_1 \stackrel{t_1}{\Longrightarrow}_{T^+} s_2$ and $s_2 \stackrel{t_2}{\Longrightarrow}_{T^+} s_3$ then $s_1 \stackrel{t_1+t_2}{\Longrightarrow}_{T^+} s_3$.

If $(S, T, L, \xrightarrow[L]{}, \xrightarrow[T^+]{})$ is a TLTS, then we use $(S, T, L, \xrightarrow[L]{}, \xrightarrow[T^+]{})^{\bullet}$ to denote the TLTS with the time closure of its timed transition relation, $(S, L, T, \xrightarrow[L]{}, \xrightarrow[T^+]{}^{\bullet})$.
We shall see in the next section that although the implementation of the calculus is not timed bisimilar to the calculus, its time-closure is.

4.4.2 Abstract bisimulations

In chapter 2 we discussed weak bisimulation as an equivalence which is similar to strong bisimulation, but abstracts from internal actions. In this chapter we have decorated actions of a transition system with extra information to form requests. To reason about the relationship between the decorated transition system (the implementation) and the original one, we need a notion of equivalence that abstracts from such decorations[81].
An abstraction function on a set of labels L is some function f with $dom(f) = L$. We define a notion of strong bisimulation equivalence with respect to abstracted actions produced by f.

4.4.1 DEFINITION *Let f be an abstraction function on L. A symmetric relation $\mathcal{S} \subseteq S \times S$ is a* strong timed bisimulation w.r.t. observability f *if for any $(s_1, s_2) \in \mathcal{S}$, $\lambda \in L$ and $t \in T^+$:*

- *if $s_1 \xrightarrow{\lambda} s_1'$, then for some s_2' and λ', $f(\lambda) = f(\lambda')$, $s_2 \xrightarrow{\lambda'} s_2'$ and $(s_1', s_2') \in \mathcal{S}$;*
- *if $s_1 \xrightarrow{t} s_1'$, there is some s_2' such that $s_2 \xrightarrow{t} s_2'$ and $(s_1', s_2') \in \mathcal{S}$.* □

States s_1 and s_2 are called strongly equivalent w.r.t. observability f (denoted as $s_1 \sim_f s_2$) if and only if there is a strong bisimulation $\mathcal{S}$ w.r.t. observability f, such that $(s_1, s_2) \in \mathcal{S}$. s_1 and s_2 are strongly equivalent if and only if s_1 and s_2 are strongly equivalent without abstracting information from the actions, i.e. if they are strongly equivalent w.r.t. the identity function Id_L. Naturally, greater abstraction leads to weaker equivalences.

4.4.2 DEFINITION *Let f and g be abstraction functions, we say that f is less abstract than g ($f \leq g$) if for all $\lambda_1, \lambda_2 \in L$, $f(\lambda_1) = f(\lambda_2) \Rightarrow g(\lambda_1) = g(\lambda_2)$.* □

Then the following is easy to verify.

4.4.3 LEMMA *Let f and g be abstraction functions on L, then $f \leq g \Rightarrow \sim_f \subseteq \sim_g$.*

4.5 Correctness of the Execution Method

In this section, it is shown that one can execute a process of the calculus using the requests and *Step* functions, in accordance with its semantics. First, we define a labelled transition system induced by the requests and the *Step* function introduced. Then, a relationship is established between this implementation LTS and the semantics of the calculus.

4.5.1 DEFINITION *Define a new labelled transition system called Impl as follows.*

$$Impl := \left(\mathcal{SP}, AR, T, \xrightarrow[AR]{}_{Impl}, \xrightarrow[T^+]{}_{Impl} \right).$$

The transition relations are defined in terms of AReq, TReq and Step.

- $R \xrightarrow[AR]{r}_{Impl} R'$ *iff* $r \in AReq(R)$ *and* $R' = Step(R,r)$;
- $R \xrightarrow[T^+]{t}_{Impl} R'$ *iff* $t \in TReq(R)$ *and* $R' = Step(R,t)$.

Then *Impl* is almost bisimilar to the original calculus with respect to the abstraction *ACT*. In particular the time-closure is bisimilar to the original calculus. This shows that an implementation according to the modified transition system *Impl* is correct. Every transition of the implementation corresponds to a transition of the calculus end every transition of the calculus can be mimicked by the implementation. Only certain time transitions are made in a (finite) number of smaller steps. Note that it follows from the definition, that the implementation *Impl* is deterministic, every request corresponds to at most one transition of a process R.

4.5.2 LEMMA *For every* $R \in \mathcal{SP}$, $r \in AReq(R)$,

$$R \xrightarrow{ACT(r)} Step(R,r).$$

PROOF By transition induction. Let $R = R_1||R_2$ (the other cases follow similarly). Then r is of one of the following forms: $L \cdot r'$, $R \cdot r'$ or $\tau \cdot (r_1, r_2)$.

- If $r = L \cdot r'$, then $r' \in AReq(R_1)$, thus by induction, $R_1 \xrightarrow{ACT(r')} Step(R_1, r')$. $ACT(r) = ACT(r')$ and by semantic rule PAR_1, $R \xrightarrow{ACT(r)} Step(R_1, r')||R_2$. Further,

$$\begin{array}{rl} & Step(R_1,r')||R_2 \\ = & \surd(Grant(R_1,r'))||R_2 \\ = & \surd(Grant(R_1,r')||R_2) \\ = & \surd(Grant(R_1||R_2,r)) \\ = & Step(R,r). \end{array}$$

- If $r = R \cdot r'$, then the proof is symmetrical to the previous case.
- If $r = \tau \cdot (r_1, r_2)$, then $r_1 \in AReq(R_1)$, $r_2 \in AReq(R_2)$ and $ACT(r_1) = \overline{ACT(r_2)}$. By induction, $R_1 \xrightarrow{ACT(r_1)} Step(R_1, r_1)$ and $R_2 \xrightarrow{ACT(r_2)} Step(R_2, r_2)$. It follows by semantical rule PAR_2 that $R \xrightarrow{ACT(r)} Step(R_1, r_1)||Step(R_2, r_2)$ (Notice that $ACT(r) = \tau$). Further,

$$\begin{array}{rl} & Step(R_1,r_1)||Step(R_2,r_2) \\ = & \surd(Grant(R_1,r_1))||\surd(Grant(R_2,r_2)) \\ = & \surd(Grant(R_1,r_1)||Grant(R_2,r_2)) \\ = & \surd(Grant(R_1||R_2,r)) \\ = & Step(R,r). \end{array}$$

4.5.3 LEMMA *Let* $R_1, R_2 \in \mathcal{SP}$ *and* $\alpha \in Act$ *such that* $R_1 \xrightarrow{\alpha} R_2$. *Then there is some* $r \in AReq(R_1)$ *such that* $\alpha = ACT(r)$ *and* $R_2 = Step(R_1, r)$.

PROOF By transition induction. We shall only prove the case where $R_1 = R_1'[f]$. Then there is some $\alpha' \in Act$, such that $\alpha = f(\alpha')$, $R_1' \xrightarrow{\alpha'} R_2'$ and $R_2 = R_2'[f]$. By induction, there is some

$r' \in AReq(R_1')$ such that $ACT(r') = \alpha'$ and $R_2' = Step(R_1', r')$. By definition of $AReq$, there is a request $r = f(ACT(r')) \cdot r' = f(\alpha') \cdot r' = \alpha \cdot r'$ in $AReq(R_1)$. Now $ACT(r) = \alpha$ and

$$\begin{array}{ll} & Step(R_1, r) \\ = & \surd(Grant(R_1, r)) \\ = & \surd(Grant(R_1'[f], \alpha \cdot r')) \\ = & \surd(Grant(R_1', r')[f]) \\ = & \surd(Grant(R_1', r'))[f] \\ = & Step(R_1', r')[f] \\ = & R_2'[f] \\ = & R_2. \end{array}$$

□

We now state similar lemmas for time requests. First, we show that every time step in the implementation is a valid time step in the calculus.

4.5.4 LEMMA *For every $R \in \mathcal{SP}$ and $t \in TReq(R)$,*

$$R \xrightarrow{t} Step(R, t).$$

PROOF By transition induction. Let $R = S_1 + S_2$ (the other cases follow similarly). Then $t \in TReq(S_1)$ and $t \in TReq(S_2)$. Thus by induction, $S_1 \xrightarrow{t} Step(S_1, t)$ and $S_2 \xrightarrow{t} Step(S_2, t)$. Then, according to semantical rule $+_2$, $R \xrightarrow{t} \surd(Step(S_1, t) + Step(S_1, t))$. However,

$$\begin{array}{ll} & \surd(Step(S_1, t) + Step(S_2, t)) \\ = & \surd(\surd(Grant(S_1, t)) + \surd(Grant(S_2, t))) \\ = & \surd(Grant(S_1, t) + Grant(S_2, t)) \\ = & \surd(Grant(S_1 + S_2, t)) \\ = & Step(R, t). \end{array}$$

□

The final part we need to prove is that the implementation can mimic the time transitions of the calculus. We have seen however that there are time transitions in the calculus, that are not present in the implementation, since the implementation does not have the equivalent of the *CNT* rule. We will show, however, that any time transition of the process can be matched by one or more time transitions in the implementation. To show this we need the following lemma which states that time steps in the implementation can be broken up into smaller ones.

4.5.5 LEMMA *Let $R \in \mathcal{SP}$ and $t \in TReq(R)$. Then for any $t_1, t_2 \in T^+$ such that $t_1 + t_2 = t$,*

$$Step(Step(R, t_1), t_2) = Step(R, t)$$

PROOF Again, by transition induction. We illustrate the proof using the basic case that $R = \langle t' \rangle$. Then $t_1 < t \leq t'$. $Step(R, t_1) = \langle t' - t_1 \rangle$ and $Step(\langle t' - t_1 \rangle, t_2) = \surd(\langle t' - t_1 - t_2 \rangle) = \surd(\langle t' - t \rangle) = Step(R, t)$. □

Thus if $R \xrightarrow{t}_{Impl} R'$ and $t_1 + t_2 = t$ then there is some R'' such that $R \xrightarrow{t_1}_{Impl} R''$ $\xrightarrow{t_2}_{Impl} R'$. Next, we define what it means for a sequence of delays to be an appropriate sequence for the implementation to match a time transition in the calculus.

4.5.6 DEFINITION *Let $R, R' \in \mathcal{SP}$ and $t \in T^+$ such that $R \xrightarrow{t} R'$. Then the sequence $t_1, t_2, \ldots t_n$, such that $\sum_{k=1}^{n} t_k = t$, is called an* implementation sequence *for the transition $R \xrightarrow{t} R'$ if there exist $R_1, R_2, \ldots R_{n+1} \in \mathcal{SP}$ such that $R = R_1$, $R' = R_{n+1}$ and*

$$R_1 \xrightarrow{t_1}_{Impl} R_2 \xrightarrow{t_2}_{Impl} \ldots R_n \xrightarrow{t_n}_{Impl} R_{n+1}.$$

4.5.7 LEMMA *Let $R, R' \in \mathcal{SP}$ and $t \in T^+$ such that $R \xrightarrow{t} R'$, then there exists an implementation sequence $t_1, t_2, \ldots t_n$ for $R \xrightarrow{t} R'$.*

PROOF By induction on the derivation of $R \xrightarrow{t} R'$. With every application of a derivation taking two processes with the same time transition, a new sequence can be constructed from the 'interleaving' of both corresponding sequences (using lemma 4.5.5). If the rule uses only a single subprocess, the same sequence applies. If the *CNT* rule is applied, the corresponding sequences can be composed one after another. As these sequences remain finite, the lemma follows. We show the cases concerning the rules $+_2$, *CNT* and *SEQ*.

- If the applied rule is $+_2$, then there exist S_1, S_1', S_2 and S_2' such that $R = S_1 + S_2$, $R' = S_1' + S_2'$, $S_1 \xrightarrow{t} S_1'$ and $S_2 \xrightarrow{t} S_2'$. By induction, there are two implementation sequences, both totalling a delay t, for $S_1 \xrightarrow{t} S_1'$ and $S_2 \xrightarrow{t} S_2'$ respectively. By lemma 4.5.5, every delay in such a sequence can (if necessary) be split into two arbitrary parts, in order to obtain a (finite) implementation sequence for both $S_1 \xrightarrow{t} S_1'$ and $S_2 \xrightarrow{t} S_2'$ at the same time. Now it is easy to show that this sequence is an implementation sequence for $R \xrightarrow{t} R'$ as well.
- If the applied rule is *CNT* then $R \xrightarrow{t_1} R''$ and $R'' \xrightarrow{t_2} R'$ for some R''. By the induction hypothesis there exist two implementation sequences, the first $t_{1,1}, t_{1,2}, \ldots, t_{1,k}$ totalling a delay of t_1 and the second, $t_{2,1}, t_{2,2}, \ldots, t_{2,m}$ totalling a delay of t_2. Then the sequence $t_{1,1}, t_{1,2}, \ldots, t_{1,k}, t_{2,1}, t_{2,2}, \ldots, t_{2,m}$ totalling a delay of $t_1 + t_2$ is an implementation sequence for $R \xrightarrow{t} R'$.
- If the applied rule is *SEQ* then there exist S_1, S_1' and S_2 such that $R = S_1; S_2$, $R' = \surd(S_1'; S_2)$ and $S_1 \xrightarrow{t} S_1'$. By the induction hypothesis, there exists an implementation sequence $t_1, t_2, \ldots, t_n$ for $S_1 \xrightarrow{t} S_1'$. For every $k < n$, $t' = \sum_{i=1}^{k} t_i$ and S_1'' such that $S_1 \xrightarrow{t'} S_1''$, $S_1'' \neq \surd$. Thus, it can be shown that this is also an implementation sequence for $R \xrightarrow{t} R'$ by using the *SEQ* rule n times. □

The proof shows that the transitions in the TLTS *Impl* correspond exactly to the transitions which can be derived by the semantical rules without using the rule *CNT*. The lemmas 4.5.2, 4.5.3, 4.5.4 and 4.5.7 together show that *Impl* is a faithful implementation of *StatCalc* in the following sense. If we take the time-closure of *Impl* and abstract from request decorations using *ACT* then *Impl* is equivalent to *StatCalc*.

4.5.8 THEOREM $Impl^{\bullet} \sim_{ACT} StatCalc$

PROOF The appropriate bisimulation relation is $Id_{\mathcal{SP}} = \{(R, R) \mid R \in \mathcal{SP}\}$.

- If $R \xrightarrow[AR]{r}_{Impl} R'$ then $R' = Step(R, r)$ and by lemma 4.5.2, $R \xrightarrow[Act]{ACT(r)} R'$.

- If $R \xrightarrow[Act]{\alpha} R'$ then by lemma 4.5.3, there is some $r \in Req(R)$ such that $ACT(r) = \alpha$ and $Step(R, r) = R'$. Thus $R \xrightarrow[AR]{r}_{Impl} R'$ and $ACT(r) = \alpha$.
- If $R \xrightarrow[T^+]{t} R'$ then by lemma 4.5.7, there exists a finite sequence of transitions $R \xrightarrow[T^+]{t_1}_{Impl} R_2 \xrightarrow[T^+]{t_2}_{Impl} R_3 \ldots \xrightarrow[T^+]{t_n}_{Impl} R'$ such that $t_1 + t_2 + \ldots + t_n = t$. Hence $R(\xrightarrow[T^+]{t}_{Impl})^{\bullet} R'$.
- If $R(\xrightarrow[T^+]{t}_{Impl})^{\bullet} R'$ then there is a sequence $R \xrightarrow[T^+]{t_1}_{Impl} R_2 \xrightarrow[T^+]{t_2}_{Impl} R_3 \ldots \xrightarrow[T^+]{t_n}_{Impl} R'$ of transitions such that $t_1 + t_2 + \ldots + t_n = t$. By lemma 4.5.4, $R \xrightarrow[T^+]{t_1} R_2 \xrightarrow[T^+]{t_2} R_3 \ldots \xrightarrow[T^+]{t_n} R'$ and by time closure of *StatCalc*, $R \xrightarrow[T^+]{t} R'$. □

4.6 Software Tools

The execution framework of this chapter has been used in a simulation tool for the object-oriented modelling language POOSL (Parallel Object-Oriented Specification Language) [87, 88]. POOSL is a language for describing concurrent real-time systems. It combines concurrent processes in the style of the calculus as presented in chapter 3 and on data objects similar to objects in traditional object-oriented programming languages such as Smalltalk[93] or Java[95] (for more information on POOSL and the specification and design methodology SHE in which it is embedded, see [156, 80]). This section briefly describes the tools and some issues that arise in the application of the method to POOSL.

4.6.1 The Tools

The implementation method developed in this chapter has been applied to the formal modelling language POOSL. A tool-set called *SHESim* [85, 88, 65] was built which supports entry of POOSL models allowing for a graphical specification of their static structure and hierarchy. It is furthermore capable of (interactively) simulating the execution of POOSL models. It has been successfully applied in several cases for specification, modelling and validation of industrial control systems, protocols, telecommunication networks and so forth [156, 88, 160, 66, 132, 170, 182]. Besides the SHESim tools, tools translating POOSL models to C++ have also been built, based upon the same principles [169, 29].
POOSL models are made up of three kinds of entities. Clusters represent the static structure of the communicating components and are used for hierarchical decomposition and refinement. Processes are the basic autonomous elements of a POOSL model and data objects are passive entities used to represent the data of a process. These types of entities are briefly discussed in the following paragraph. A concrete example of a POOSL model (of the APS protocol) can be found in appendix A.

Clusters

The calculus of this thesis and the language POOSL focus on systems with a static structure of collaborating components. In the calculus (and in the formal syntax of

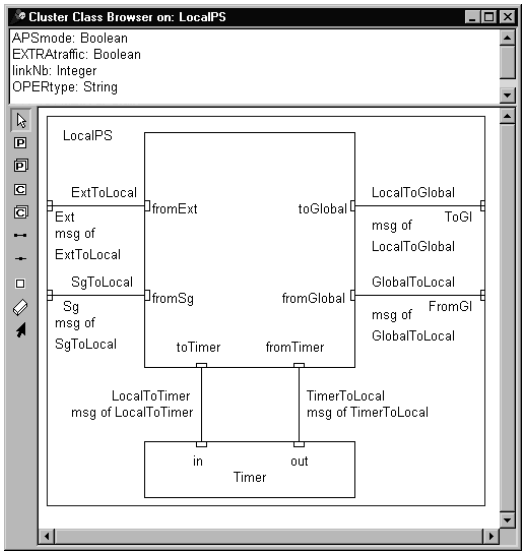


Figure 4.8: Specifying clusters in the SHESim tools

POOSL) this structure is specified using particular operators (e.g. parallel composition, relabelling and hiding). In practice it is easier to specify structure graphically, using a graphical syntax which can be translated into processes of POOSL. A structure in terms of clusters and processes connected by channels is translated into a process expression using appropriate hiding and relabelling operators. In POOSL, clusters are entities which are used as an abstraction of a structure of other clusters and processes. Figure 4.8 shows a window of the tool in which a cluster is being edited. Definitions of clusters are called cluster classes and correspond to our definitions of static process constants.

Process Objects

POOSL process objects are the basic components from which clusters are built. They correspond to the dynamic processes of the calculus in this thesis. Apart from operators corresponding to the ones of our calculus, POOSL supports some extra operators such as interrupts and aborts, which have been left out of the calculus for simplicity. Moreover, POOSL processes have access to data objects. They are specified in a textual language. In figure 4.9 it is shown how such a definition is entered in the tools. Such a definition is called a process class, since a single definition can be used for multiple instances of similar processes. The process class definitions correspond to definitions of dynamic process constants.

Data Objects

Next to clusters and the active process objects, the POOSL language also has passive data objects. These objects follow the concept of data objects found in many object-orientated programming languages such as C++[167], Smalltalk, [93] and Java[95].

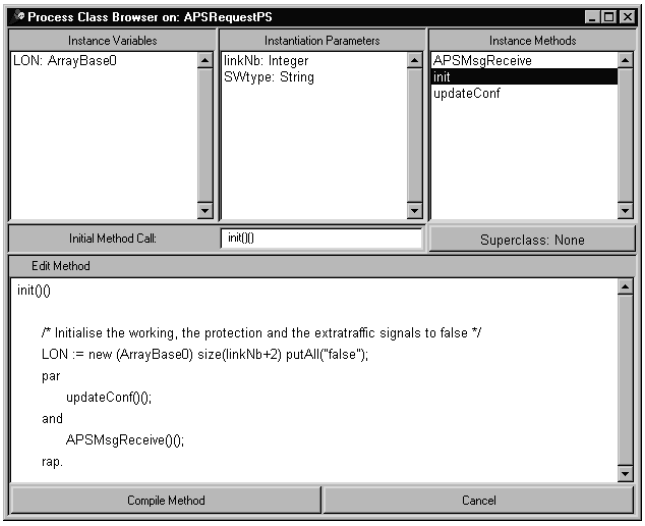


Figure 4.9: Specifying processes in the SHESim tools

Process objects employ data objects, communicate data objects to each other, but process objects do not share references to the same data object.

Simulation

Once a model has been specified completely, it can be simulated (see figure 4.10). As the scheduler traverses the transition system corresponding to the model, the behaviour can be observed in many ways, allowing the designer to interactively validate the behaviour of the system. Channels light up whenever a communication takes place. A process object can be inspected during simulation, exposing the values of its instance variables, the point(s) in the source code where it is currently executing or delaying. An interaction diagram (message sequence chart) can be displayed showing the communications that have taken place, the times at which they took place and the data that was passed. In the model of the APS protocol for instance, one can monitor, the messages that are exchanged between the APS nodes using an interaction diagram in order to validate the protocol's functional and timing requirements. It is also possible to have processes write data to files that can be used for off-line analysis.

4.6.2 Tool Implementation

In this section some issues are discussed that arise in a practical implementation of the entire language POOSL, but cannot be tied directly to the calculus.

Adding Data

POOSL incorporates data as well as communication and concurrency. The formal semantics of POOSL [156, 86] is defined in two layers. The interpretation of data

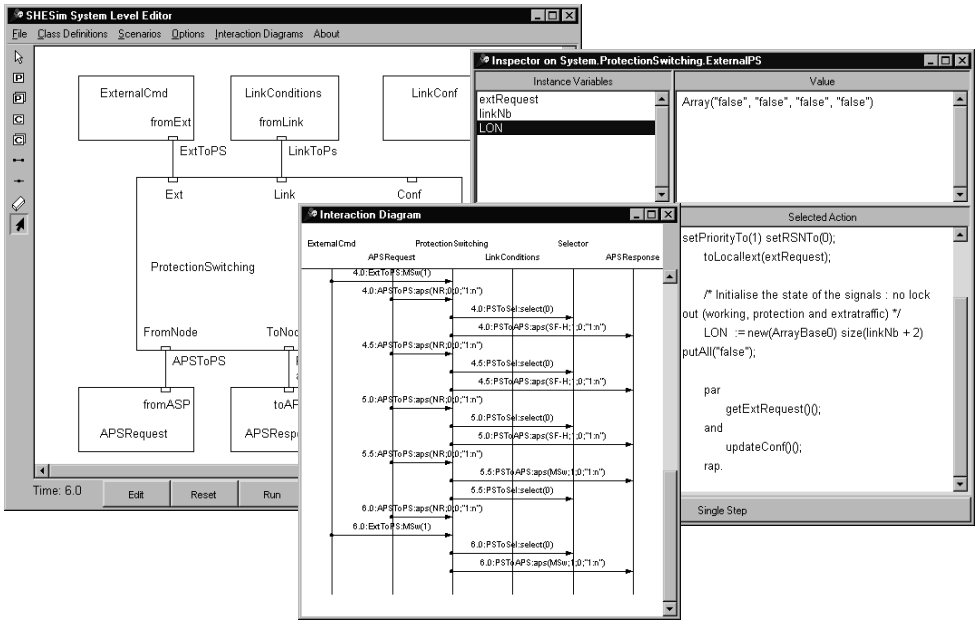


Figure 4.10: Simulating process behaviour in the SHESim tools

elements is defined by a computational semantics. The semantics is given as a transition system and the semantics $\mathcal{M}(E, \mathcal{E})$ of a data expression E in environment $\mathcal{E}$, is the set of all successfully terminated computations starting from E in the data environment $\mathcal{E}$. The process part of POOSL is similar to the calculus in this thesis, but employs conditions on data expressions in the premises of the semantic rules. There exists for instance a guarded command, $[E]S$, having the same transitions as process S, but only if the data expression E evaluates to the object *true.* The semantical rule of the guarded command looks like the following rule.

$$\frac{\mathit{true} \in \mathcal{M}(E, \mathcal{E}), (S, \mathcal{E}) \xrightarrow{\alpha} (S', \mathcal{E}')}{([E]S, \mathcal{E}) \xrightarrow{\alpha} (S', \mathcal{E}')}\ GRD$$

Since the data part of POOSL is Turing-complete, such data-dependent conditions render the transition relation undecidable. When trying to determine the possible transitions, such data expressions are evaluated. By the definition, the semantics of a data expression contains only the successfully terminated data computations, although there might also be non-terminating executions. Once an implementation decides to evaluate a data expression, it might end up on an infinite computation, resulting in diverging behaviour which is not present in the semantics. As this problem is related to the desired level of expressiveness of the language, reducing it to guarantee termination of data statements and data expression is impossible. To maintain the correspondence between the implementation and the formal semantics of the language, the possibility of divergence should be made a part of the semantics of data expressions and of the entire language. In fact, the same need to deal explic-

itly with non-terminating behaviour has been observed in ongoing efforts to create a probabilistic semantics for POOSL.
Another issue related to the implementation of data structures is storage reclamation. Data objects are created during execution and in POOSL there is no explicit way to destroy data when it is no longer needed. In a practical implementation, this means that a garbage collector is required to reclaim memory occupied by objects that are no longer used by the model. The SHESim tool is implemented in Smalltalk and inherits its garbage collection ability from the Smalltalk environment. An implementation in C++ requires garbage collection to be built-in explicitly [29].

Requests and Data

If the language includes data, it is not enough to consider a data transformation associated with the execution of a transition. The actual form or presence of a transition might depend on data in the process, such as in the case of the guarded command we have seen earlier. Other examples include message-send operations with data and message-receive operations in which the opportunity for synchronisation depends on the data being communicated.
In the case of shared data (see section 3.5), the data makes the transition relation non-compositional in the sense that if a process (sub) expression has a data-dependent request, then a transition which does not affect this expression, might still affect the request. In case of the guarded command for instance, a transition might modify the data environment such that the value of the guarding expression changes and thus the presence of one of its requests.
To execute such behaviour, the static (compositional) part of the semantics is separated from the data dependent (non-compositional) part. For the guarded command $[E]S$ for instance, a request is produced for every request of S. These request are labelled as being dependent on the data expression E. Then it can be globally determined (by valuating the expression) whether the request actually represents a transition or not. This way, granting a request outside of the subprocess $[E]S$ does not have an impact on the presence of the request itself, only possibly on the presence of an actual transition.

Binary Operators

For reasons of simplicity, the calculus employs associative binary operators for choice ($+$), interleaving ($|$), sequential ($;$) and parallel composition ($||$). These can be generalised to operators having an arbitrary number of subprocesses, resulting in a representation of smaller depth. This requires a straightforward adaptation to the mechanism of decorating requests.

$\mathbb{R}$ as a Time Domain

For theoretical reasons, the positive real numbers often form an attractive time-domain[23, 5]. In a physical implementation, such as the SHESim tool, concrete real numbers need to be replaced by finite approximations. This may introduce rounding errors which, depending on the model, may quickly produce large differences

in behaviour. If $[r]$ denotes the finite representation of the real r, and $\oplus$ the corresponding realisation of the $+$ operator, then it is not true that $[x] \oplus [y] = [x + y]$. Therefore, if two process synchronise and subsequently execute a number of delays that amount to the same total, then according to the semantics, they should end up in the same instant in time. In a realisation this may not be the case. If the two processes are in a race for a synchronisation with a third process, then the outcome may be determined by the rounding errors. POOSL allows processes to delay for an amount of time specified by an arbitrary real-valued expression, including the use of common mathematical operations on numbers. The current implementations of POOSL employ a straightforward replacement of real numbers with standard 64-bit IEEE floating point representations. As explained above, these can lead to small deviations in value from what is prescribed by the semantics. These deviations can be exploited to introduce deviations in the control flow as well.

Incremental Generation of Requests

Many requests are independent in the sense that granting one of them has no influence on the presence of the other. For instance two requests originating from different sides of a parallel composition are independent. Such transitions are called concurrent in [37] and may satisfy the 'diamond property' (after the diamond shape of a graphical representation of this fact), meaning that they can be taken in any order and the resulting process is the same (although in the presence of shared data, the order of execution can be of influence). If many requests in a system are independent it may be more efficient to produce requests incrementally, i.e. instead of recomputing all requests at every step, one only computes the new requests and removes requests that were dependent on the granted request.

Additional Operators

The POOSL language includes more operators than we have included in the calculus in this thesis. Powerful (dynamic) operators are the interrupt and abort operators. S_1 interrupt S_2 is a process which behaves as S_1 until an action of S_2 is performed. Then, the behaviour of S_1 is interrupted and the process continues to execute the remaining behaviour of S_2. Once this is finished, the behaviour of S_1 is resumed and can be interrupted again by S_2. The process terminates when the behaviour S_1 terminates. The S_1 abort S_2 operator behaves similarly, but once S_2 starts its behaviour, the behaviour of S_1 is aborted (and will thus never resumed). The process finishes when the behaviour of S_1 finishes or the behaviour of S_2 finishes. Both operators fit very well in the calculus and the implementation framework that is introduced.

Decoration of Requests

The use of decorated actions as requests is used to be able to trace the requests back to the subprocess from which it originated. In an actual implementation there are often easier ways to achieve this. First, rather than having every operator create new requests corresponding to requests of its subprocess, it might also pass on references to the same data structures. If subprocesses are represented using particular

data structures of the implementation language, a reference to the data structure is enough to replace the decorations.

4.7 Related Work

In this chapter we have introduced a method for the generation of the transition system corresponding to a model described in the calculus. This method can be used for automatic analysis of such models.

Execution of formal languages

Requests are introduced as decorated version of the labels of the calculus, to uniquely identify individual (derivations of) transitions. Such labelling of transitions occurs in different forms for different purposes. For instance, in [54], a transition is decorated with the 'address' from which the particular transition originates, in order to find the right parameter to be substituted for the data that is passed with a message. In [37], transitions are decorated with their derivation from the rules, in order to establish what transitions are dependent on each other and what transitions are truly 'concurrent'. In probabilistic calculi it is useful to be able to distinguish different derivations of the same transition in order to be able to add up probabilities [180]. Moreover we decorate our transitions even further in the next chapter to accommodate structural information in the transitions.
There exist several approaches to the execution of formal specifications such as CCS [140], LOTOS [71], SPADES [61] or μCRL [96]. Work on executing LOTOS/CCS specifications by van Eijk [69, 70] is most closely related to the work in this thesis. A simulator tool called Hippo is designed for the simulation of LOTOS specifications. The implementation is derived from the inference rules and provides a 'menu' function that gives all the available transitions and a function called 'next' that completes the move to a subsequent state when it is provided with the process and the selected transition. The data part of LOTOS consists of abstract data types (ADT's). The simulator uses a rewriting system for ADT's to implement this. [70] introduces a number of transformations on the semantics that make it more constructive, such as a labelling of the available transitions that makes them uniquely identifiable (similar to our requests) and the introduction of data environments instead of the traditional parameter substitution. [70] also suggests to reuse parts of the parse tree of a process for the representation of the process resulting from the transition. The approach taken in our simulator is to modify the original parse tree to represent the new process, rather than produce a new one. [41] shows another interpreter for LOTOS, based on a translation to CCS* (following the semantics of LOTOS, being defined in terms of CCS*, an extension of CCS), a CCS* interpreter and a rewriting rule interpreter for ADT's (both implemented in PROLOG). The LOTOS interpreters do not deal with time and the use of ADT's in LOTOS requires a non-constructive implementation of the data part. In POOSL, data is defined in an imperative style and therefore straightforwardly constructively implemented in e.g. C++.
The need for more constructive approaches towards data in process calculi is also answered by [54] which gives a redefinition of Milner's value passing calculus preventing the use of individual messages for every data value that could be received,

and [172] which introduces a calculus built from a timed version of CCS, operating in data environments, thus allowing shared data to be modelled. In [172], data is specified in VDM-SL. The calculus has provisions for backtracking changes in data environment. In a non-deterministic choice, alternatives may evolve into different data environments before a choice for one alternative has been made. Thus, the data environment must be split and if large, this may give a severe penalty in efficiency when implemented. An algebraic treatment of value-passing processes and equivalence relations can be found in [109].
In [97], the verification of specifications in μCRL is described. The construction of the system's state space is described from so-called linear process expressions, μCRL processes of a particular restricted form. Other μCRL processes are first transformed into linear process expressions. In our setting it is imperative to maintain the structure of the system in order to give appropriate feedback to the user about the system's state.
In [131] a 'virtual machine' implementation is described for TyCO (Typed Concurrent Objects) a variant of the π-calculus with objects built-in. The calculus does not describe timed systems. Tools exist that can automatically generate prototype implementations from transition-rule specifications of a language or calculus, such as ASF+SDF [26, 119]. Such systems are typically based on rewriting expressions. Although very efficient term rewriting engines have been implemented (such as ARM [77]), this approach is often not very efficient for languages that are not especially suited for this approach. Another example is [61], for a discrete event simulation generated from stochastic process algebraic specification (SPADES), implemented in Haskell. The model is based on (stochastic) timed automata. Timing is based on expiration of timers that are set according to some distribution. The model does not include data. χ is a language with accompanying tools for modelling production systems. For a part of the language, a formal semantics has been defined [33].

Simulation/execution of concurrent distributed models

There is a wide variety of languages and corresponding simulators for distributed concurrent systems. Some are informal, some have a formal definition of the underlying language, but not for the simulator implementation. Some are aimed at real-time systems, others at probabilistic systems and performance analysis. Some are designed for structured specification and dealing with complexity, others give precedence to efficiency and simulation speed. POOSL is designed for systematic modelling and specification of complex systems including time and complex data structures, based upon a mathematically defined model, allowing for the construction of tools that respect its formal interpretation.

4.8 Conclusions and Future Work

In this chapter a framework is introduced for the implementation of CCS-like formal languages, based on a Plotkin-style structural operational semantics. It is based upon the collection of so-called requests to represent the transitions available to a process. Every request uniquely determines a derivation of a transition. After the selection of such a request, the construction of the target process of the corresponding transition

can be performed along the lines of the *Step* function. It has been shown that this format leads to practically feasible and correct implementations. The approach has been applied to the industrial-strength modelling and specification language POOSL to develop an simulation tool that is in accordance with the formal semantics of the language. Implementation aspects are discussed as well as aspects relating to the language POOSL that are not reflected in the calculus of this thesis.
Future work includes the design of execution methods that enable more profound analyses of the model, for instance the generation of the model's entire transition system or an abstract version thereof. Another issue is the efficiency of the implementation. Analysis often requires extensive simulations if an exhaustive verification is infeasible. In particular, simulations for performance analysis can be very extensive. Therefore, efficiency of an implementation is of great importance. A simulator program might in certain circumstances serve as an (automatically generated) implementation. However, more insight is needed in the relationship between such an implementation and the formal model. In particular timeless execution of actions in the model cannot be physically realised. Another issue for further study is the construction of distributed simulators from POOSL models. Such a distributed implementation might allow for faster analysis, and the automatic generation of distributed implementations.

Chapter 5

Structure and Behaviour of Components

Models of complex distributed systems can be specified and can be executed according to their formal semantics by construction of the corresponding labelled transition system. Possibly, this construction is done only partially and / or on-the-fly. This can be used to validate that the specification captures the intended behaviour or to verify that certain forms of behaviour will be exhibited by the specification under all circumstances. Validation is effectively performed by simulation of the behaviour in a tool which gives the user a maximum of assistance in observing all of the system's aspects and managing its complexity. Verification on the other hand can be largely automated. Verification tools analyse a system for properties explicitly stated in a form that can be understood by the program. This analysis is done with little or no aid from the user.

The type of semantics in terms of labelled transition systems as used in the previous chapters focusses on behaviour exposed in transitions of the system. This is also reflected in the equivalence relations such as different forms of bisimulation. This approach is particularly useful when studying open systems and their interactions with possible environments. We have already seen that as system specifications become less abstract, closed specifications including the system that is being designed as well as the environment in which it must operate become more useful.

In a closed system all external interactions are prohibited. The only actions that can be performed are internal or silent (τ) actions. Although the presence of the τ action can be attributed to a particular subexpression or to the synchronisation of two subexpressions, this information is lost in the derivation of the transition. Yet, in tools such as the simulation tools for POOSL, communications within components are visualised. This is also desirable, since it is precisely this behaviour that is being studied. In fact, the transition system introduced in the previous chapter to model the implementation already employs requests that carry information about the origin of the transition they represent.

Moreover, if we take a closed system and make certain statements about its behaviour, then we have to refer to the internals of the system. We have to pinpoint a particular subexpression which represents a particular component of the entire sys-

tem. The calculus allows the composition of subsystems, but it is not obvious how to refer to subcomponents to state their properties.
In this chapter we try to remedy this deficiency by decorating subcomponents with identifier. By decorating actions of the calculus, similarly to the requests, we are able to reconstruct the behaviour of any subcomponent from the behaviour of the total.
In order to deal with the inherit complexity of systems, one often employs hierarchical decomposition techniques and modularity. Components interact through simple and well defined interfaces and their behaviour should not depend on the internal structure of components they interact with or of their subcomponents. In order to scale up formal description techniques to larger systems, a need arises for the same concepts such as hierarchy, modularity and encapsulation. This becomes necessary in particular when using techniques that permit the analysis of large models. Such techniques include for instance non-exhaustive analysis such as verifying temporal logic properties during simulations (see e.g. [84]), powerful automatic abstraction techniques, or estimation of performance figures from random state-space traversals ([180]). Note that the calculus is not in the first place used as an algebraic tool, but instead to describe the semantics of modelling constructs for complex concurrent systems. Therefore, we neglect a number of questions that would be interesting from an algebraic point of view. Such issues are addressed in other work on related issues (e.g. [46, 38, 145, 136, 109]).
Sections 5.1 and 5.2 introduce syntax and semantics respectively of an extension to the calculus to solve the problems described above. How individual contributions of subcomponents can be deduced from the behaviour of the total is shown in section 5.3. In section 5.4 we turn to the APS protocol example. Section 5.5 discusses the use of the issues addressed in this chapter in the context of POOSL and illustrate the use of the component semantics in the simulation tool SHESim with the APS protocol.
In section 5.6 we will investigate how we can adapt linear temporal logic to be used effectively in combination with identified components. Finally, related work and conclusions are given in sections 5.7 and 5.8 respectively.

5.1 A Component Calculus

We start by extending the calculus with the option of assigning identification to subprocesses. In fact we will only allow names to be assigned to static subprocesses, because we would like the identified components to persist during the execution of the system. First, we introduce a set $\mathcal{CID}$ of component identifiers and let X, Y range over $\mathcal{CID}$. Then, the syntax of the static process expressions, extended with identification of subcomponents, $\mathcal{CP}$ (ranged over by Q), is defined by:

$$Q \quad ::= \quad S \quad | \quad Q_1 || Q_2 \quad | \quad Q[f] \quad | \quad Q \backslash L \quad | \quad X \quad | \quad [Q]_X \,.$$

Recall that S is a dynamic process as defined in section 3.1.1. The processes are the same as the static processes of section 3.1.2 with the exception of the processes X and $[Q]_X$. X is a component identifier and replaces the static process constants of section 3.1.2. Thus, it is assumed to be defined in a set of (non-recursive) identifier equations $\{X \stackrel{\text{def}}{=} Q_X \mid X \in \mathcal{CID}\}$. The equations describe a finite hierarchy of components. It is assumed that the component names are unique (unless they occur within different

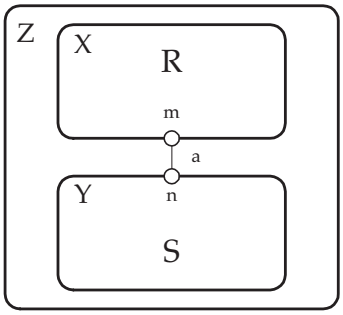


Figure 5.1: Process expression with identified components

components, allowing a process like $[Y]_X || Y$). In terms of the function *Components* which returns the components mentioned in a process expression Q (to be defined next), $\mathit{Components}(Q_1) \cap \mathit{Components}(Q_2) = \varnothing$ in the construct $Q_1 || Q_2$.
Note that one could have separated the concepts of static process constants and identifiers, as it is done in POOSL. There, process constants are called *process classes* and *cluster classes* and the instances of these classes are referred to by an identifier. For simplicity, we unify both concepts in our calculus. The process expression $[Q]_X$ represents a process identified as X with behaviour Q. A process X, defined by the equation $X \stackrel{\text{def}}{=} Q$ will be instantiated to $[Q]_X$. Derivatives of $[Q]_X$ retain the identifier.
For example the system depicted in figure 5.1 can be defined by the process expression Z with the following definitions

$$Z \stackrel{\text{def}}{=} (X[\mathrm{a}/\mathrm{m}] || Y[\mathrm{a}/\mathrm{n}]) \backslash \{\mathrm{a}\}$$
$$X \stackrel{\text{def}}{=} R$$
$$Y \stackrel{\text{def}}{=} S$$

R and S are assumed to be process expressions without further component identifiers. Components are depicted by rectangles having rounded corners and the identifiers are written in the upper-left corner. The process consists of a component Z, and within Z, the components X and Y. These subcomponents will be referred to as $Z.X$ and $Z.Y$ respectively. In general, $\mathcal{FCID}$ is the set of fully qualified component names and the smallest set such that $\mathcal{CID} \subseteq \mathcal{FCID}$ and for every $X \in \mathcal{CID}$ and $\chi \in \mathcal{FCID}$, $X.\chi \in \mathcal{FCID}$. If Q is a process expression, we can determine the set $\mathit{Components}(Q)$ of components of Q. The function *Components* is defined in table 5.1. For the example process expression, *Components* yields $\{Z, Z.X, Z.Y\}$.

5.2 Semantics of the Component Calculus

We have extended the syntax of processes to include identifiers for components. In this section we extend the semantics accordingly, with the objective to be able to deduce from any transition, what components were involved and what actions they contributed. In order to achieve this we first extend the set $\mathcal{L}$ of labels as follows.

Q	$Components(Q)$
S	$\varnothing$
$Q_1 \| \| Q_2$	$Components(Q_1) \cup Components(Q_2)$
X	$\{X\} \cup X.Components(Q')^a$ where $X \stackrel{\text{def}}{=} Q'$
$[Q']_X$	$\{X\} \cup X.Components(Q')$
$Q'[f]$	$Components(Q')$
$Q' \backslash L$	$Components(Q')$

[a] $X.C = \{X.\chi \mid \chi \in C\}$

Table 5.1: Determining the components of process Q

5.2.1 DEFINITION *Let $\mathcal{L}^e$ be the smallest set such that:*

- $\mathcal{L} \subseteq \mathcal{L}^e$;
- $\ell_{X,\lambda} \in \mathcal{L}^e$ *if* $\lambda \in \mathcal{L}^e$, $\ell \in \mathcal{L}$ *and* $X \in \mathcal{CID}$. □

We let λ range over $\mathcal{L}^e$. An extended label $\ell_{X,\lambda}$ is used to represent a label that is externally visible as ℓ and produced by component X in which it was characterised by the extended label λ.
Similarly, the silent transition τ of the calculus is also extended such that one can reconstruct the way the internal action was produced (for example as a synchronisation between two components). This gives rise to a set *Silent* of silent actions.

5.2.2 DEFINITION *The set Silent of silent actions, ranged over by θ, is the smallest set such that:*

- $\tau \in Silent$;
- $\tau(\lambda_1, \lambda_2) \in Silent$ *if* $\lambda_1, \lambda_2 \in \mathcal{L}^e$;
- $\tau_{X,\theta} \in Silent$ *if* $\theta \in Silent$ *and* $X \in \mathcal{CID}$. □

Let furthermore the set Act^e of extended actions be defined as $Act^e = \mathcal{L}^e \cup Silent$ and let β range over Act^e. We furthermore use ζ to range over $Act^e \cup T^+$. $\tau(\lambda_1, \lambda_2)$ denotes an internal synchronisation between the (complementary) actions λ_1 and λ_2; similar to the requests of the previous chapter, the individual actions constituting the synchronisation are remembered. Finally, $\tau_{X,\theta}$ is a silent action originating from a component named X in which it was θ.
Sometimes one would like to remove all decorations from an action and only consider its external form. The function *ACT* abstracts all decorations from an extended action.

5.2.3 DEFINITION *The function $ACT : Act^e \to Act$ is defined as follows*

$$ACT(\beta) = \begin{cases} \beta & \text{if } \beta \in Act \\ \ell & \text{if } \beta = \ell_{X,\lambda} \\ \tau & \text{if } \beta \in Silent \end{cases}$$

□

$$\frac{Q_1 \xrightarrow{\beta} Q_1'}{Q_1||Q_2 \xrightarrow{\beta} Q_1'||Q_2, \quad Q_2||Q_1 \xrightarrow{\beta} Q_2||Q_1'} \; PAR_1$$

$$\frac{Q_1 \xrightarrow{\lambda_1} Q_1', Q_2 \xrightarrow{\lambda_2} Q_2', ACT(\lambda_1) = \overline{ACT(\lambda_2)}}{Q_1||Q_2 \xrightarrow{\tau(\lambda_1,\lambda_2)} Q_1'||Q_2'} \; PAR_2$$

$$\frac{Q \xrightarrow{\beta} Q'}{Q[f] \xrightarrow{f^e(\beta)} Q'[f]} \; REL_1 \qquad \frac{Q \xrightarrow{\beta} Q', ACT(\beta) \notin L, \overline{ACT(\beta)} \notin L}{Q\backslash L \xrightarrow{\beta} Q'\backslash L} \; HID_1$$

$$f^e(\beta) = \begin{cases} \beta \text{ if } \beta \in Silent \\ f(\beta) \text{ if } \beta \in Act \\ f(\ell)_{X,\lambda} \text{ if } \beta = \ell_{X,\lambda} \end{cases}$$

Table 5.2: Modified rules of the static processes with component identifiers

The semantics of the calculus extended with component identifiers is now defined by the tuple

$$CompCalc := \left(\mathcal{CP}, Act^e, T, \xrightarrow[Act^e]{}, \xrightarrow[T^+]{}\right).$$

The semantic rules of dynamic processes are left unchanged, as well as the time transition rules of the static processes. The modified action transition rules for the existing static processes are displayed in table 5.2. When two processes synchronise, the individual actions are remembered. The relabelling is straightforwardly adapted to all extended actions. It relabels the action in front of the extended actions and leaves the rest unchanged. The semantics of the new syntactic constructs are shown in table 5.3. The first rule describes how an action produced by a component is decorated with the name of the component and the action itself to be remembered. The second rule states that time transitions remain unchanged and undecorated and the third is the appropriate replacement of the process constant rule of section 3.2, stating that a process X is equivalent to its defining expression Q enclosed by the brackets and name, identifying it as component X.
One can now easily verify that the component structure of a process Q is indeed static, it remains the same for any derivative process expression.

5.2.4 LEMMA *If* $Q \xrightarrow{\beta} Q'$ *then* $Components(Q) = Components(Q')$.

Similar to the transition system of requests, it can be shown that the added decoration does not change the behaviour in the following sense. We can abstract away the component identification from a process expression. In the sequel we show that if R is the process expression Q with all identification removed, then the transition system of R is strongly timed bisimilar to the transition system of Q with respect to

$$\frac{Q \xrightarrow{\beta} Q'}{[Q]_X \xrightarrow{ACT(\beta)_{X,\beta}} [Q']_X}\ CMP_1 \qquad \frac{Q \xrightarrow{t} Q'}{[Q]_X \xrightarrow{t} [Q']_X}\ CMP_2$$

$$\frac{[Q]_X \xrightarrow{\zeta} Q', X \stackrel{\text{def}}{=} Q}{X \xrightarrow{\zeta} Q'}\ CMP_3$$

Table 5.3: Semantics of the new component operators

Q	$Pure(Q)$
S	S
$Q_1 \| Q_2$	$Pure(Q_1) \| Pure(Q_2)$
$Q[f]$	$Pure(Q)[f]$
$Q \backslash L$	$Pure(Q) \backslash L$
X	$Pure(Q')$ where $X \stackrel{\text{def}}{=} Q'$
$[Q]_X$	$Pure(Q)$

Table 5.4: The abstraction *Pure*

observability *ACT*, i.e. $R \sim_{ACT} Q$. Along the lines of [38], one can define a function $Pure : \mathcal{CP} \to \mathcal{SP}$ that removes all identification from a component process and thus returns a pure process of the calculus of chapter 3 (in fact a process of this calculus, *StatCalc*, without any process constants).

5.2.5 DEFINITION *Let $Q \in \mathcal{CP}$ then $Pure(Q)$ is the pure StatCalc expression obtained by abstracting from component information. The precise definition of Pure is straightforward and given in table 5.4.* □

Now the following lemmas state the similarity between a process Q and $Pure(Q)$ as depicted in figure 5.2.

5.2.6 LEMMA *If $Pure(Q) \xrightarrow{\alpha} R'$ then there exist Q' and β, such that $Q \xrightarrow{\beta} Q'$, $R' = Pure(Q')$ and $\alpha = ACT(\beta)$.*

PROOF By structural induction on Q. We only consider the case that $Q = [Q_1]_X$. Then

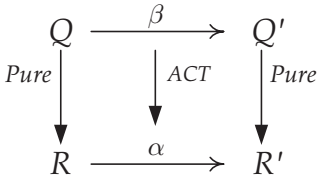


Figure 5.2: Relationship between *CompCalc* processes and their *Pure* counterparts

$Pure(Q) = Pure(Q_1) \xrightarrow{\alpha} R'$. By induction, there exist Q_1' and β' such that $Q_1 \xrightarrow{\beta'} Q_1'$, $Pure(Q_1') = R'$ and $\alpha = ACT(\beta')$. But then, $Q \xrightarrow{\alpha_{X,\beta'}} [Q_1']_X$, $R' = Pure([Q_1']_X)$ and $\alpha = ACT(\alpha_{X,\beta'})$. □

The similarity in the other direction is stated by the following lemma.

5.2.7 LEMMA *If* $Q \xrightarrow{\beta} Q'$ *then* $Pure(Q) \xrightarrow{ACT(\beta)} Pure(Q')$.

PROOF Straightforward by transition induction. For instance, if $Q = Q_1||Q_2$, then either of the following is the case.

- β stems from one of the subprocesses, say Q_1, $Q_1 \xrightarrow{\beta} Q_1'$ and $Q' = Q_1'||Q_2$. Then by induction, $Pure(Q_1) \xrightarrow{ACT(\beta)} Pure(Q_1')$ and thus $Pure(Q) = Pure(Q_1)||Pure(Q_2) \xrightarrow{ACT(\beta)} Pure(Q_1')||Pure(Q_2) = Pure(Q')$.
- $\beta = \tau(\lambda_1, \lambda_2)$, $Q_1 \xrightarrow{\lambda_1} Q_1'$, $Q_2 \xrightarrow{\lambda_2} Q_2'$, $ACT(\lambda_1) = \overline{ACT(\lambda_2)}$ and $Q' = Q_1'||Q_2'$. By induction, $Pure(Q_1) \xrightarrow{ACT(\lambda_1)} Pure(Q_1')$ and $Pure(Q_2) \xrightarrow{ACT(\lambda_2)} Pure(Q_2')$. Thus $Pure(Q) = Pure(Q_1)||Pure(Q_2) \xrightarrow{\tau} Pure(Q_1')||Pure(Q_2') = Pure(Q')$.

□

Similar lemmas for time transitions follow straightforwardly from the semantics; the component constructs do not add, remove or modify time transitions.

5.2.8 LEMMA *For any CompCalc process* Q, $Q \sim_{ACT} Pure(Q)$.

PROOF It can be shown using lemmas 5.2.6 and 5.2.7 (and the timed versions thereof) that $\mathcal{S} = \{(Q', Pure(Q')) \mid Q' \in \mathcal{CP}\}$ is a bisimulation with observability *ACT*. □

This means that if we abstract from the decorations, both on the process expressions and the actions, we have the same interpretation as the calculus of chapter 3.
Other evidence that the additions in the calculus are just decorations stems from the fact that equivalence with respect to observability *ACT* is still a congruence for all operators of the calculus, since all semantical rules 'ignore' the decorations when determining their possible transitions.

5.2.9 THEOREM *For any two CompCalc processes* Q_1 *and* Q_2,

$$Q_1 \sim_{ACT} Q_2 \Leftrightarrow Pure(Q_1) \sim Pure(Q_2)$$

PROOF Follows from the previous lemma and the fact that for pure processes, $\sim$ and $\sim_{ACT}$ coincide. □

The equivalences between component processes and their abstractions are represented in the following diagram.

$$\begin{array}{ccc} Q_1 & \sim_{ACT} & Q_2 \\ \sim_{ACT} & & \sim_{ACT} \\ Pure(Q_1) & \sim & Pure(Q_2) \end{array}$$

Q	$Q\downarrow_X$
S	$\surd$
$Q_1 \| \| Q_2$	$\begin{cases} Q_1\downarrow_X \text{ if } X \notin \mathit{Components}(Q_2) \\ Q_2\downarrow_X \text{ if } X \notin \mathit{Components}(Q_1) \end{cases}$
Y	$\begin{cases} Q \text{ if } Y = X \text{ and } X \stackrel{\text{def}}{=} Q \\ \surd \text{ if } X \neq Y \end{cases}$
$[Q]_Y$	$\begin{cases} Q \text{ if } X = Y \\ \surd \text{ if } X \neq Y \end{cases}$
$Q[f]$	$Q\downarrow_X$
$Q\backslash L$	$Q\downarrow_X$

Table 5.5: Component reduction

We have added decoration in both the syntax of process expression and the actions they perform, to be able to interpret processes and their transitions not just by their externally visible behaviour, but at the level of internal components as well. In the next section this decoration will be exploited to deduce information about internal components of a process.

5.3 Reduction through hierarchy

The extensions to syntax and semantics introduced in the previous section, allow the internal behaviour to be observed for all identified components. A trace now describes the behaviour of all identified components of the system together. We shall see in this section how the contributing trace of a subcomponent can be effectively regained from the execution trace of the entire system. A similar purpose serves the localisation operator in process algebra [22].

5.3.1 Reduction of process expressions

We first take a look at process expressions decorated with component identifiers. In a *CompCalc* process, one can 'find' a particular component, guided by the added decoration. A component X of a process expression Q is denoted by $Q\downarrow_X$. For instance in the example of section 5.1 (figure 5.1),

$$Z\downarrow_{Z.X} = R.$$

The definition of the reduction operation is given in table 5.5. Note that reduction to a non-existing component will (a bit arbitrarily) result in the process $\surd$. The operation is extended to full component names by letting $Q\downarrow_{X.Y} = (Q\downarrow_X)\downarrow_Y$.

5.3.2 Reduction of execution traces

We have seen how one can reduce a process expression to any of its constituting components. We shall now see how the same can be done for extended actions and for execution traces. The decorated actions of the component calculus record the

origin of the action. Reversely one can extract any component's contribution to the action. The reduction of an action to a component's contribution ($\cdot\downarrow_\cdot: Act^e \times \mathcal{CID} \to Act \cup \{\bot\}$) is done as follows. (A result $\bot$ is added to indicate that the component did not contribute to the action.)

5.3.1 DEFINITION *Let $\beta \in Act^e$, $X \in \mathcal{CID}$ then*

$$\beta\downarrow_X = \begin{cases} \beta' & \text{if } \beta = \alpha_{X,\beta'} \\ \bot & \text{if } \beta = \alpha_{Y,\beta'},\ Y \neq X \\ \bot & \text{if } \beta \in Act \\ \lambda_1\downarrow_X & \text{if } \beta = \tau(\lambda_1,\lambda_2) \text{ and } \lambda_2\downarrow_X = \bot \\ \lambda_2\downarrow_X & \text{if } \beta = \tau(\lambda_1,\lambda_2) \text{ and } \lambda_1\downarrow_X = \bot. \end{cases}$$

□

For convenience, we also define reduction on time transitions. Since time transitions are made synchronously by all components and are not decorated, $t\downarrow_X = t$. The reduction of actions and time transitions is also extended to full component names, $\zeta\downarrow_{X.Y} = (\zeta\downarrow_X)\downarrow_Y$.
Being able to determine components in a process expression and every component's contribution to an action, we are now able to define the reduction of a trace, $s\downarrow_X$. In chapter 2, we introduced a (timed) trace as an infinite sequence of transitions through some (timed) labelled transition system. For simplicity we ignored the possibility of finite paths through the LTS. If we take an timed trace that has infinitely many time transitions and look at the contribution to this trace of one of its components, then this will also be an infinite trace. It is possible that after some period of time, the component will execute no more action transitions, but it will still continue to make time transitions. In contrast, if we look at untimed traces, then the reduced version of an infinite trace may no longer be infinite.
The contributing trace of a component X can be determined by reducing all process expressions of the trace to X and skipping those transitions in which X did not take any part.

5.3.2 DEFINITION *Let s be a timed trace, $X \in \mathcal{CID}$ and $s = (\overline{Q}, \overline{\zeta})$ then the operator $\cdot\downarrow_\cdot$ is such that*

$$s\downarrow_X = \begin{cases} s^1\downarrow_X & \text{if } \overline{\zeta}(0)\downarrow_X = \bot \\ (\overline{Q}(0)\downarrow_X, \overline{\zeta}(0)\downarrow_X)(s^1\downarrow_X) & \text{otherwise.} \end{cases}$$

□

The transitions that do not involve X can be removed, since the component X cannot have changed if the transition did not involve X. One can show that if $\beta\downarrow_X = \bot$, then component X cannot have changed by any transition labelled β. This is expressed by the following lemma.

5.3.3 LEMMA *If $Q \xrightarrow{\beta} Q'$ and $\beta\downarrow_X = \bot$ then $Q\downarrow_X = Q'\downarrow_X$.*

PROOF This can be proved by induction on the derivation of the transition $Q \xrightarrow{\beta} Q'$. We will only consider $Q = Q_1||Q_2$. Then either of the following is the case:

- $\beta = \alpha_{Y,\beta'}$ and $Y \neq X$. Assume (without loss of generality) that $Q_1 \xrightarrow{\beta} Q_1'$ and $Q' = Q_1'||Q_2$. By induction, $Q_1 \downarrow_X = Q_1' \downarrow_X$. Thus $Q \downarrow_X = (Q_1||Q_2) \downarrow_X = (Q_1'||Q_2) \downarrow_X = Q' \downarrow_X$.
- $\beta \in Act$. Then $Q \downarrow_X = Q' \downarrow_X = \surd$.
- $\beta = \tau(\lambda_1, \lambda_2)$ and both $\lambda_1 \downarrow_X = \perp$ and $\lambda_2 \downarrow_X = \perp$. There are again two possibilities,
  - If $Q_1 \xrightarrow{\lambda_1} Q_1'$, $Q_2 \xrightarrow{\lambda_2} Q_2'$ and $Q' = Q_1'||Q_2'$ for some Q_1' and Q_2' then, by induction, $Q_1' \downarrow_X = Q_1 \downarrow_X$ and $Q_2' \downarrow_X = Q_2 \downarrow_X$. Thus, $Q' \downarrow_X = Q \downarrow_X$.
  - Otherwise, assume without loss of generality that $Q_1 \xrightarrow{\beta} Q_1'$ and $Q' = Q_1'||Q_2$. By induction, $Q_1' \downarrow_X = Q_1 \downarrow_X$ and thus, $Q' \downarrow_X = Q \downarrow_X$. □

If however, $\beta \downarrow_X \neq \perp$ then the transition is maintained and reduced to X. The fact that this yields a valid transition is shown next.

5.3.4 LEMMA *If* $Q \xrightarrow{\beta} Q'$ *and* $\beta \downarrow_X \neq \perp$ *then* $Q \downarrow_X \xrightarrow{\beta \downarrow_X} Q' \downarrow_X$.

PROOF By induction on the derivation of $Q \xrightarrow{\beta} Q'$. One of the following cases applies.

- $\beta = \alpha_{X,\beta'}$. Component X is some subprocess of Q. If for instance, $Q = Q_1[f]$ then $Q_1 \xrightarrow{\alpha'_{X,\beta'}} Q_1'$ for some α' and Q_1'. By induction, $Q_1 \downarrow_X \xrightarrow{\beta'} Q_1' \downarrow_X$ and thus $Q \downarrow_X \xrightarrow{\beta \downarrow_X} Q' \downarrow_X$.
- $\beta = \tau(\lambda_1, \lambda_2)$ and $\lambda_2 \downarrow_X = \perp$. Component X contributed to the communication in action λ_1 and the result follows by the previous case.
- $\beta = \tau(\lambda_1, \lambda_2)$ and $\lambda_1 \downarrow_X = \perp$. Similar to the previous case.

□

The previous lemmas (5.3.3 and 5.3.4) together demonstrate that the reduction is sound, and results in a trace.

5.3.5 THEOREM *If* s *is a timed trace with infinitely many time transitions and* $X \in \mathcal{CID}$, *then* $s \downarrow_X$ *is a trace (with infinitely many time transitions).*

PROOF The key is that one can show that if $s = (\overline{Q}, \overline{\zeta})\ \overline{Q}(n) \xrightarrow{\overline{\zeta}(n)} \overline{Q}(n+1)$ is a step of the trace s, then

- if $\overline{\zeta}(n) \downarrow_X = \perp$, then by lemma 5.3.3, $\overline{Q}(n+1) \downarrow_X = \overline{Q}(n) \downarrow_X$ and the transition can be safely removed;
- if $\overline{\zeta}(n) \downarrow_X \neq \perp$, then according to lemma 5.3.4 (and the timed version thereof), $\overline{Q}(n) \downarrow_X \xrightarrow{\overline{\zeta}(n) \downarrow_X} \overline{Q}(n+1) \downarrow_X$.

Moreover, the resulting trace is still infinite, since at least for all time transitions $\overline{\zeta}(k)$, $\overline{\zeta}(k) \downarrow_X \neq \perp$. □

The reduction operator on traces is again extended to full component names, letting $s \downarrow_{X.Y} = (s \downarrow_X) \downarrow_Y$. Figure 5.3 shows two steps of a trace of the example system of figure 5.1. The execution at the bottom shows the trace of the entire system performing two silent (internal) actions. The bottom execution in the box named Z shows the same execution as performed by component Z. Similarly the execution in the boxes X and Y display to contributing executions of the components $Z.X$ and $Z.Y$ respectively.

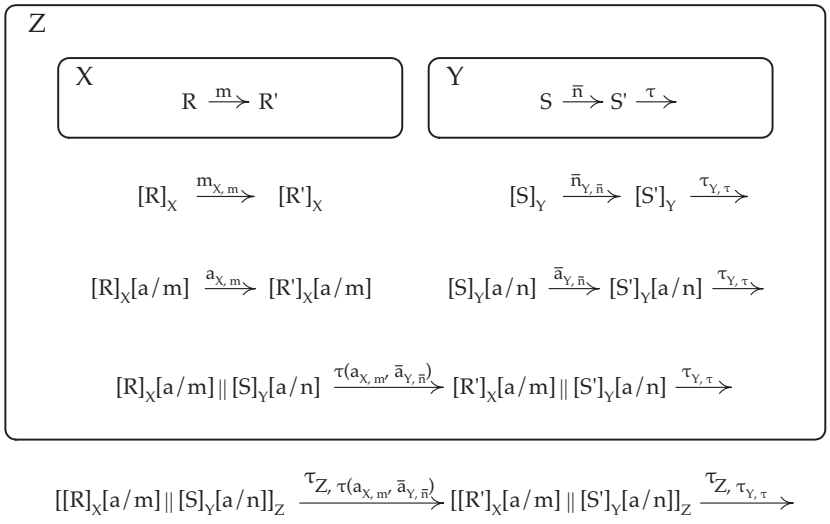


Figure 5.3: Reducing a trace to its components

5.4 Automatic Protection Switching Protocol (3)

As an example of the use of components and the added decorations, we revisit the model of the protection switching protocol of sections 2.7 and 3.6. The architecture of the APS model is depicted graphically in figures 5.4 and 5.5.
The static processes (*LocalProcess*, *ExternalProcess*, *GlobalProcess*, *APSMessages*, *NetworkManagement* and *Environment*) of the model of 3.6 are now viewed upon as components. From these components, one creates larger components, such as the generic APS node

$$APSNode \stackrel{\text{def}}{=} (LocalProcess||ExternalProcess||GlobalProcess||APSMessages)\setminus \{lop, locReq, extReq, glbReq, rmtReq\}.$$

the individual APS nodes

$$APSNode1 \stackrel{\text{def}}{=} APSNode[extComm1/extComm, aps1/sndAPS, aps2/recAPS, sgnCond1/sgnCond]$$

$$APSNode2 \stackrel{\text{def}}{=} APSNode[extComm2/extComm, aps2/sndAPS, aps1/recAPS, sgnCond2/sgnCond]$$

and the entire protocol by placing the two nodes in their environment.

$$APSProtocol \stackrel{\text{def}}{=} (NetworkManagement||APSNode1||APSNode2||Environment)\setminus \{extComm1, extComm2, aps1, aps2, sgnCond1, sgnCond2\}.$$

While we had modelled the same *APSProtocol* process as a closed system in the example of chapter 3, its semantics was a labelled transition system that could only perform internal (τ) actions and time steps. It would not reveal anything about the correct operation of the protocol. In terms of the semantics given in this chapter however, the contributing behaviour of all components can be observed.

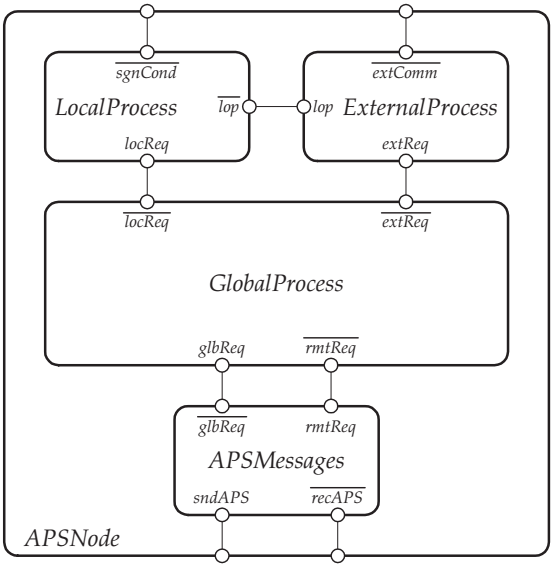


Figure 5.4: The architecture of an APS node

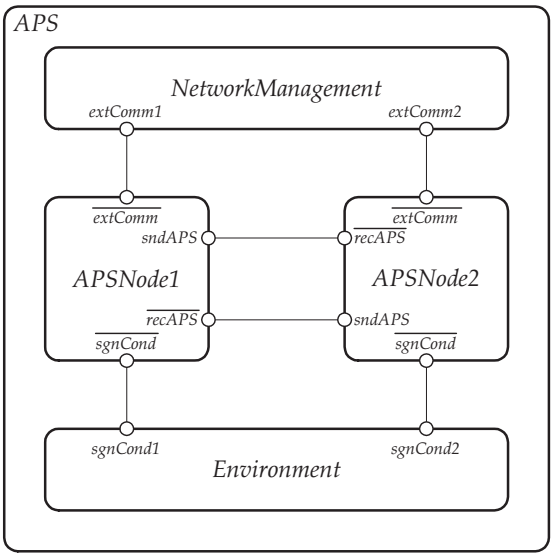


Figure 5.5: The architecture of the APS protocol

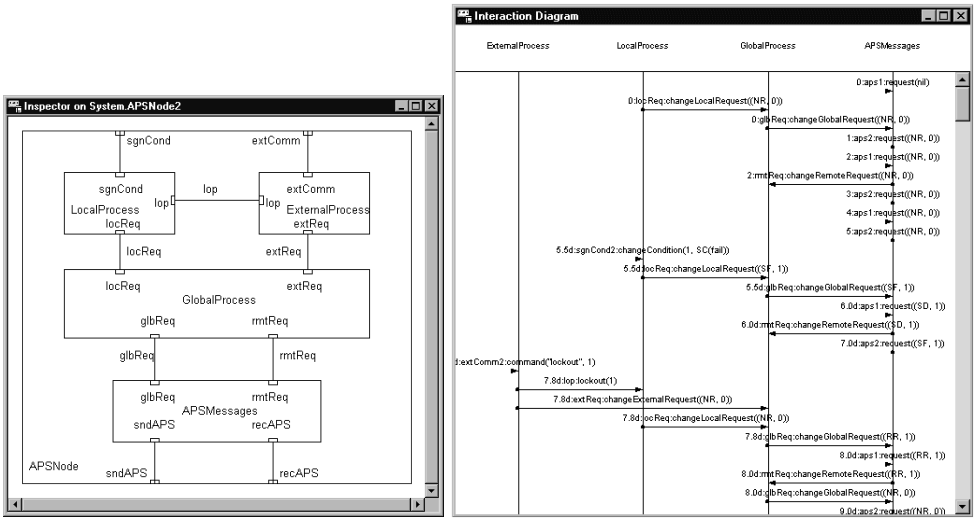


Figure 5.6: Inspecting components of an APS node and behaviour of the node in SHESim

5.5 Components in POOSL and SHESim

In POOSL the static part of process expressions is defined in terms of structural entities called *clusters* and behavioural entities called *processes* or *process objects*, connected through ports and channels. Process objects and clusters are instances of process classes and cluster classes that define their behaviour and structure respectively. As already mentioned in section 5.1, the concepts of identifiers and the process constants are separated in POOSL. The process constants correspond to POOSL class names and the component identifiers correspond to the names of the actual instances of such classes. Thus, instead of using two layers of components for the APS nodes, both *APSNode1* and *APSNode2* consisting of a component *APSNode*, we have two *instances* called *APSNode1* and *APSNode2* of *cluster class APSNode* (see the (textual) POOSL specification of the APS protocol in appendix A).
In the *APSNode* cluster depicted in figure 5.6, there are four process objects of classes *LocalProcess*, *ExternalProcess*, *GlobalProcess* and *APSMessages*, which are in this case also named *LocalProcess*, *ExternalProcess*, *GlobalProcess* and *APSMessages*. In the APS-protocol model depicted in figure 5.7, one can see the instances *APSNode1* and *APSNode2* of cluster class *APSNode*.
If we visualise the behaviour of a particular cluster during a simulation of a POOSL model, we are observing the local trace of this component. We observe its behaviour, but only behaviour that it displays in the environment in which it is embedded. Figure 5.6 shows a window viewing one of the APS nodes (embedded in the protocol and environment) and a window that displays the behaviour of the component in the form of an interaction diagram. The latter shows messages exchanged between its subcomponents and between the node and its environment. The SHESim simulation tool thus allows one to navigate the model's architecture and observe behaviour of an arbitrary selection of the model's components. Moreover, interaction diagrams on a component allow the choice to abstract from internal behaviour of subcomponents,

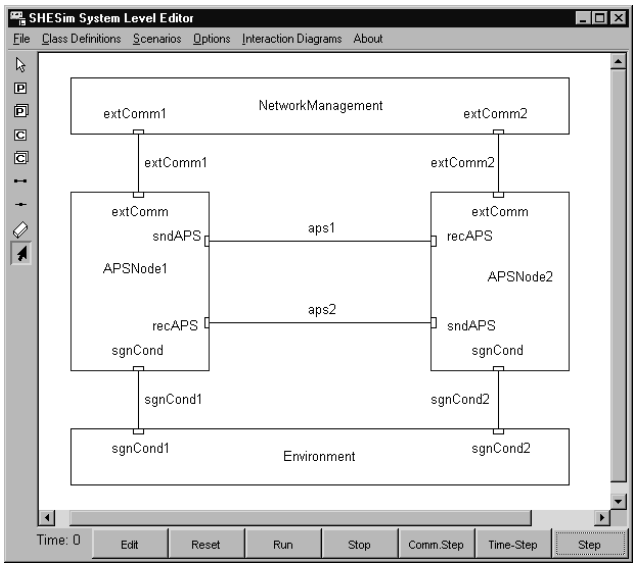


Figure 5.7: Components of the APS protocol in SHESim

or exposing it as well.

5.6 Temporal Logic for Components

Now that we can identify components and know how their local behaviour contributes to the entire system, we can express and evaluate properties of components. We shall show how properties expressed in linear temporal logic can be used to formulate properties of a component's behaviour.

5.6.1 States and Transitions in Temporal Logics

Linear temporal logic is usually interpreted over transition systems that have properties that can be observed in the states of the system. Other logics such as the modal μ-calculus([121]) are more tailored to labelled transition systems which expose their observable behaviour in transitions. The difference is not significant and one interpretation can be cast in the form of the other quite easily. In fact, in more complex modelling formalisms one would like to be able to refer to both communications (exposed in transitions) and state properties (for instance conditions on variables). One possibility is to encode this into pure state-based observations by letting states be pairs of process expressions and actions which represent the action observed in the transition to the new state. This is comparable to our traces which are sequences of such pairs. We will not further elaborate on this; more on this can be found in [146, 40]. We will not give a precise definition of a temporal logic to be interpreted on traces of both transitions and states. In the remainder we focus on temporal logics which traditionally use atomic propositions interpreted in states, but this restriction is not significant.

5.6.2 Extension of Linear Temporal Logic

To fit to the component structure one can extend linear temporal logic ([134]) with constructs that support such needs. We add a construct to target a formula to a particular component ($X.\varphi$) as well as property constants (ranged over by $\mathcal{P}$). Although property constants do not add any expressive power, they are added here since they are required for a practical property specification formalism to allow encapsulation of a component's internals. The syntax of such an extension of linear temporal logic is given by the following grammar.

$$\varphi ::= \text{true} \mid p \mid \neg\varphi' \mid \varphi_1 \vee \varphi_2 \mid \bigcirc \varphi' \mid \varphi_1 \mathsf{U} \varphi_2 \mid X.\varphi' \mid \mathcal{P}.$$

The formulas are interpreted over traces of the system as follows.

- $s \models \text{true}$ for every trace s;
- $s \models p$ iff $s(0) \models p$;
- $s \models \neg\varphi'$ iff not $s \models \varphi'$;
- $s \models \varphi_1 \vee \varphi_2$ iff $s \models \varphi_1$ or $s \models \varphi_2$;
- $s \models \bigcirc\varphi'$ iff $s^1 \models \varphi'$;
- $s \models \varphi_1 \mathsf{U} \varphi_2$ iff there is some $k \geq 0$, such that $s^k \models \varphi_2$ and for all $0 \leq m < k$, $s^m \models \varphi_1$;
- $s \models X.\varphi'$ iff $s \downarrow_X \models \varphi'$;
- $s \models \mathcal{P}$ iff $s \models \varphi$ where $\mathcal{P} \stackrel{\text{def}}{=} \varphi$.

p is an atomic proposition and is assumed to state some boolean property about the current state or about the current transition. We assume that p can be evaluated from $s(0)$. The operators true, negation ($\neg$), disjunction ($\vee$), next ($\bigcirc$) and until (U) are unchanged from LTL. The interpretation of the formula $X.\varphi$ is as follows. The formula φ is interpreted over the local trace of component X. Just as components can be defined using component constants, a property constant $\mathcal{P}$ ranging over the set $\mathcal{PID}$ of property constants can be used to define logical properties. We will use property constants to define properties by a set of (non-recursive) definitions $\{\mathcal{P} \stackrel{\text{def}}{=} \varphi_{\mathcal{P}} \mid \mathcal{P} \in \mathcal{PID}\}$.

The Next Operator

The 'next' operator ($\bigcirc$) is often considered to be problematic in the context of interleaving concurrency. The interleaving semantics may lead to a number of extra transitions which correspond to local transitions in other components. Since formulas using the $\bigcirc$ operator are sensitive to such transitions, formulas employing them may evaluate differently when a system is composed with other systems, even if these systems do not influence each other. Such external steps are said to cause 'stuttering', a repetition of (locally) identical states (see figure 5.8). The $\bigcirc$ operator is sensitive to stuttering transitions. To remedy this problem the next operator is

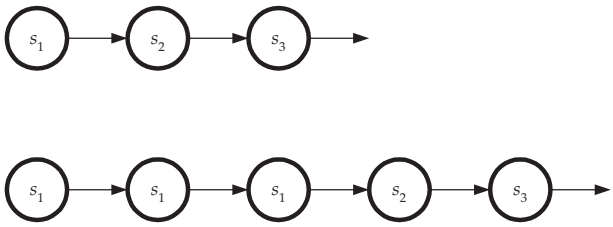


Figure 5.8: Two sequences of states, the second is obtained from the first by 'stuttering' of s_1

often removed (see [124]) and the remaining properties are then insensitive to (finite) stuttering. Others prefer to give the operator a different interpretation, such as letting it refer to the first subsequent state where any of its observable properties have changed [142, 56]. The disadvantage of the latter approach is that if the transition originates from the component itself but does not change any of the observable properties, it is skipped as well.
We can express temporal logic formulas over local traces, thus the next operator does not suffer from the stuttering problem[1]. If we specify a property $X.\bigcirc\phi$, then the $\bigcirc$ operator refers to the local next state, all transitions external to X are removed by the reduction of the trace.

5.6.3 Example

Some examples of properties one might want to express about the components of the APS protocol are discussed in this section (none of which are claimed to actually hold for the APS model). The following property of an APS node can for instance be expressed:

$$\mathit{LocalProcess}.(\Box \mathit{act(sigCond(SF\text{-}1))} \Rightarrow \bigcirc(\mathit{request.priority} \succeq \mathit{priority\_SF\text{-}1}))$$

(assuming the possibility to convey that a signal condition message indicating a signal failure on line 1 has just been received by the expression *act(sigCond(SF-1))* and that the priority of the currently dominating local request is at least equal to that of a signal failure on line 1 by the expression $\mathit{request.priority} \succeq \mathit{priority\_SF\text{-}1}$. The property then says that immediately after a new signal condition is received, the local request is updated to reflect the new situation.
The following property (partly) states the fact that the APS protocol messages alternate. Whenever APS node 1 communicates a *sndAPS* message, then after that it will not send any *sndAPS* messages until the other node does.

$$\Box(\mathit{APSNode1.act(sndAPS)} \Rightarrow \bigcirc((\neg \mathit{APSNode1.act(sndAPS)})\mathsf{U}(\mathit{APSNode2.act(sndAPS)})))$$

Again, using $\mathit{act}(\alpha)$ to express that a communication α has just occurred.

[1] There are other objections to the use of the $\bigcirc$ operator that are not remedied in our approach. It refers to the state reached after the execution of some atomic action and thus does not support the refinement of actions [124].

The following formula states that at any time, at least one of the APS nodes' global request is RR (indicating an acknowledgement that the request of the other node is of higher priority than its own).

$$\Box(\mathit{APSNode1.globRequestRR} \vee \mathit{APSNode2.globRequestRR})$$

The property makes use of the following definitions of property constants.

$$\mathit{globRequestRR} \stackrel{\text{def}}{=} \mathit{GlobalProcess.requestRR}$$

$$\mathit{requestRR} \stackrel{\text{def}}{=} \mathit{request.type==RR}.$$

Note that properties *globRequestRR* and *requestRR* of the APS node and global process are being used without having to know how these conditions can be expressed in terms of their internal implementation or structure.
To effectively apply such a logic in the context of POOSL, it should have certain first-order capabilities, that can be based on the data part of the language. Atomic propositions can be expressed in terms of data expressions yielding a boolean result; such expressions may refer to variables, call (side-effect free) data methods, and so forth. Moreover, if property constants are extended to express not only boolean properties or temporal formulas, but also data objects, then the property above can more easily be expressed as follows:

$$\Box(\mathit{APSNode1.request} = \mathit{RR} \vee \mathit{APSNode2.request} = \mathit{RR}).$$

5.7 Related Work

5.7.1 Calculi with Locations and Components

Process calculi focus on the behaviour of concurrent systems, rather than on their structure and architecture. There have been several approaches to make structure and distribution more explicit in process calculi. This is supported by the idea that it can be convenient to say that communicating with a process reveals its internal distributed structure. Most calculi assume that it is the distributed nature that is observed and not the actual (names of) components or their locations. Communication with the system may reveal that certain actions originate from different locations, but not from what particular location. To achieve this type of observational power, location names are assigned dynamically while interacting with the system (e.g., [38]). Other approaches statically assign locations giving them names that are exposed in interactions (e.g., [47, 145]).
A corresponding notion of distributed bisimulation was introduced by Castellani in [46]. A good overview on distributed process algebra and distributed bisimulations can be found in [47]. In [38] a distributed semantics for CCS was given using dynamic locations. A corresponding equivalence relation called *location equivalence* was given that equates two processes only if they have the same distribution structure. In [47], based on [1], Castellani proposes an extension of CCS allowing static allocation of locations, although the equivalence is still insensitive to the actual names

that were given, as long as the topology of the locations is the same. A similar approach is taken in [144]. Furthermore every $\|$ operator in a process has to specify two location names for its two subprocesses. In our calculus operands can be left unidentified. Murphy proposes in [145] a different extension to CCS in which locations with names are given statically and considered to be observable by his notion of distributed bisimulation. Murphy has a layered approach in which local processes are given locations and composed in parallel. Nesting of locations, however, is not supported. Furthermore, communication between different locations is restricted. Equivalences are considered that are parameterised by a topology on the locations, that describes which locations are distinguishable by the observer and which locations are not. Murphy furthermore extends the communication with the ability to explicitly request a communication with a certain location. We did not use such an extension for the reason that it is not present in POOSL (although the same result can easily be obtained in POOSL by using data and a concept called conditional message reception). Such additions no longer extend only the observability, but also the behaviour of a model.
Concepts of structure architecture and components are of course prevalent in (object-oriented) modelling and specification methods and languages and programming languages for complex distributed systems. They are also applied to algebraic specification formalisms such as PSF [136]. They are an essential key to mastering the size and complexity of such systems.

5.7.2 Logics for Objects and Distribution

Integration of temporal logic properties in object-orientated methods is practiced for instance in the field of temporal constraints on object-oriented databases in [163]. The focus is on expressing temporal preconditions for database transactions using a past-time variant of LTL, in the context of the modelling language TROLL. A distributed temporal logic and proof system for objects in TROLL is introduced in [68], focussing on locality and encapsulation of objects, but not on structure and architecture. A similar approach is taken in [115] which also incorporates both state and action properties and focusses on theories for objects for encapsulation and reuse.
A temporal logic with locations was given in [38] in the form of a logical characterisation in Hennessy-Milner style for their location equivalence. It uses formulas such as $\langle a \rangle_l \varphi$ to express that a process can perform an action labelled a at location l and after that satisfy φ. It furthermore uses quantification over location variables to achieve the insensitivity to specific location names.
Modular reasoning and compositional object specification are addressed in [142] and [56], focussing in particular on remedying the defects of the Next operator for compositional reasoning. The sensitivity to finite stuttering introduced by interleaving semantics is countered by removing duplicate states from a state sequence or modifying the logic's interpretation to skip (locally) identical states.
Works that employs a decomposition of execution traces in the way we have done in this thesis, have not been found. This is probably due to the fact that CCS-like processes typically express externally observable behaviour, rather than the internal states of a system. In the context of the language POOSL, this is more natural however, since in POOSL a process expression explicitly defines the hierarchical struc-

ture and states of the system components, besides their behaviour. In particular the SHESim tool simulates closed POOSL specifications in which external behaviour is absent.

5.8 Conclusions

We have decorated our systems with identification of components. Moreover, the semantics of the calculus has been decorated such that it does not just define the external behaviour of a system, but exposes the internal actions at the borders of components as well. It has been shown that the decoration and corresponding changes to the semantics have not changed the behaviour; if all decorations are removed, the external behaviour is still the same. The decorated semantics provide a formal basic reasoning about the internals of a system.
The added decorations in both the process expressions and in the actions they perform allows the contributions of individual components to be regained from the behaviour of the entire system. For any component of the system, its local trace, states and transitions can be determined. It has been demonstrated how this can be used to interpret temporal logic formulas expressing properties of individual components or relating properties of multiple components. The interpretation of logical properties of components is insensitive to transitions of parts of the system outside of the component itself. This gives a compositional interpretation of the Next operator and allows one to ignore such external transitions during the verification of logical properties associate with a component.
Future work includes the precise definition of a full language to express properties of POOSL models, which can then be used for automatic verification of such properties. Linear temporal logic by itself may not be powerful enough to express all desired properties, it will sometimes be necessary to add behaviour to the model to allow the expression of the desired property. If for instance, one would like to relate the data content of different communications referred to in a property, one may need to add a variable containing the message last received.
Another problem that needs to be addressed is the fact that designers are not accustomed to using formalisms like temporal logic directly and it does not seem to be accepted very well in industrial practice. There exist however different kinds of notations to express temporal properties that are more user friendly, such as scenarios, timing diagrams and message sequence charts. Such notations could be used directly or be automatically translated to temporal logic formulas. Another option is to use certain predefined properties, captured using structured natural language as practiced in [113].

Chapter 6

Automata Theoretic Verification

In the previous chapters we have seen how it is possible to model complex distributed reactive and interactive systems and execute these models according to the formal semantics of the modelling formalism. The remainder of this thesis will focus on the automatic (formal) verification of properties of these systems. The formal techniques of the previous chapters are aimed at the description and execution of complex structured systems. Such complex systems suffer in particular from the state-space explosion problem. In this chapter we discuss the automata theoretic approach to verification [178] and its applicability to these systems. In particular we discuss the use of non-exhaustive verification techniques that increase the range of systems to which it can be applied, and simulation as one of the most frequently used means of verification in practice.
In section 6.1 we discuss the automata theoretic approach to verification and in particular to temporal logic model checking. Section 6.2 discusses the use of verification approaches that are non-exhaustive. In these approaches, the guarantee of finding a definite answer to the verification question is traded for the applicability to larger systems. Automatic verification of simulation results is discussed in section 6.3. Related work and conclusions are presented in sections 6.4 and 6.5.

6.1 Automata Theoretic Verification

Concurrent systems have been modelled by labelled transition systems. If the transition system is finite, it can be effectively analysed. If the original model is not finite, finiteness is often achieved after abstraction of irrelevant details. Timed systems often have infinitely many states, (densely timed systems inherently have (uncountably) infinitely many states), but analysis may still be possible by considering a finite quotient of the system with respect to suitable equivalence relations. This technique can be used for instance, to effectively test the emptiness of timed automata [7]. Finite labelled transition systems are finite state automata without acceptance conditions (often called safety automata). If required, fairness constraints can be imposed on such systems using acceptance conditions on the automata. Algorithms for analysing ω-automata (finite-state automata on infinite words) can then be used for the analysis of such systems.

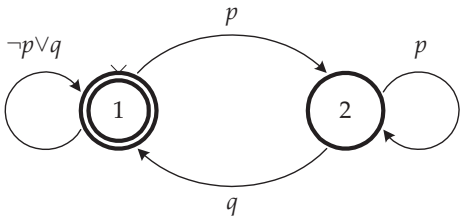


Figure 6.1: Tableau automaton of the formula $\Box\,(p \Rightarrow (p \mathsf{U} q))$

In order to automatically verify properties of a model using ω-automata, the properties must also be expressed in terms of automata, or be expressed in some other formalism and automatically be translated into automata. In the remainder we consider in particular the verification of properties expressed in (timed) linear temporal logic [155, 134, 12, 9]. Temporal logic is a popular formalism to express properties of concurrent and reactive systems and can be used to formalise specifications such as timing diagrams ([162]), message sequence charts ([111, 15]) or natural language patterns ([113]), that are more accessible to designers. This approach requires a connection between temporal logic formulas and automata. This connection is realised by a translation, called a *tableau construction* [184] that transforms a temporal logic formula into an ω-automaton that captures the same property. Besides the verification of LTL properties, it is possible to use other formalisms to specify properties, such as process algebraic specifications, or to use state-machines and automata directly.

6.1.1 LTL Model Checking

It is well known ([184, 89, 60]) that it is possible to effectively translate an LTL formula φ into an ω-automaton A_φ (called the *tableau automaton* of φ) that accepts precisely the state sequences that satisfy φ, i.e.

$$\mathcal{L}(A_\varphi) = \{\overline{\sigma} \mid \overline{\sigma} \models \varphi\}.$$

The automaton in figure 6.1, for example, accepts precisely all state sequences that satisfy the LTL formula $\Box\,(p \Rightarrow (p \mathsf{U} q))$. How such an automaton is constructed from an LTL formula is discussed in chapter 7. It allows one to check whether every execution of a system S, described by automaton A_S, satisfies the LTL property φ. This can be achieved by taking the formula φ and constructing the corresponding tableau automaton A_φ. Now the model-checking problem can be stated entirely in terms of automata, namely the system S satisfies the LTL property φ if and only if $\mathcal{L}(A_S) \subseteq \mathcal{L}(A_\varphi)$.

6.1.2 Model-Checking Algorithms

LTL model-checkers are algorithms that produce an answer to the question whether the system S satisfies the property φ by determining if $\mathcal{L}(A_S) \subseteq \mathcal{L}(A_\varphi)$. The typical procedure to get this answer is the following (see figure 6.2). Instead of trying to

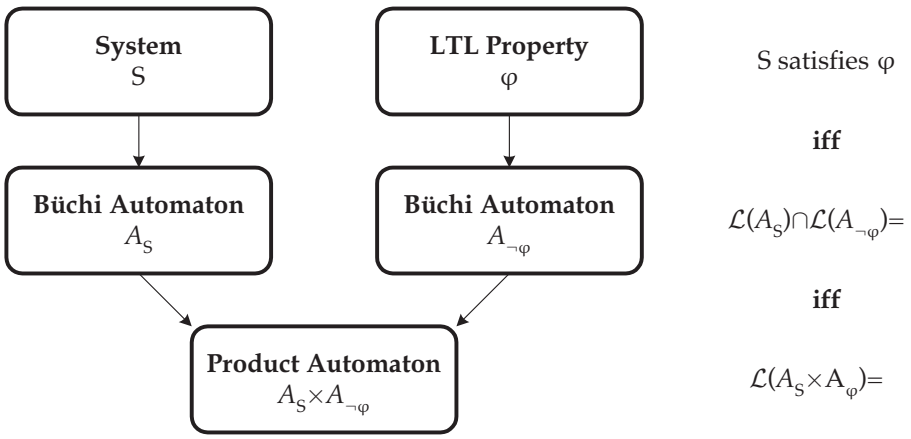


Figure 6.2: LTL model checking

check the language inclusion directly, another approach is taken for reasons of complexity. In principle, the inclusion can be checked by computing the complement $\overline{A_\varphi}$ (the automaton accepting precisely all state sequences that are *not* accepted by A_φ) and checking if $\mathcal{L}(A_S) \cap \mathcal{L}(\overline{A_\varphi}) = \varnothing$. The complexity of complementation of an automaton however, is exponential in the size of the automaton ([171]). Instead, the complement is computed directly from the negation of the formula, by observing that $\mathcal{L}(\overline{A_\varphi}) = \mathcal{L}(A_{\neg\varphi})$.
Subsequently, the synchronous product, $A_S \times A_{\neg\varphi}$, of both automata is computed (knowing that $\mathcal{L}(A_S \times A_{\neg\varphi}) = \mathcal{L}(A_S) \cap \mathcal{L}(\neg\varphi)$) and for S to satisfy φ, this new automaton should not accept any state sequences at all. This can be checked by performing a nested depth-first search on the state space of the product automaton to search for a reachable accepting cycle through the states of the automaton space[179]. If a corresponding accepted state sequence is found, there is an execution that satisfies $\neg\varphi$ and that hence does not satisfy φ. This state sequence and the corresponding path through the automaton serve as a counterexample which demonstrates this fact. The ability to complete a negative result with a counterexample is considered to be one of the biggest advantages of the model-checking approach to verification.

6.2 Non-Exhaustive Model Checking

6.2.1 The Limits of Model Checking

The model-checking procedure described in the previous section results in an answer to the question whether a system is correct with respect to a particular LTL property. The practical problem to this approach however is that the size of the Büchi automaton A_S is for most systems exponentially related to the size of the description of the system S. This problem is called the *state-space explosion problem* and makes exhaustive verification very hard and limitations in memory and/or time resources make it practical for small or medium-sized systems only. Although methods have been

and are being developed, such as symbolic verification [44], partial order reduction [90, 34], compositional verification [53] and abstraction techniques [58, 114], that try to alleviate the problem, it is a fundamental one and remains the limiting factor in the application of model-checking techniques.

Models must be kept small and abstractions are required to achieve this. Sometimes it can be shown that the abstraction preserves the properties that are being checked or that it preserves positive or negative results of the verification only. Often however, these abstractions are made by the designer who believes them to be valid or reasonable, without demonstrating this explicitly. There is a risk that the abstract system no longer adequately describes the actual behaviour, or that erroneous behaviour is introduced or removed from the model by the abstraction. The risk becomes bigger if the designer is forced into making more or bigger abstractions, fighting the limitations in memory and time resources.

6.2.2 Non-Exhaustive Verification

To address these problems, one can try giving up the conclusive answer to the verification question and consider non-exhaustive verification. This means that a verification program will search for an answer to the question as far as the resource limitations allow; giving an answer if it finds one, but possibly no answer at all. A model-checking tool for example may have a mode in which the state-space search starts back-tracking the depth-first search when a certain fixed search depth is reached, thus excluding a part of the state space from the search. If no erroneous behaviour is found this way, the result is inconclusive, since there might have been an error in the parts of the state space that were skipped. If an error is found however, the conclusion that the system does not satisfy the property is valid. Such a mode of operation is for example available in the model checker Spin [107].

Besides the complexity and efficiency of the algorithm, for non-exhaustive methods it is important to improve the likelihood of finding problems, by adjusting the parameters that govern the direction of the search. Time or memory-resource-intensive analyses are often run until one runs out of either memory or patience. Thus, improvements in the error-finding capabilities can in this case be achieved by manipulating the search algorithm to search in 'interesting' parts of the state space first [185, 78].

An example of a technique that saves memory at the cost of knowing for certain if a particular location was visited before, is the following. Instead of storing all the locations that have been visited, one could store only an abstract representation of the state, for instance, control locations only without data, or a hash value ([57]). This information can be used to detect that a state has not been visited before (the converse can no longer be established with certainty). If a state is found that has possibly been visited before, the search in that direction is aborted. To establish the faulty execution trace, it is still necessary to remember all states visited on the path that is currently being investigated (the depth-first-search stack)[1].

[1] A full state space search is even possible with storing the DFS stack of states only, using a technique called *state space caching* [92].

6.2.3 Simulation

If state representations become extremely large and hard to manipulate efficiently, one may have to refrain from storing any states, even those on the path leading to the current state. Unfortunately, a systematic search of the state space is then virtually impossible[2] and infinite paths (in the form of reachable cycles in the state space) can no longer be detected. Such a technique, that that does not require states to be stored, is random simulation and the errors that it allows to detect are those which can be expressed as the reachability of some 'bad' location (this is elaborated in section 6.3). Also in simulation, the odds of finding particular locations can be improved by guiding the simulation instead of selecting transitions at random. This can be achieved for instance by modifying the conditions to steer the model towards exceptional behaviour, such as deliberately increasing the probabilities of the occurrence of events to levels higher than in the actual realisation.
Simulations are used frequently and are a valuable tool in practical design. The evaluation of the simulation results however is often done manually or in an ad hoc and informal fashion. The addition of verification of formal properties could improve the quality of the design and reduce the gap to (more) exhaustive, dedicated verification tools.

6.2.4 A Comparison of Verification Methods

The different methods in the spectrum from exhaustive verification to simulation techniques have different characteristics that make them complementary and suitable for different stages of the design trajectory. We summarise some advantages and disadvantages, beginning with those of exhaustive verification methods [84].

- The state-space explosion problem limits the applicability of exhaustive verification methods to relatively small models.
- As a consequence, considerable abstractions are necessary requiring a significant investment in the construction of abstract models. Correctness and adequacy of the abstraction may be questionable.
- Although formal verification methods are slowly finding their way into commercial tool sets and industrial practice, they are still frequently immature and hard to use and understand. Many require in-depth knowledge of formal methods in order to be used effectively as illustrated by the fact that success stories of applied formal verification often include the original makers of the tools or the theory [117].
- If resources allow, exhaustive verification methods are guaranteed to find all errors that are present in the model if covered by the formalised correctness requirements.

[2] In the Verisoft tool [91], backtracking is achieved by storing the sequence of events leading to the current state, not the states themselves and replaying the sequence from the initial state. This can be extremely time consuming and cycles cannot be detected. Aggressive partial order reduction techniques are applied to prevent backtracking as much as possible.

- The methods are particularly good at examining parts of the state space that are hard to reach by simulations, such as very particular orders of occurring events. The probability of reaching these parts by a random walk through the state space can be very low.
- There are also some positive 'side-effects' of using such precise and formal methods. It requires designers to look at their designs and requirements very precisely. This activity alone tends to reveal many problems already.

In contrast, properties of non-exhaustive verification methods and simulation in particular are listed below.

- The search of the state space is not complete in non-exhaustive methods. It is difficult to quantify the coverage obtained by such an analysis and careful tuning of an algorithm's parameters can have a large impact on the obtained results [78].
- Many non-exhaustive verification methods provide a trade-off between the quality of the analysis and the amount of resources or effort put into it. This makes that one is not left empty-handed if the problems turn out to be too hard to be tackled completely.
- A similar trade-off exists between the level of detail and adequacy of the model and the quality of the analysis. A more detailed model may better reflect the characteristics of the physical artifact that is being designed. A more abstract model allows for a more thorough analysis.

As a particular case of non-exhaustive methods, simulation possesses the following characteristics.

- The coverage (defined as either the fraction of the state space that has been examined or as the fraction of errors that is found) is lower than in an exhaustive analysis. If one relies on probability to eventually visit a representative fraction of the state space, the odds of finding an error depend heavily on the kind of scenario that evokes it [181]. If it requires a very specific order of events and subtle interactions of many different subsystems, it might be extremely hard to find.
- Simulation is a technique that is generally easy to use. Modelling is typically done using languages that resemble traditional imperative programming languages. This makes that simulation tools are often much easier to use than many of the verification tools.
- In a simulation, storage of states is not required. The operation of a typical (automata theoretic) exhaustive verification tool involves the frequent manipulation of system states. In order to perform such an analysis on a large state space in a limited amount of memory, it must be possible to efficiently produce and manipulate compact state representations. In more complex models, this may quickly become a problem. In particular this is very hard in the context of an object-oriented language where data structures can be quite complex and which allows for data objects to be created dynamically [74, 156]. Simulation is

thus easier applicable to such models and languages as it avoids dealing with states in this manner.

- A simulation run explores a single execution of the model. This makes it suitable for the verification of *linear* temporal logic properties (expressing aspects of individual executions) only. To be valid, a property should hold for all executions of the system. Since this number can be large (it is often infinite), this seems to lead to extremely low coverage. If however the system has recurrent behaviour, then a single execution will engage in the same behaviour over and over again, having the same effect as performing multiple execution runs. If the behaviour of the system does not exhibit such recurrent behaviour it could be necessary to perform a large number of individual simulation runs.

An executable specification is (or should be) one of the first tangible results in the design trajectory. Then a simulation of such a specification can be first applied for initial validation and verification of the system. The number of errors in the design is likely to be large and many of these will be caught during these initial simulations. As the design matures, some errors may remain that have escaped detection during simulations. If required, deeper analyses can be performed on (parts of) the system, giving better confidence in the correct operations of the system at the expense of more involved analysis and the use of larger amounts of resources.

6.3 Checking Properties in Simulations

Although simulation has some strong limitations, its biggest advantages are that it is applicable to large size systems and that it is (relatively) easy to use. A first validation and verification can be performed using simulations of the initial executable models. Many errors can be found and corrected this way before starting more laborious efforts of verification using dedicated methods and tools. In this section, the practical approach towards checking LTL properties in simulations is set out.
The temporal logic discussed in section 2.5 expresses properties of infinite sequences of states. In an exhaustive verification, the tool infers properties of infinite computations from cycles through the (finite) state space. During a simulation, it is not possible to detect that one has reached a state that was visited before and thus cycles cannot be discovered. One must infer properties of infinite executions without witnessing such an infinite path. One way to deal with this problem is to consider the set of all possible extensions of the finite prefix of the execution that has been observed so far. Now there are three possible situations which can be discriminated. If the set of all extensions includes both traces that do and traces that do not satisfy the property, than we cannot draw any conclusions about the validity of the property and we have to continue simulation. If the set contains only executions that violate the property then one does not need to simulate any further and it can be concluded that the system violates the property. Such a prefix is called a *bad prefix* in [123]. For example, suppose that it is specified for a particular system that between any two instances that a message m is received, there must be an instant where some condition c holds. If a simulation produces a prefix of a trace of this system that shows that two such messages m are received, but no instant when c was true, then this prefix is bad with respect to the property. The remainder of the trace cannot restore the property.

The third possibility is that all extensions of the trace do satisfy the property, then the prefix is called a *good prefix*. However, in contrast with the bad prefix we cannot conclude that the property is valid for the system. We can only conclude that the property is valid for the particular execution that we have witnessed. The difference is that the existence of a single trace violating the property makes that the system violates the property, but a single trace satisfying it does not prove that the system satisfies it. It only shows that the system *may* satisfy the property.

6.3.1 Automata for Bad Prefixes

Two types of temporal logic properties can be distinguished, *safety properties* and *liveness properties* (sometimes called eventuality property) [4, 164]. Informally, a safety property states that "something bad will never happen", whereas a liveness property states that "something good will eventually happen". Since a simulation reveals only finite prefixes of an infinite execution trace, only the violation of safety properties can be detected by a simulation (and the satisfaction of liveness properties on some trace). One may detect that something bad has happened, but it is impossible to conclude after a finite amount of time that 'something good' is never going to happen. Several different formal definitions of safety properties exist ([125, 134, 4, 164]). The following definition of safety originates from [4]. Let a property be a set of state sequences. We define safety properties as precisely those properties for which every state sequence that violates the property has a *bad prefix*. This holds for instance for the property described above; any trace that fails it, displays some point where two messages are received without witnessing the condition. The property fails to hold, irrespective of the remainder of the trace.

A prefix of a state sequence is a bad prefix for a safety property P, if it cannot be extended into an infinite state sequence that satisfies P. For instance, suppose we have the following prefix of a state sequence: a number of states where p holds and q does not hold, followed by a state where neither p nor q holds. This prefix forms a bad prefix of the property expressed by the LTL formula $p\mathsf{U}q$. No extension of the prefix can satisfy the formula. More about safety properties and model checking of safety properties can be found in [123] and in chapter 7 on tableau automata, were the construction is discussed of finite state automata that recognise bad prefixes.

A modification of the tableau algorithm produces an automaton on finite words that recognises bad prefixes ([84, 123]). In model checking during forward simulations, one essentially builds a deterministic automaton that incrementally monitors the prefix of the infinite execution that has been simulated so far. If the prefix is such that the execution can no longer lead to satisfaction of the property φ (it is a bad prefix), then the simulation can be stopped and the witnessed execution demonstrates that the property does not hold for the system. [123] discusses complexity results of constructing automata that recognise bad prefixes and a slightly weaker (but smaller) automaton that recognises the so-called 'informative bad prefixes'. The construction of a tableau automaton (on infinite state sequences) A_φ is exponential in the length of the formula φ [179]. Constructing an automaton which recognises the bad prefixes of $L(A_\varphi)$ yields an automaton (on finite prefixes of state sequences) exponential in the size of A_φ and thus doubly exponential in the length of the formula. An alternative construction which recognises just the informative bad prefixes (see figure 6.4)

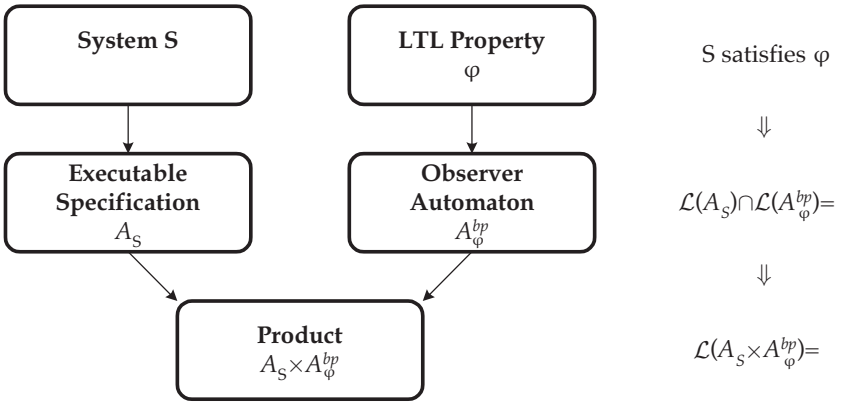


Figure 6.3: Model checking in simulations

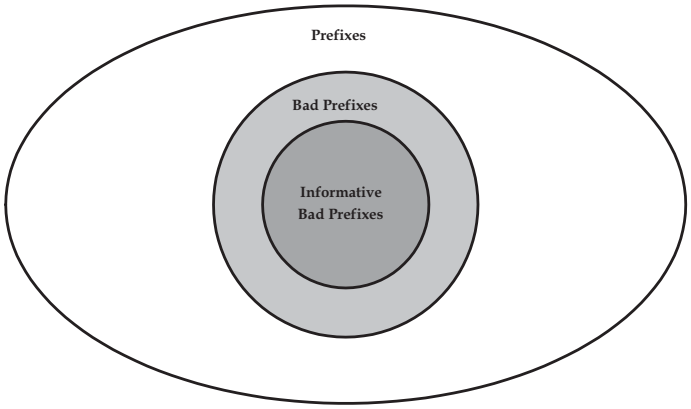


Figure 6.4: Prefixes of state sequences

is only singly exponential in the length of the formula (yielding a non-deterministic automaton). In practice this automaton is adequate, because the informative bad prefixes are precisely those prefixes that 'tell the whole story' about why the formula is violated (see [123] and chapter 7). The formula $(\Box p) \wedge \Diamond \neg p$ for instance, is not satisfiable and therefore any prefix of some state sequence is a bad prefix. Not every prefix however contributes anything to the explanation *why* the formula does not hold. Informative bad prefixes for the property do exist, a prefix showing a state where p does not hold is informative. It tells us that $\Box p$ does not hold and thus neither does $(\Box p) \wedge \Diamond \neg p$.

Another issue that arises if we use the automata theoretic approach to model checking for simulations, is non-determinism. We know that the system automata are inherently non-deterministic and the simulation explores only a single of all possible executions. In general however, tableau automata may also be non-deterministic. Non-determinism in the system automaton originates for an important part from inherent non-determinism in the model and as such the modeler may exercise some

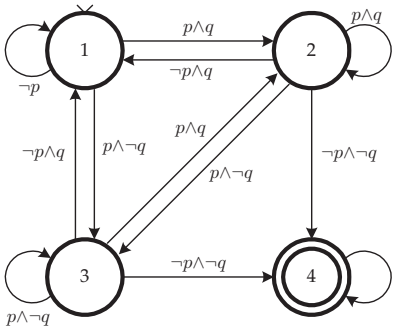


Figure 6.5: Determinised tableau automaton for bad prefixes, of the formula $\Box\ (p \Rightarrow (p \mathsf{U} q))$

influence on such choices if required. Non-determinism in the property automaton however is not under control of the designer and also unnecessary. It is therefore desirable that the observer automaton is deterministic [84]. The automaton that recognises bad prefixes is an automaton on finite words rather than a Büchi automaton. This makes that the automaton can be determinised by a subset construction at the cost of (another) exponential blowup in size. The explicit construction of the deterministic automaton can in practice be avoided by implicitly doing the subset construction while traversing the automaton during a simulation. Figure 6.5 shows a determinised version of the tableau automaton of figure 6.1. It can be shown that no informative bad prefix of the formula $\Box\ (p \Rightarrow (p \mathsf{U} q))$ has a path through the automaton (and in this case no bad prefix at all). The precise definition of an informative bad prefix is postponed until the next chapter.

Constructing Observer Automata

Observing multiple properties during a simulation can be done by observing the property formed by taking the conjunction of the individual properties. If a bad prefix is found however, it may not be clear which of the properties was violated. Alternatively, one can produce an automaton for every property individually and use the synchronous product of these automata, with a state of the product being marked as final (indicating a bad prefix) if any of the constituting states is accepting. This acceptance marking then reveals which of the properties was violated. Again, this product need not be constructed explicitly.

If an observer is insensitive to (arbitrary, finite) repetitions of states in a state sequence (called stuttering)[3] and its propositions refer only to a particular component of the system, then the synchronous product with the observer can be formed at the level of the component, rather than at the level of the entire systems. This could save unnecessary updating of the observer's state.

[3]Properties expressed in LTL without using the $\bigcirc$ operator for instance, are insensitive to stuttering.

6.3.2 Observer Automata in the SHESim Tool

For every property that is used in the model, an observer automaton can be constructed. The atomic propositions to which the property refers, are associated with the appropriate expressions defining them in terms of the model. The tableau automaton can be built as described in section 6.3.1 for every property that is thus observed.

The simulator is built as a discrete-event simulator that continually selects a possible transition of the system and executes it. Instead of explicitly constructing a new system which represents the synchronous product of the POOSL specification and the observer automata, steps of the system and steps of the observers are alternated. Since the observers are deterministic, their new states will be uniquely determined once the system has changed its state. The only remaining non-determinism originates from the system. The scheduler selects one of the available transitions according to some scheduling mechanism (possibly at random), executes the transition, computes the new values of the atomic propositions and makes the corresponding moves in all the observers. If a bad prefix and thus a violation of the property is detected, the user is able to determine the cause by inspection of the current state of the system and whatever is remembered about the history of the execution, such as the sequence of communicated messages in an interaction diagram [87]. If this is not enough the same simulation can be run again and observed more closely, since one now knows what property will be violated. For example, suppose that a system is simulated and produces the following state sequence $\{p,q\}\{p\}\{p\}\varnothing$. It can easily be verified that the observer automaton of figure 6.5 moves along the path $1-2-3-3-4$ to the final location 4. This implies that the state sequence observed violates the property $\Box\,(p \Rightarrow (p\mathsf{U}q))$, and the trace executed by the scheduler witnesses this fact (informatively).

6.4 Related Work

Automatic verification activities necessarily take place at the border of what is computationally feasible (some accounts of formal verification efforts can be found, for instance, in [117]). Therefore, methods are needed that extend this border, such as improved techniques, combining different techniques, automating abstractions, or more efficient data structures and implementations. At the same time, methods are needed that cross that border in the sense that certain aspects of the results sacrificed in exchange for application to a larger class of problems, such as non-exhaustive verification methods.

One such method is simulation. The effectiveness of simulation for the verification of protocols is the topic of [181] and [138]. [116] is an account of a practical case-study. It stresses the benefits of having (temporal) assertions in simulation models that are monitored whenever the model of a component is used in a simulation. It was found that errors were frequently uncovered by the observers during simulations that did not target the verification of that component in particular.

Non-Exhaustive Verification

As non-exhaustive verification techniques do not establish the correctness of a system, they are targeted at bug-hunting in particular. It becomes important to tune algorithms to find as many errors as possible and to find them as quickly as possible. Guiding the search towards suspicious parts of the state space [185, 108, 24] or combining depth-first searching and breadth-first searching techniques to find shorter traces leading to the error [78] are examples of such strategies.

Verification in Simulations

Simulation can be seen as a non-exhaustive search through a system's state space without back-tracking. Verification techniques can be applied albeit with some adaptations and limitations. Checking LTL properties during simulations is discussed in [45]. Although the basic unfolding principle of the construction of a tableau automaton is used, the main disadvantage is that formulas are manipulated directly during simulation, which may not be very efficient. Besides the basic LTL formulas, also (discrete-) timed formulas of the form $\varphi_1 \mathsf{U}^{\leq n} \varphi_2$ for example, are supported in [45]. $\varphi_1 \mathsf{U}^{\leq n} \varphi_2$ states that φ_2 is to happen before n units of time have passed and until then, φ_1 should hold. Also in [105], the observation of LTL properties in simulations of System-C descriptions is discussed. Formulas are interpreted over finite state sequences and given a three-valued interpretation. There are many others that put forward the use of observers of different kinds, to run along simulations models, e.g. [76].

Safety Properties and Bad Prefixes

So-called safety properties can be monitored during simulations by recognising bad prefixes. Kupferman and Vardi suggest a similar approach, but in the context of symbolic state space exploration in [123]. Their main goal is to simplify the traditional model-checking procedure to a simple reachability problem using a depth-first search as opposed to a nested depth-first search which is required when looking for reachable cycles in the state space required for the verification of liveness properties. [123] does not focus in particular on the aspects of simulations. More on safety and liveness can be found in [4, 164].

Verification of Complex Systems

Modern modelling formalisms, such as object-oriented systems and systems with complex data structures, do not blend naturally with efficient verification tools and their modelling formalisms. Verification of complex and object-oriented systems specified in a formalisation of the Fusion methodology ([55]), called FUS++, is discussed in [74]. The focus is on translating FUS++ models to Promela, the input language of the model-checking tool Spin. A distinguishing feature is the translation of dynamically created and destroyed objects into the static vector-based data of Promela, although this does not come without efficiency problems.

Property formalisms

The issue of formally specifying the properties to which a system should adhere, in a designer-friendly way is also important. Scenarios, timing diagrams and message sequence charts are notations that are popular among designers and describe wanted, required or forbidden behaviours. Using structured natural language could also be a way to simplify the collection of correctness properties. Deriving formalised properties, ready to be used in verification tools is discussed in [165]. Other work on formal requirements includes [15] on analysis using message sequence charts, [135] on formalising the semantics of message sequence charts, [162] about converting timing diagrams into formal specifications and [113] about the use of structured natural language patterns to derive requirements. The conversion of process algebraic specifications to equivalent timed automata is the topic of [148].

6.5 Conclusions

The maturity and capabilities of formal verification tools increase with performance of computing machinery and new and better techniques. Yet, it is to be expected that such verification methods will not be able to deal with arbitrarily large and complex systems, such as are often used for initial validation and simulations. Dedicated adaptations and abstractions of the models are required, some of which can be automated.

To reduce the time involved in a thorough and detailed analysis of a system's correctness, it is essential to reduce the number of errors in the design as much as possible before starting such an investigation. To achieve this we would like to improve the error-finding capabilities of simulation tools.

Non-exhaustive verification techniques are introduced to stretch the applicability of verification methods to larger systems, in return for giving up the exact answer to the verification question. This enables a trade-off between scalability and quality of the verification results. Simulation augmented with the automatic verification of (safety) linear temporal logic properties is such a technique. To do this, some adaptations to the automata theoretic verification method are required and in particular to the construction of observer automata. The observers have to (incrementally) analyse finite prefixes of executions of the system rather than infinite state sequences and they have to be deterministic. The construction of such observers is the topic of the following chapters. Tableaux constructions for untimed linear temporal logic and for simulations are discussed in chapter 7. Tableau constructions for real-time temporal logic are the topic of chapter 8.

Chapter 7

The Tableau Method for Linear Temporal Logic

In this chapter we describe the translation from linear temporal logic to automata, for describing properties of infinite (linear) executions of untimed systems. This chapter will review the tableau method for linear temporal logic. It introduces a general framework for defining 'complete' as well as 'on-the-fly' tableau constructions. It also introduces an approach to temporal logic verification during simulations. In the following chapter these results are extended to a tableau method for real-time temporal logic, allowing model checking of real-time properties and the construction of real-time temporal logic observers for simulations.
Section 7.1 introduces some preliminary concepts to describe the tableau automata and tableau automata themselves, similar to the ones of [184] and [122]. Then we make a generalisation to accommodate the so-called 'on-the-fly tableaux' [89, 60, 163, 67, 118] in section 7.2. Subsequently, in section 7.3, we discuss some interesting subsets of LTL and the corresponding tableaux, and in section 7.4, automata on finite prefixes to use the techniques in simulations. We finish with related work and conclusions in section 7.5.

7.1 Complete Tableaux

In this section we review the construction of an ω-automaton A from an LTL formula φ that accepts precisely all models of φ.
Recall the following definition of LTL formulas from section 2.5.2.

$$\varphi ::= \text{true} \mid p \mid \neg\varphi' \mid \varphi_1 \vee \varphi_2 \mid \bigcirc \varphi' \mid \varphi_1 \mathsf{U} \varphi_2.$$

We assume that we build tableau automata only for LTL formulas that do not contain the $\bigcirc$ operator. Inclusion of the $\bigcirc$ operator is straightforward, but it has no place in the timed extensions we investigate in the next chapter.
The semantics is as defined in section 2.5.2. Remember that LTL formulas are interpreted over sequences of *states*, with states being sets of atomic propositions that

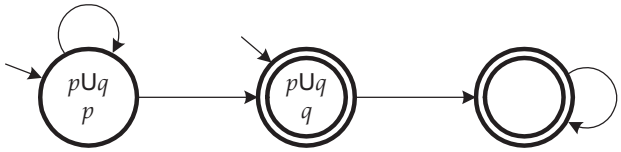


Figure 7.1: Example tableau of $p\mathsf{U}q$

indicate which propositions are true. Recall also that the nodes of the automata are called locations (rather than states) to avoid confusion.

7.1.1 Intuitive description of the construction

To build an automaton that accepts precisely those state sequences satisfying a formula φ, we associate with the locations of the automaton, sets of LTL formulas (subformulas of φ) in such a way that runs starting from some location match those state sequences that satisfy all these formulas. Moreover, locations are labelled with states, consistent with those formulas.

The formulas in a location determine the states that are allowed as the first state of the sequence. They thus determine the labelling of the locations with states. If there are several ways to satisfy a formula ($\varphi_1 \vee \varphi_2$ can be satisfied by satisfying φ_1 or by satisfying φ_2), non-deterministic choice allows the automaton to 'guess' the right choices for the future. Thus a disjunction of formulas is captured by non-deterministic choice in the automaton. As there is no corresponding concept for conjunctions[1], conjunctions of formulas are represented explicitly by sets of formulas in the locations of the automaton. Depending on the Until and Release formulas in a particular location, restrictions must be imposed on the rest of the state sequence; this defines the edges leaving the location.

Figure 7.1 shows a tableau automaton for the formula $p\mathsf{U}q$. Any run starting from the left-most location satisfies the formulas $p\mathsf{U}q$ and p. If we consider states over the propositions p and q ($\{\varnothing, \{p\}, \{q\}, \{p,q\}\}$) then this location is labelled with the states $\{p\}$ (p holds and q does not hold) and $\{p,q\}$ (p and q both hold). Since the occurrence of p alone does not guarantee that $p\mathsf{U}q$ is satisfied, the outgoing edges only lead to locations labelled with $p\mathsf{U}q$ as well (if p holds in the first state and $p\mathsf{U}q$ holds for the state sequence starting with the next state, then $p\mathsf{U}q$ holds for the entire state sequence). Similarly, the second location, also an initial location, contains the formulas $p\mathsf{U}q$ and q; thus it is labelled with the states $\{q\}$ and $\{p,q\}$. Therefore a run from this location enforces that the first state satisfies q. From this it follows immediately that the entire run satisfies $p\mathsf{U}q$. This is why the outgoing edge leads to a location containing no formulas at all; the remainder of the state sequence is irrelevant. From this location, anything is allowed and no more constraints are placed on the following states. It is therefore labelled with all states of $2^{\{p,q\}}$.

[1] Automata with universal choice do exist, they are called alternating automata [48, 141]. Tableau constructions using alternating automata can be found in [177]. However, analysis of alternating automata is more complicated.

Consider the following state sequence

$$\{p,q\}\{p,q\}\{q\}\varnothing\ldots$$

This state sequence satisfies the formula $p\mathsf{U}q$. In fact, there are three different initial runs for the state sequence. Firstly, there is a run starting from the middle location, continuing to the right location and remaining there. This run corresponds to the observation that the first state satisfies q and thus the entire state sequence satisfies $p\mathsf{U}q$, no matter what the rest of the states are. Another run starts from the left location and goes via the middle location to the right location. This run demonstrates that p holds in the first state and that the remainder of the sequence satisfies $p\mathsf{U}q$. Together this demonstrates that the entire state sequence satisfies $p\mathsf{U}q$. Similarly there is a run for this state sequence that starts in the left location, remains there for another state, and then moves to the middle and the right locations. Note that only the middle and the right location are marked as accepting locations. This prevents a run from staying at the left location forever. From the labelling of the location we see that it requires state sequences that pass through the location to satisfy $p\mathsf{U}q$, by having the current state satisfy p and the remainder of the sequence satisfy $p\mathsf{U}q$. Although it may take arbitrarily long for a state sequence to produce a state satisfying q, it may not refuse forever to satisfy q. This is enforced by the acceptance conditions.
We discuss the different concepts that are used to build tableau automata in more detail.

Local Consistency Only sets of formulas can be considered that are consistent in themselves. A location can, for example, not require that $\psi_1 \vee \psi_2$ is true, without requiring also either ψ_1 or ψ_2 to hold. This is covered by the notion of local consistency that will be introduced.

Temporal Consistency Constraints are also imposed on the formulas at subsequent moments. This is illustrated in figure 7.2, the (large) circles in this figure (and in figures 7.4, 7.5 and 7.6 to follow) denote certain *classes of locations* (instead of individual locations), characterised by the formulas they are labelled with. The presence of an arrow between such classes of locations denotes the consistency of an edge between any two locations in the respective classes w.r.t. the constraints imposed by the particular Until formula under consideration. It does not mean that there actually is an edge between any two such locations. Conversely however, it is true that if there is no arrow between classes in the figure, then there is no edge between any two locations in those classes. A double circle denotes that all locations in that class are accepting w.r.t. the acceptance conditions imposed by that Until formula, as explained below ('liveness').
If the automaton requires that $\psi_1\mathsf{U}\psi_2$ be true at some moment k, but does not require ψ_2, then it must require at moment $k+1$ that $\psi_1\mathsf{U}\psi_2$ is still true. A location that contains the formula $\psi_1\mathsf{U}\psi_2$ and ψ_2 as well is called trivial for $\psi_1\mathsf{U}\psi_2$ (locations represented at the bottom of figure 7.2), since $\psi_1\mathsf{U}\psi_2$ follows immediately and the remainder of the state sequence is irrelevant (as far as the formula $\psi_1\mathsf{U}\psi_2$ is concerned). If it does not contain ψ_2, then the location is called non-trivial for $\psi_1\mathsf{U}\psi_2$ (right of figure 7.2); local consistency requires that such a location is labelled with ψ_1 and a transition to another location is only allowed if it contains $\psi_1\mathsf{U}\psi_2$ again.

A similar thing holds for the dual formula $\neg(\psi_1 \mathsf{U} \psi_2)$: if a location requires it to hold then local consistency demands that the location contains $\neg\psi_2$. If it also contains $\neg\psi_1$, it is called trivial, the remainder of the state sequence is irrelevant and any kind of transition is permitted. If the location does not contain $\neg\psi_1$ it is non-trivial and every following location must still require $\neg(\psi_1 \mathsf{U} \psi_2)$. These constraints on the edges of the automaton are covered by a notion called 'temporally consistent successor'. A location ℓ_2 is a temporally consistent successor of location ℓ_1 (w.r.t. the Until formula under consideration) precisely if there is an arrow between the class of locations to which ℓ_1 belongs and the class of locations of ℓ_2 in figure 7.2.

Completeness Initially, we will look at tableaux in which every location contains for every subformula ψ of the original formula φ either ψ or $\neg\psi$ (such a location is said to be complete w.r.t. φ). W.r.t. every Until subformula $\psi = \psi_1 \mathsf{U} \psi_2$ of φ, we can then partition the locations of the tableau into four kinds. Locations can either require ψ or $\neg\psi$ and in either case they can be trivial or non-trivial. A location containing ψ ($\neg\psi$) is trivial for ψ ($\neg\psi$) if the satisfaction of ψ ($\neg\psi$) follows immediately from the other requirements of that location without posing any restrictions on the remainder of the state sequence. It is called non-trivial otherwise.

Liveness Acceptance conditions are used to capture the 'liveness aspects' of the consistency. If the automaton requires at some time that $\psi_1 \mathsf{U} \psi_2$ holds, then it must see to it that at some later time, ψ_2 is required to hold (see figure 7.2). Temporal consistency alone is not enough, since the moment that ψ_2 is true may be postponed forever. To prevent this, an acceptance set is added for every Until formula occurring in φ, using generalised Büchi acceptance (see section 2.6). In this acceptance set, all locations requiring $\neg(\psi_1 \mathsf{U} \psi_2)$ and all locations requiring ψ_2 are marked as accepting. This guarantees that along any run, whenever a location that requires $\psi_1 \mathsf{U} \psi_2$ and $\neg\psi_2$ is entered, at some later moment, a location requiring ψ_2 is entered.

7.1.2 Definition of the Tableau Automaton

In this section we make the definition of the tableau according to the ideas of the previous section precise. Sometimes it will be convenient to use formulas in positive form (replacing the negation with duals of all operators, see section 2.5.2) and at other times the use of the negation will be convenient. We will identify a negated formula with the same formula in positive form. For example, $\neg(p \mathsf{U} \neg q)$ and $(\neg p) \mathsf{V} q$ denote the same formula and the same holds for $\neg\neg\psi$ and ψ. The formal definitions and proofs are all based on formulas in positive form and assume that $\neg\psi$ denotes the corresponding formula in positive form.
If φ is an LTL formula, then we use $sub(\varphi)$ to denote the set of syntactic subformulas of φ.

7.1.1 DEFINITION *The set* $sub(\varphi)$ *of* subformulas *of an* LTL *formula* φ *is the smallest set* Φ *of formulas such that*

- $\varphi \in \Phi$;
- *if* $\psi_1 \vee \psi_2 \in \Phi$ *then* $\psi_1 \in \Phi$ *and* $\psi_2 \in \Phi$;

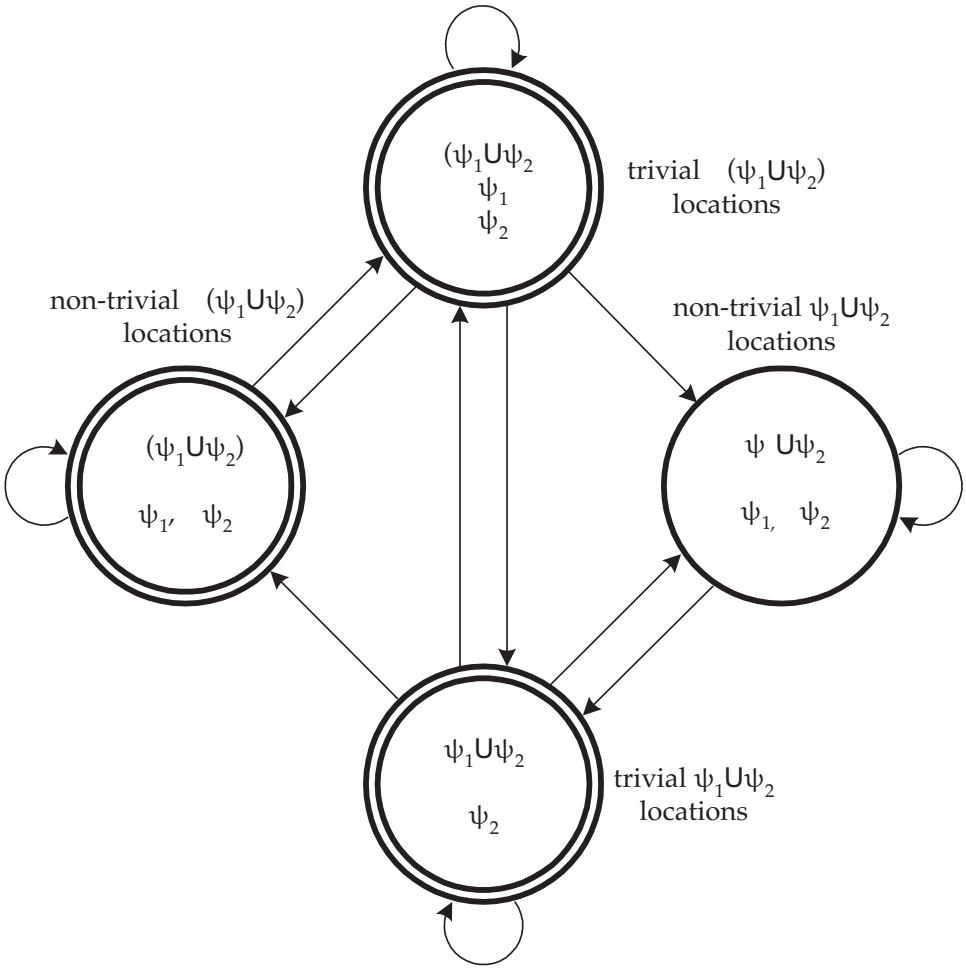


Figure 7.2: General pattern of temporal consistency with respect to the formula $\psi = \psi_1 U \psi_2$

- *if* $\psi_1 \wedge \psi_2 \in \Phi$ *then* $\psi_1 \in \Phi$ *and* $\psi_2 \in \Phi$*;*
- *if* $\bigcirc\psi' \in \Phi$ *then* $\psi' \in \Phi$*;*
- *if* $\psi_1 \mathsf{U} \psi_2 \in \Phi$ *then* $\psi_1 \in \Phi$ *and* $\psi_2 \in \Phi$*;*
- *if* $\psi_1 \mathsf{V} \psi_2 \in \Phi$ *then* $\psi_1 \in \Phi$ *and* $\psi_2 \in \Phi$*.* □

We will frequently need the set of some formula's syntactic subformulas *and* their negations; such a set is called the Fischer-Ladner closure.

7.1.2 DEFINITION *The* (Fischer-Ladner) closure *of an* LTL *formula* φ *is the set*

$$cl(\varphi) = sub(\varphi) \cup \{\neg\psi \mid \psi \in sub(\varphi)\}.$$

□

The following definition formulates when a set of formulas is considered to be *locally consistent*. A set is said to be locally consistent if it is 'informative' in the sense that the requirement of every compound formula in the set is supported by simpler requirements. Together the formulas explain *why* a requirement will hold. If a location requires $\psi_1 \wedge \psi_2$ to hold, then it must also require both ψ_1 and ψ_2. Similarly if a location requires $\psi_1 \mathsf{U} \psi_2$ then it must require ψ_1 or ψ_2 as well.

7.1.3 DEFINITION *A set* Φ *of formulas is called* locally consistent *if,*

1. false $\notin \Phi$*;*
2. *if* $\psi_1 \vee \psi_2 \in \Phi$ *then* $\psi_1 \in \Phi$ *or* $\psi_2 \in \Phi$*;*
3. *if* $\psi_1 \wedge \psi_2 \in \Phi$ *then* $\psi_1 \in \Phi$ *and* $\psi_2 \in \Phi$*;*
4. *if* $\psi_1 \mathsf{U} \psi_2 \in \Phi$ *then* $\psi_1 \in \Phi$ *or* $\psi_2 \in \Phi$*;*
5. *if* $\psi_1 \mathsf{V} \psi_2 \in \Phi$ *then* $\psi_2 \in \Phi$*.* □

Moreover, a set Φ of formulas is called *complete* w.r.t. the formula φ if for every $\psi \in cl(\varphi)$, exactly one of ψ and $\neg\psi$ is in Φ. If we use only those sets of formulas that are complete w.r.t φ for the tableau automaton of the formula φ, then it has a unique run for every state sequence. Note that this does not mean that the automaton is deterministic, the automaton is able to follow different paths, but only one of these will ultimately be accepting.
Completeness and local consistency constrain the sets of formulas that will be used as locations of the tableau automaton. Besides local constraints, there are also constraints on the sets of formulas that are required at subsequent moments of a state sequence. These constraints relate to the temporal operators U and V. To define these constraints we need the notion of non-trivial (and trivial) sets.

7.1.4 DEFINITION *a set* Φ *containing an Until or Release formula* ψ *is called* non-trivial for ψ

- *if* $\psi_2 \notin \Phi$ *in case* $\psi = \psi_1 \mathsf{U} \psi_2$ *and*

- *if* $\psi_1 \notin \Phi$ *in case* $\psi = \psi_1 \mathsf{V} \psi_2$. □

As shorthand notation, we say that $\Phi \subseteq cl(\varphi)$ is a ψ set if $\psi \in \Phi$. We call a ψ set trivial for ψ if it is not non-trivial for ψ. Sets of formulas are non-trivial for some Until or Release formula if the truth value of the Until formula cannot be locally determined from the truth values of its subformulas. Non-trivial sets will play an important role in the tableau constructions, because they pose constraints on the remainder of the state sequence.

7.1.5 DEFINITION *A set* Φ' *of formulas is a* temporally consistent successor *of the set* Φ *of formulas if for every Until formula* $\psi = \psi_1 \mathsf{U} \psi_2$ *and every Release formula* $\psi = \psi_1 \mathsf{V} \psi_2$, Φ *is non-trivial for* ψ *implies that* $\psi \in \Phi'$. *The fact that* Φ' *is a temporally consistent successor of* Φ *is denoted as* $\Phi \to \Phi'$. □

Another way to look at temporal consistency, is to say that for Φ' to be a temporally consistent successor of Φ, it must contain at least certain formulas that are determined by Φ. This is captured by the following definition.

7.1.6 DEFINITION *Let* Φ *be a set of formulas, then the set* $\mathrm{Next}(\Phi)$ *of* temporal consistency constraints *is the smallest set such that for any Until or Release formula* ψ, *if* Φ *is non-trivial for* ψ *then* $\mathrm{Next}(\Phi)$ *contains* ψ. □

An alternative definition for temporal consistency is then ($\Phi, \Phi' \subseteq$ LTL):

$$\Phi \to \Phi' \text{ iff } \mathrm{Next}(\Phi) \subseteq \Phi'.$$

Now the complete tableau automaton can be defined.

7.1.7 DEFINITION *The* complete tableau automaton A_φ *of an* LTL *formula* φ *is the* ω*-automaton* $(L, \Sigma, L_0, Q, E, F)$ *where*

- $L \subseteq 2^{cl(\varphi)}$ *consists of all sets of formulas that are locally consistent, and complete w.r.t.* φ;
- $\Sigma = 2^{Prop(\varphi)}$;
- L_0 *is the set of all* $\Phi \in L$ *such that* $\varphi \in \Phi$;
- Q *is the mapping that assigns to the location* Φ, *the set of all states* $\sigma \in 2^{Prop}$ *such that for all* $p \in Prop$
  - $p \in \sigma$ *if* $p \in \Phi$,
  - $p \notin \sigma$ *if* $\neg p \in \Phi$;
- E *is the set of all edges* (Φ, Φ') *such that* $\Phi \to \Phi'$;
- F *is the set which contains for every Until formula* $\psi = \psi_1 \mathsf{U} \psi_2$ *in* $cl(\varphi)$, *the set* $f_\psi = \{\Phi \in L \mid \psi \in \Phi \Rightarrow \psi_2 \in \Phi\}$ *and nothing else, i.e. precisely those locations that are not non-trivial for* ψ, *either because they are not* ψ *sets at all or because they are trivial for* ψ. □

Notice that in the definition of the mapping Q, we use $p \in \sigma$ if $p \in \Phi$ (rather than if and only if which would have been equivalent) in order to reuse the definition in the case that locations are not necessarily complete in section 7.2.1. Notice moreover that for every location ℓ in the complete tableau automaton, $Q(\ell)$ contains a single state (over $Prop(\varphi)$, the atomic propositions occurring in φ). The trouble of labelling locations with sets of formulas will pay off however when we consider locations that are not complete. A construction of complete and consistent sets is given in [122].

7.1.3 Example

As an example, we construct the complete tableau automaton of the formula $\varphi = p\mathsf{U}q$, the result of which is depicted in figure 7.3. The subformulas of φ are the following: $\{p\mathsf{U}q, p, q\}$. The closure set $cl(\varphi)$ including also their negations is

$$\{p\mathsf{U}q, \neg(p\mathsf{U}q), p, \neg p, q, \neg q\}.$$

Inspecting all the subsets of $cl(\varphi)$, we obtain the following locally consistent and complete subsets which will form the locations of the automaton:

1. $\{p, q, p\mathsf{U}q\}$,
2. $\{p, \neg q, p\mathsf{U}q\}$,
3. $\{p, \neg q, \neg(p\mathsf{U}q)\}$,
4. $\{\neg p, q, p\mathsf{U}q\}$,
5. $\{\neg p, \neg q, \neg(p\mathsf{U}q)\}$.

We observe that set number 2 is non-trivial for $p\mathsf{U}q$ and set number 3 is non-trivial for $\neg(p\mathsf{U}q)$. We can now determine the temporally consistent pairs of sets (e.g. $\{p, q, p\mathsf{U}q\} \rightarrow \{p, \neg q, \neg(p\mathsf{U}q)\}$ but not $\{p, \neg q, p\mathsf{U}q\} \rightarrow \{\neg p, \neg q, \neg(p\mathsf{U}q)\}$). The initial locations are the sets 1, 2 and 4 since they contain $p\mathsf{U}q$.
With respect to the only Until formula in the closure, the locations 1, 3, 4 and 5 are accepting, only location 2 being non-accepting since it requires that $p\mathsf{U}q$ is true but it does not yet fulfil q. In figure 7.3 the locations of the only acceptance set are designated by a double circle. Initial locations are indicated by a small arrow leading to the location and not originating from another location (locations 1, 2 and 4). Note that since the locations are complete sets with respect to the propositions p and q, the locations are labelled with exactly one state over these propositions. Location 1 for instance, is labelled with state $\{p, q\}$ and location 5 with state $\varnothing$. Figure 7.4 shows how the locations, edges and acceptance conditions of the automaton match the general pattern of figure 7.2. Here the pattern is depicted with dashed lines. Remember that a dashed arrow between two classes of sets indicates that a transition between two such sets exists, unless prohibited by other temporal consistency constraints (if there are multiple Until formulas in the original formula). If there is no dashed arrow between them, such a transition cannot exist. Had there been (unlike in this example) multiple Until formulas in the original formula, then the locations of the tableau could be arranged in the corresponding pattern for every Until subformula separately.

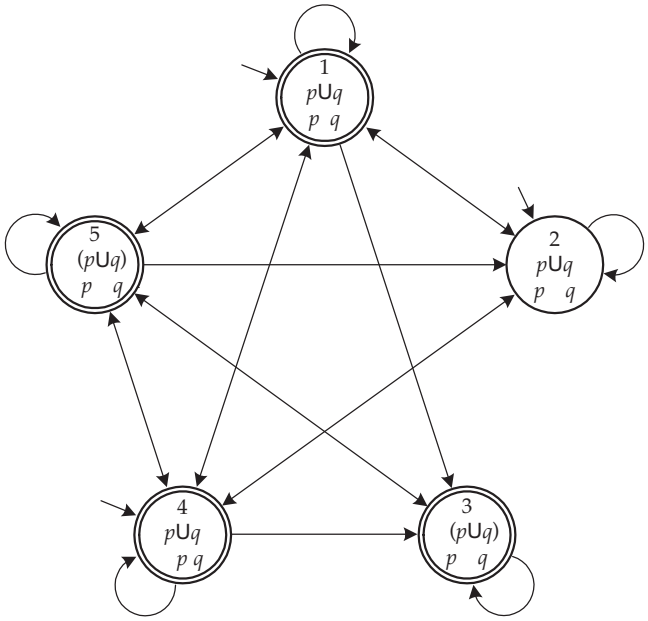


Figure 7.3: Tableau automaton of the formula $p\mathsf{U}q$

7.1.4 Correctness

In this section we prove that the complete tableau automaton A_φ for the LTL formula φ accepts precisely all the state sequences that satisfy φ.

7.1.8 THEOREM *For any* LTL *formula* φ *and its tableau automaton* A_φ, $\mathcal{L}(A_\varphi) = \mathcal{L}(\varphi)$.

This theorem follows immediately from the lemmas 7.1.12 and 7.1.14, which will prove soundness ($\mathcal{L}(A_\varphi) \subseteq \mathcal{L}(\varphi)$) and completeness ($\mathcal{L}(\varphi) \subseteq \mathcal{L}(A_\varphi)$) respectively.

Soundness

Here we show that every state sequence accepted by the tableau automaton of the formula φ satisfies the formula. We show that a run starting from a location labelled with a formula ψ matches only state sequences that satisfy ψ.
The following lemmas show that Until and Release formulas are dealt with correctly by the automaton.

7.1.9 LEMMA *Let the sequence* $\overline{\Psi}$ *of locations be a run on* A_φ *and let* $\psi = \psi_1 \mathsf{U} \psi_2 \in \overline{\Psi}(0)$. *Then there is some* $k \geq 0$, *such that* $\psi_2 \in \overline{\Psi}(k)$ *and for all* $0 \leq m < k$, $\psi_1 \in \overline{\Psi}(m)$.

PROOF For any $i \geq 0$ such that $\psi \in \overline{\Psi}(i)$, either of the following is true

- $\overline{\Psi}(i)$ is trivial for ψ. In this case $\psi_2 \in \overline{\Psi}(i)$.

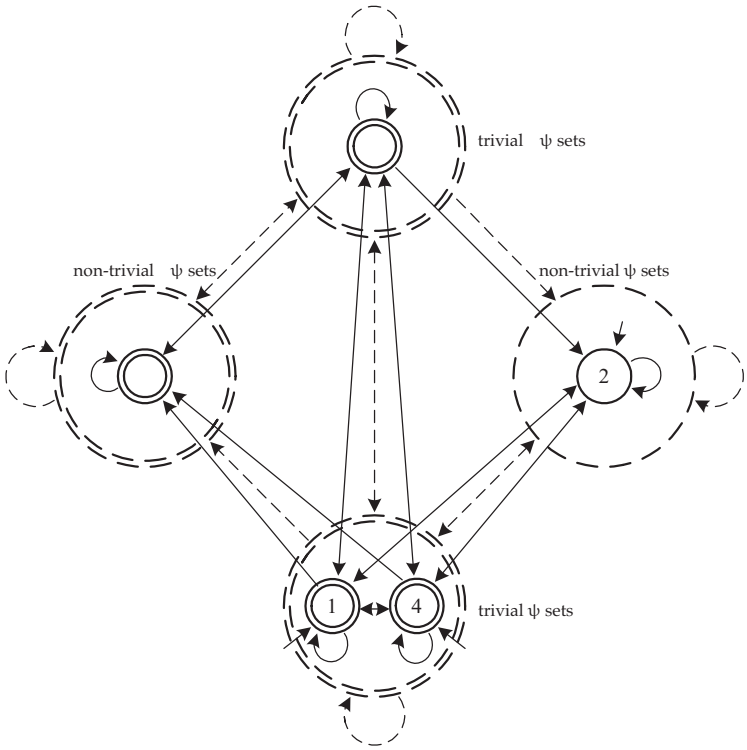


Figure 7.4: Tableau automaton of the formula $\psi = p \mathsf{U} q$ and the general pattern

- $\overline{\Psi}(i)$ is non-trivial for ψ. Then $\psi_1 \in \overline{\Psi}(i)$ by local consistency and $\psi \in \overline{\Psi}(i+1)$ by temporal consistency.

Thus if $\psi \in \overline{\Psi}(i)$ then either $\psi_2 \in \overline{\Psi}(i)$ or $\psi_1 \in \overline{\Psi}(i)$ and $\psi \in \overline{\Psi}(i+1)$. Moreover, since $\overline{\Psi}$ is accepting, there must be some $k \geq 0$ such that $\psi \notin \overline{\Psi}(k)$ or $\psi_2 \in \overline{\Psi}(k)$. Together this shows that there is some $k \geq 0$, such that $\psi_2 \in \overline{\Psi}(k)$ and for all $0 \leq m < k$, $\psi_1 \in \overline{\Psi}(m)$. □

The next lemma states a similar result for Release formulas.

7.1.10 LEMMA *Let $\overline{\Psi}$ be a run on A_φ, let $\psi = \psi_1 \mathsf{V} \psi_2$ and $\psi \in \overline{\Psi}(0)$. Then for all $k \geq 0$, $\psi_2 \in \overline{\Psi}(k)$ or there is some $0 \leq m < k$, such that $\psi_1 \in \overline{\Psi}(m)$.*

PROOF For any $i \geq 0$ such that $\psi \in \overline{\Psi}(i)$, by local consistency, $\psi_2 \in \overline{\Psi}(i)$ and either of the following is true.

- $\overline{\Psi}(i)$ is trivial for ψ. In this case $\psi_2 \in \overline{\Psi}(i)$ and $\psi_1 \in \overline{\Psi}(i)$.
- $\overline{\Psi}(i)$ is non-trivial for ψ. Then $\psi_2 \in \overline{\Psi}(i)$ and by temporal consistency, $\psi \in \overline{\Psi}(i+1)$.

Thus if $\psi \in \overline{\Psi}(i)$ then either $\psi_1 \in \overline{\Psi}(i)$ and $\psi_2 \in \overline{\Psi}(i)$ or $\psi_2 \in \overline{\Psi}(i)$ and $\psi \in \overline{\Psi}(i+1)$. One can therefore show that either for all $k \geq 0$, $\psi_2 \in \overline{\Psi}(k)$ or there is some $k \geq 0$, such that $\psi_1 \in \overline{\Psi}(k)$ and for all $0 \leq m \leq k$, $\psi_2 \in \overline{\Psi}(m)$. From this the lemma follows straightforwardly. □

We are now ready to show that whenever a state sequence is accepted, starting from some location Φ, then the state sequence satisfies all the formulas in Φ.

7.1.11 LEMMA *Let $\overline{\Psi}$ be a run of A_φ for $\overline{\sigma}$ and let $\psi \in \overline{\Psi}(0)$. Then $\overline{\sigma} \models \psi$.*

PROOF By induction on the structure of ψ:

- if $\psi = \text{true}$ then $\overline{\sigma} \models \psi$ holds vacuously;
- $\psi = \text{false}$ cannot be in a consistent set;
- if $\psi \in \mathit{Prop}(\varphi)$ and $\psi \in \overline{\Psi}(0)$, then $\psi \in \overline{\sigma}(0)$ since $\psi \in \sigma$ for all $\sigma \in Q(\overline{\Psi}(0))$, and thus $\overline{\sigma} \models \psi$;
- if $\psi = \neg p$ for some $p \in \mathit{Prop}(\varphi)$ and $\psi \in \overline{\Psi}(0)$, then $p \notin \sigma$ for all $\sigma \in Q(\overline{\Psi}(0))$, so $p \notin \overline{\sigma}(0)$ and thus $\overline{\sigma} \models \psi$;
- if ψ is of the form $\psi_1 \vee \psi_2$ and $\psi \in \overline{\Psi}(0)$ then by local consistency, ψ_1 or ψ_2 is in $\overline{\Psi}(0)$ and by the induction hypothesis $\overline{\sigma} \models \psi$;
- if ψ is of the form $\psi_1 \wedge \psi_2$ and $\psi \in \overline{\Psi}(0)$ then ψ_1 and ψ_2 are in $\overline{\Psi}(0)$ by local consistency and by the induction hypothesis $\overline{\sigma} \models \psi$;
- if $\psi = \psi_1 \mathsf{U} \psi_2$ then by lemma 7.1.9, there is some $m \geq 0$ such that $\psi_2 \in \overline{\Psi}(m)$ and $\psi_1 \in \overline{\Psi}(n)$ for all $0 \leq n < m$. By the induction hypothesis (by lemma 2.6.3, for any $i \geq 0$, $\overline{\Psi}^i$ is a run on A_φ for $\overline{\sigma}^i$ starting from $\overline{\Psi}(I)$)) it follows that $\overline{\sigma} \models \psi$;
- if $\psi = \psi_1 \mathsf{V} \psi_2$ then by lemma 7.1.10, for every $m \geq 0$ either $\psi_2 \in \overline{\Psi}(m)$ or there is some $0 \leq n < m$ such that $\psi_1 \in \overline{\Psi}(n)$. By the induction hypothesis it follows that $\overline{\sigma} \models \psi$. □

Soundness of the tableau automaton is a direct result of this lemma.

7.1.12 LEMMA *If A_φ accepts the state sequence $\overline{\sigma}$ then $\overline{\sigma} \models \varphi$.*

PROOF Follows from lemma 7.1.11 and the fact that there is some initial run $\overline{\Psi}$ (and thus $\varphi \in \overline{\Psi}(0)$), on A_φ for $\overline{\sigma}$. □

Completeness

In order to show that the tableau automaton A_φ accepts all state sequences that satisfy φ, we will point out the (unique) run which demonstrates this. The construction of this run is given in the following definition.

7.1.13 DEFINITION *Let φ be an* LTL *formula and $\overline{\sigma}$ a state sequence. Then the* φ-fine run $\overline{\Psi}$ for $\overline{\sigma}$, *is the sequence of sets such that $\overline{\Psi}(k) \subseteq cl(\varphi)$ and for all $\psi \in cl(\varphi)$ and $k \geq 0$, $\psi \in \overline{\Psi}(k)$ iff $\overline{\sigma}^k \models \psi$.* □

The φ-fine run for $\overline{\sigma}$ is an accepting run on A_φ and if $\overline{\sigma} \models \varphi$, then it is an initial run as well.

7.1.14 LEMMA *If $\overline{\sigma} \models \varphi$ then A_φ accepts $\overline{\sigma}$.*

PROOF Let $\overline{\Psi}$ be the φ-fine run for $\overline{\sigma}$. Then $\overline{\Psi}$ is an initial run for $\overline{\sigma}$ on A_φ. This follows from the facts that:

- $\overline{\Psi}$ is a sequence of locations of A_φ, since $\overline{\Psi}(k) \subseteq cl(\varphi)$ and $\overline{\Psi}(k)$ is locally consistent and complete w.r.t. φ, by the semantics of LTL and the definition of the Fischer-Ladner closure;
- [Symbol match] for every $k \geq 0$, $\overline{\sigma}(k) \in Q(\overline{\Psi}(k))$. This is obvious since $\overline{\Psi}(k)$ contains precisely the propositions that are true at moment k;
- [Consecution] for every $k \geq 0$ there is an edge $\big(\overline{\Psi}(k), \overline{\Psi}(k+1)\big)$, since $\overline{\Psi}(k) \rightarrow \overline{\Psi}(k+1)$ by the semantics of the logic;
- [Acceptance] from the semantics it follows that whenever for some k, $\overline{\Psi}(k)$ contains an Until formula $\psi = \psi_1 \mathsf{U} \psi_2$ then there is some $m \geq k$, such that $\overline{\Psi}(m)$ contains ψ_2 and thus, for every Until formula ψ in $sub(\varphi)$, $\inf(\overline{\Psi}) \cap f_\psi \neq \varnothing$;
- [Initiality] since $\overline{\sigma} \models \varphi$, $\varphi \in \overline{\Psi}(0)$. Thus $\overline{\Psi}(0) \in L_0$. □

7.2 Other Tableaux

In this section we will look at other tableau methods such as on-the-fly methods and their relationship with the complete tableaux method.

7.2.1 Non-Complete Tableaux

The complete tableau is not always the best to be used for model checking. The number of locations may become very large as the formula gets larger. There have been many improvements to the construction in order to obtain smaller automata. One of the key elements is the observation that a location of the automaton does not need to commit itself to either ψ or $\neg\psi$ for every subformula of φ, i.e. it need not be complete. Moreover, it is not always necessary to add all locations to the automaton. If we drop the requirement that locations of the tableau automaton are complete sets of formulas, we arrive at an automaton requiring neither ψ nor $\neg\psi$ in certain locations. It is shown that such a tableau still accepts precisely the models of φ as long as it still adheres to the consistency constraints. Although the non-complete tableau automaton will generally be much larger than the complete automaton, it is

used as an intermediate step towards the on-the-fly generation of a sufficient part of this automaton that will often be much smaller than the complete tableau.

7.2.1 DEFINITION *The* non-complete tableau automaton B_φ *of* LTL *formula* φ *is the* ω*-automaton* $(L, \Sigma, L_0, Q, E, F)$ *where* L *consists of all (also non-complete) locally consistent sets* $\Phi \subseteq 2^{cl(\varphi)}$ *of formulas and the rest is defined exactly the same as the complete tableau automaton in definition 7.1.7.* □

Note that now $Q(\Phi)$ may contain more than one state or even no state at all. For example, $Q(\varnothing) = 2^{Prop}$ (the location is not labelled with any formulas and thus any state is allowed) and $Q(\{p, \neg p\}) = \varnothing$ (the location is labelled with contradicting propositional constraints; note that local consistency does not prohibit this).
Using these new definitions we can prove lemma 7.1.11 for non-complete automata, showing that they are sound as well.

7.2.2 LEMMA *Let* $\overline{\Psi}$ *be a run on* B_φ *for* $\overline{\sigma}$ *and let* $\psi \in \overline{\Psi}(0)$ *then* $\overline{\sigma} \models \psi$.

The proof of lemma 7.1.11 is valid for non-complete automata as well, since completeness of the locations of A_φ is never used in this proof.
Now we can show that the non-complete tableau automaton accepts exactly the same state sequences as the complete tableau automaton and thus also precisely those that satisfy the corresponding formula φ.

7.2.3 THEOREM *Let* B_φ *be the non-complete tableau automaton of* φ *and* A_φ *the complete tableau automaton, then* $\mathcal{L}(B_\varphi) = \mathcal{L}(A_\varphi)$.

PROOF The two inclusions are shown separately. Let $A_\varphi = (L_A, \Sigma, L_{0,A}, Q_A, E_A, F_A)$ and $B_\varphi = (L_B, \Sigma, L_{0,B}, Q_B, E_B, F_B)$.

- $\mathcal{L}(A_\varphi) \subseteq \mathcal{L}(B_\varphi)$. It can be shown that any run on A_φ is still a run on B_φ, since $L_A \subseteq L_B$, $L_{0,A} \subseteq L_{0,B}$, $E_A \subseteq E_B$, $Q_A(\ell) = Q_B(\ell)$ for all $\ell \in L_A$, and for every $f_\psi \in F_B$ there is some $f'_\psi \in F_A$ such that $f'_\psi \subseteq f_\psi$.
- $\mathcal{L}(B_\varphi) \subseteq \mathcal{L}(A_\varphi)$. Let $\overline{\sigma}$ be a state sequence for which $\overline{\sigma} \in \mathcal{L}(B_\varphi)$, then it follows from lemma 7.2.2 that $\overline{\sigma} \models \varphi$ and thus that $\overline{\sigma} \in \mathcal{L}(A_\varphi)$. □

The entire non-complete tableau automaton of the formula $p \mathsf{U} q$ used in the example of section 7.1.3, has 42 locations and is too large to be shown here.

7.2.2 On-the-fly Tableaux

The purpose of the introduction of non-complete tableaux was to come to smaller tableau automata to be used for model checking. The non-complete tableaux as introduced are larger than the complete tableaux. As a second step however, one can construct a sufficient part of this non-complete automaton 'on-the-fly' [89, 163, 60, 67, 112, 133]. This part will generally be much smaller than the complete automaton. From any location Φ, one can generate 'just enough' successor locations. These successor locations are generated by starting with the minimum set of formulas necessary to be temporally consistent with Φ, i.e. by starting with $\mathrm{Next}(\Phi)$. $\mathrm{Next}(\Phi)$ contains ψ for every Until or Release formula $\psi \in cl(\varphi)$, if Φ is non-trivial for ψ.

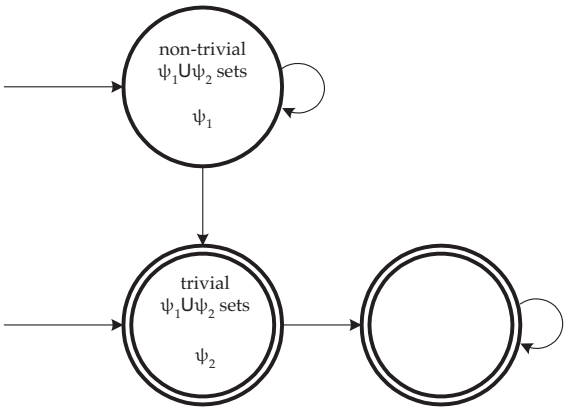


Figure 7.5: General pattern of the on-the-fly tableau for $\psi_1 U \psi_2$

Subsequently, formulas are added and the locations are split until a set of locally consistent locations is obtained that is large enough to guarantee the existence of a run. This leads to automata that are approximately the ones as produced by the algorithm of [89][2].
The reduction of the size of the automaton is achieved since only those formulas that arise from the local consistency constraints of φ and the temporal consistency constraints are propagated; the rest does not matter and need not be present in the locations. This gives fewer combinations of subformulas and thus (often) gives fewer locations.
Checking a formula of the form $\psi = \psi_1 U \psi_2$ for instance proceeds along the lines displayed in figure 7.5. One starts in a ψ set (trivial or non-trivial). The edges of the non-trivial sets lead only to locations containing ψ again. Once the trivial ψ set has been visited, one can forget about ψ. The non-trivial ψ sets are not accepting, the rest are. For Release formulas $\psi = \psi_1 V \psi_2$, a similar structure as in figure 7.6 arises. One starts in a ψ set and remains in ψ sets until a trivial ψ set has been visited, after that one can forget about ψ. Release formulas do not introduce acceptance conditions.

Normal Form Formulas

Another way to look at the construction of the on-the-fly tableau automaton is the following. One can interpret the location as a formula that every state sequence accepted from that location should satisfy. Then the generation of successor locations is guided by rewriting this formula in a specific form that separates constraints on the current state from constraints on the remainder of the state sequence. The constraints on the current states determine the labelling of the location with states and the constraints on the remainder of the sequence determine the outgoing edges. The fact that the formula in this form is equivalent to the original formula essentially guarantees soundness[3] and completeness of the construction.

[2] In contrast with [89], the temporal consistency constraints are collected when a location is finished. This might lead to fewer constraints.

[3] Except for dealing with eventualities of Until formulas.

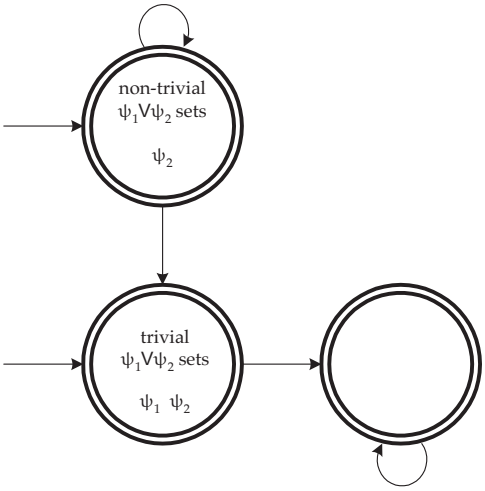


Figure 7.6: General pattern of the on-the-fly tableau for $\psi_1 V \psi_2$

7.2.4 DEFINITION *An* LTL *formula is said to be in* disjunctive temporal normal form *(DTNF) if it is of the form*

$$\begin{aligned}\varphi \quad = \quad & \left(\pi_{1,1} \wedge \pi_{1,2} \wedge \ldots \wedge \pi_{1,n_1} \wedge \bigcirc\left(\varphi_{1,1} \wedge \varphi_{1,2} \wedge \ldots \wedge \varphi_{1,m_1}\right)\right) \vee \\ & \left(\pi_{2,1} \wedge \pi_{2,2} \wedge \ldots \wedge \pi_{2,n_2} \wedge \bigcirc\left(\varphi_{2,1} \wedge \varphi_{2,2} \wedge \ldots \wedge \varphi_{2,m_2}\right)\right) \vee \\ & \ldots \\ & \left(\pi_{k,1} \wedge \pi_{k,2} \wedge \ldots \wedge \pi_{k,n_k} \wedge \bigcirc\left(\varphi_{k,1} \wedge \varphi_{k,2} \wedge \ldots \wedge \varphi_{k,m_k}\right)\right)\end{aligned}$$

where all $\pi_{i,j}$ are atomic propositions or negated atomic propositions and $\varphi_{i,j}$ are LTL *formulas of the form $\psi_1 U \psi_2$ or $\psi_1 V \psi_2$.* □

Any LTL formula can be rewritten into an equivalent formula in DTNF using the following equivalences as rewriting rules from left to right (here, $\psi_1 \equiv \psi_2$ denotes that for any state sequence $\overline{\sigma}$, we have $\overline{\sigma} \models \psi_1$ iff $\overline{\sigma} \models \psi_2$).

- False
  - $\psi \vee \text{false} \equiv \psi$
  - $\psi \wedge \text{false} \equiv \text{false}$

  We say that true and false are in DTNF, identifying true with an empty conjunction and false with an empty disjunction.
- Distribution
  - $\psi_1 \wedge (\psi_2 \vee \psi_3) \equiv (\psi_1 \wedge \psi_2) \vee (\psi_1 \wedge \psi_3)$
  - $(\psi_1 \vee \psi_2) \wedge \psi_3 \equiv (\psi_1 \wedge \psi_3) \vee (\psi_2 \wedge \psi_3)$
  - $\bigcirc\psi_1 \wedge \bigcirc\psi_2 \equiv \bigcirc(\psi_1 \wedge \psi_2)$
  - $\bigcirc(\psi_1 \vee \psi_2) \equiv \bigcirc\psi_1 \vee \bigcirc\psi_2$

- Unfolding
  - $\psi_1 \mathsf{U} \psi_2 \equiv \psi_2 \vee (\psi_1 \wedge \bigcirc \psi_1 \mathsf{U} \psi_2)$
  - $\psi_1 \mathsf{V} \psi_2 \equiv \psi_2 \wedge (\psi_1 \vee \bigcirc \psi_1 \mathsf{V} \psi_2)$

Using these rules one can rewrite any LTL formula into disjunctive normal form [163].

Constructing the Automaton

The automaton is created on-the-fly. This means that from a location of the automaton already constructed, we can incrementally produce the reachable successor locations. To generate the successor locations of a location Φ, we start with the set of formulas needed for any set to be a temporally consistent successor, i.e. the set $\mathrm{Next}(\Phi)$, and use this set to compute (mimicking the normal-form rewrite rules and local consistency rules) a sufficiently large set of temporally consistent successor locations that are locally consistent and all include the formulas $\mathrm{Next}(\Phi)$.
Figure 7.7 shows an example of the operation of the procedure to obtain a set of locally consistent successor locations and the correspondence between this procedure, the normal form and local consistency. The procedure is started from the set $\{(p \mathsf{U} q) \vee r\}$ (graphically represented as a location, being split by the procedure into a collection of locations). In the first step of the procedure, it is recognised that the formula $(p \mathsf{U} q) \vee r$ already consists of a disjunction of two terms, and the set is split in two sets representing those terms respectively (conform local consistency rule 2). The formula is marked with an asterisk (*) to denote that it has already been processed and need not be dealt with again. In a second step, the formula $p \mathsf{U} q$ in the left of the two sets is processed and the location is split into two new locations, one containing p and the other containing q following local consistency rule 4. This corresponds to the rewriting rule $p \mathsf{U} q \equiv q \vee (p \wedge \bigcirc p \mathsf{U} q)$; note that the part $\bigcirc p \mathsf{U} q$ is not explicitly represented in the corresponding location. It is however implicitly represented since the location is non-trivial for $p \mathsf{U} q$ and thus any of its successor locations must contain $p \mathsf{U} q$ again in order to be temporally consistent. In the subsequent steps in the procedure, the atomic propositions in the locations are marked and nothing else changes. Note that the formulas in a location contain the formulas in the corresponding term of the normal form as well as the formulas that have been rewritten to obtain that term.
During the procedure, a set of formulas represents (intuitively, this is not used in the construction) the conjunction of these formulas *and* the consistency constraints arising from the formulas already marked ($\{p \mathsf{U} q, p\}$ represents $(p \mathsf{U} q) \wedge p$, whereas $\{p \mathsf{U} q^*, p\}$ represents $(p \mathsf{U} q) \wedge p \wedge \bigcirc p \mathsf{U} q$). $\bigcirc p \mathsf{U} q$ arises from the fact that $\{p \mathsf{U} q^*, p\}$ is non-trivial for $p \mathsf{U} q$. A set of such sets represents the disjunction of the corresponding conjunctive formulas ($\{\{p\}, \{q, \neg r\}\}$ represents the formula $p \vee (q \wedge \neg r)$).
We also say (again informally) that the set $\{\{p \mathsf{U} q, q\}, \{p \mathsf{U} q, p\}\}$ represents the DTNF formula $q \vee (p \wedge \bigcirc p \mathsf{U} q)$ without the $p \mathsf{U} q$ formulas that are present in the set, since these are in fact redundant, which can be seen from the reduction to normal form. In the remainder we let Ψ range over sets of (partially) marked formulas and Φ range over ordinary sets of formulas without markings. The actual procedure is given in the following definition.

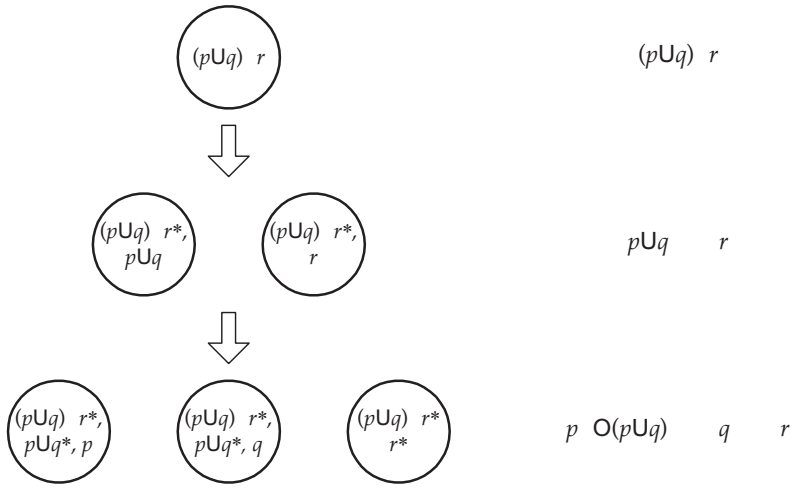


Figure 7.7: Normal form procedure for $(p\mathsf{U}q) \vee r$

	Case	$P \cup \{\Psi \cup \{\psi\}\}$ reduces to:
1	$\psi = \text{false}$	P
2	$\psi = \text{true}$	$P \cup \{\Psi \cup \{\psi^*\}\}$
3	$\psi = p$	$P \cup \{\Psi \cup \{\psi^*\}\}$
4	$\psi = \neg p$	$P \cup \{\Psi \cup \{\psi^*\}\}$
5	$\psi = \psi_1 \vee \psi_2$	$P \cup \{\Psi \cup \{\psi^*, \psi_1\}, \Psi \cup \{\psi^*, \psi_2\}\}$
6	$\psi = \psi_1 \wedge \psi_2$	$P \cup \{\Psi \cup \{\psi^*, \psi_1, \psi_2\}\}$
7	$\psi = \psi_1 \mathsf{U} \psi_2$	$P \cup \{\Psi \cup \{\psi^*, \psi_1\}, \Psi \cup \{\psi^*, \psi_2\}\}$
8	$\psi = \psi_1 \mathsf{V} \psi_2$	$P \cup \{\Psi \cup \{\psi^*, \psi_1, \psi_2\}, \Psi \cup \{\psi^*, \psi_2\}\}$

Table 7.1: Local consistency procedure. If an unmarked formula is added to a set already containing that formula, the formula will be unmarked in the new set. We assume that $\psi \notin \Psi$ and $\Psi \cup \{\psi\} \notin P$

7.2.5 DEFINITION *The* local consistency procedure *that generates a set of locally consistent sets from a set Ψ of formulas is defined as follows.*

- *Initially, let $P_0 = \{\Psi\}$.*
- *Then, as long as P_n contains a set with an unmarked formula, use one of the reduction rules of table 7.1 to generate a new set P_{n+1}.* □

Note that the result of the procedure may depend on the order in which the formulas are selected. Although results may differ, they are all in normal form and represent equivalent formulas[4]. We assume in the remainder that NF is some fixed *deterministic* procedure operating in accordance with the given procedure. We use $\mathrm{NF}(\Phi)$ to denote a result of the normalisation procedure performed on the set Φ and $\mathrm{NF}(\varphi)$ to denote a result of the procedure performed on the set $\{\varphi\}$.

[4]It makes sense to carefully pick the formula to unfold. In particular to deal with largest formulas first, to avoid handling the same formula twice.

```
New := NF(φ), L := ∅, L_0 := New, Edges := ∅
while New ≠ ∅ do
  Let Φ ∈ New
  New := New\{Φ}
  L := L ∪ {Φ}
  for every Φ' ∈ NF(Next(Φ)) do
    Edges := Edges ∪ {(Φ, Φ')}
    if Φ' ∉ L then New := New ∪ {Φ'}
  od
od
```

Figure 7.8: Algorithm for constructing locations and edges of the on-the-fly tableau automaton

The edges and locations of the on-the-fly tableau are constructed by the algorithm presented in figure 7.8. After that, the labelling of the locations and the acceptance sets are determined in the same way as for complete tableau automata.

Example

As a small example we construct the on-the-fly tableau of the formula $p\mathsf{U}q$, the result of which is displayed in figure 7.1. The initial locations are generated by starting the normal form procedure from $\{\{p\mathsf{U}q\}\}$. The result is $\{\{p\mathsf{U}q^*, p^*\}, \{p\mathsf{U}q^*, q^*\}\}$, which corresponds to the DTNF representation $p\mathsf{U}q \equiv (p \wedge \bigcirc p\mathsf{U}q) \vee q$. This leads to two initial locations $\{p\mathsf{U}q, p\}$ and $\{p\mathsf{U}q, q\}$. The temporal consistency constraints of the first initial location are $\{p\mathsf{U}q\}$, thus the outgoing edges of the first initial location lead to the same two locations $\{p\mathsf{U}q, p\}$ and $\{p\mathsf{U}q, q\}$. The temporal consistency constraints of the second initial location are $\varnothing$. Therefore that location has an outgoing edge to a new location $\varnothing$ ($\mathrm{NF}(\{\varnothing\}) = \{\varnothing\}$). This new location also has the temporal consistency constraints $\varnothing$ and thus has an edge to itself.
The symbol labelling is such that the first location matches all states that satisfy p, the second location matches the states that satisfy q and the third location matches all states. Finally, the acceptance conditions have to be determined. There is one Until formula $p\mathsf{U}q$ and thus one acceptance set. Only the first location is non-trivial for $p\mathsf{U}q$ and is thus not accepting.

Correctness of the On-the-Fly Tableau

Correctness of the on-the-fly algorithm is stated in the following theorem.

7.2.6 THEOREM *Let A_φ be the on-the-fly tableau automaton of φ, then $\mathcal{L}(A_\varphi) = \mathcal{L}(\varphi)$.*

Soundness and completeness of the on-the-fly tableau are demonstrated in the lemmas 7.2.8 and 7.2.11 that follow. Since the on-the-fly automaton is a subset of the entire non-complete automaton it is obviously still sound (i.e. any accepted state sequence satisfies φ). Completeness follows from the way the successor locations have

been created. Whenever a state sequence satisfies all formulas in a location, then there is an appropriate successor location, labelled with formulas that are true for the tail of the sequence. Furthermore this successor location can be chosen such that all acceptance conditions are met. We use some shorthand notation. If Φ is a set of formulas, we write $\overline{\sigma} \models \Phi$ instead of 'for all $\psi \in \Phi$, $\overline{\sigma} \models \psi$'. This allows us to write things like: $\overline{\sigma} \models \mathrm{Next}(\Phi)$.

Soundness

Demonstrating soundness of the automaton amounts to showing that the generated locations and edges satisfy the local and temporal consistency constraints. This demonstrates that the on-the-fly tableau is a sub-automaton of the non-complete tableau and as such it must also be sound.

7.2.7 LEMMA *Let Φ be a set of formulas and let $\Phi' \in \mathrm{NF}(\Phi)$. Then Φ' is locally consistent. Moreover, $\Phi \rightarrow \Phi'$ for any $\Phi' \in \mathrm{NF}(\mathrm{Next}(\Phi))$.*

For the proof, see appendix B.1.1. The initial locations of the automaton are created by computing the normal form of φ, i.e. $\mathrm{NF}(\varphi)$.

7.2.8 LEMMA *Let A_φ be the on-the-fly tableau automaton of φ, then $L(A_\varphi) \subseteq L(\varphi)$.*

PROOF Let B_φ be the non-complete tableau of φ, and let $\overline{\sigma}$ be a state sequence that satisfies φ. It follows from lemma 7.2.7 and the construction of the automaton that an initial run on A_φ matching $\overline{\sigma}$ is also an initial run on B_φ and thus from the soundness of B_φ it follows that $\overline{\sigma} \models \varphi$. □

Completeness

We demonstrate that the procedure generates enough successor locations, by showing that if we have a set Φ of formulas and a state sequence $\overline{\sigma}$ that satisfies these formulas, then there is at least one set in $\mathrm{NF}(\Phi)$ of which all formulas are satisfied by $\overline{\sigma}$. In terms of the DTNF this corresponds to the claim that if $\overline{\sigma}$ satisfies a formula φ then there is a disjunctive term in the DTNF of φ that is satisfied by $\overline{\sigma}$.
If a state sequence satisfies the temporal consistency constraints $\mathrm{Next}(\Phi)$ of a location Φ of the on-the-fly tableau, then a suitable edge exists to show the acceptance of the state sequence. This is expressed in the next two lemmas.

7.2.9 LEMMA *Let Φ be a set of formulas. Let $\overline{\sigma} \models \psi$ for all $\psi \in \Phi$. Then there is some $\Phi' \in \mathrm{NF}(\Phi)$ such that (i) $\overline{\sigma} \models \psi$ for all $\psi \in \Phi'$, (ii) $\overline{\sigma}^1 \models \mathrm{Next}(\Phi')$ and (iii) for every Until formula $\psi_1 \mathrm{U} \psi_2$ in Φ' such that $\overline{\sigma} \models \psi_2$, we have $\psi_2 \in \Phi'$.*

The proof is given in appendix B.1.2. The following lemma shows how to incrementally construct an accepting run for a state sequence that satisfies the formulas in a location using the previous lemma.

7.2.10 LEMMA *Let Φ be a location in the on-the-fly tableau automaton A_φ, let $\overline{\sigma} \models \psi$ for all $\psi \in \mathrm{Next}(\Phi)$. Then there is some edge (Φ, Φ') of A_φ such that (i) $\overline{\sigma} \models \psi$ for all $\psi \in \Phi'$, (ii) $\overline{\sigma}^1 \models \mathrm{Next}(\Phi')$ and (iii) for every Until formula $\psi = \psi_1 \mathrm{U} \psi_2 \in sub(\varphi)$, $\Phi' \in F_\psi$ if $\overline{\sigma} \models \psi_2$.*

PROOF It follows from lemma 7.2.9 and the construction of the automaton that there is an edge to such a location Φ'. (i) and (ii) follow immediately and if $\overline{\sigma} \models \psi_2$ and $\psi \in \Phi'$, then $\psi_2 \in \Phi'$ and hence, $\Phi' \in F_\psi$. □

Together, this shows that the on-the-fly tableau automaton is complete.

7.2.11 LEMMA *Let A_φ be the on-the-fly tableau automaton of φ. Then $\mathcal{L}(\varphi) \subseteq \mathcal{L}(A_\varphi)$.*

PROOF Let $\overline{\sigma}$ be a state sequence and $\overline{\sigma} \models \varphi$. It follows from lemmas 7.2.9 and 7.2.10 that there is an accepting run for $\overline{\sigma}$ on A_φ. Lemma 7.2.9 guarantees the existence of a suitable initial location and lemma 7.2.10 can be repeatedly applied to construct the accepting run. □

Concluding Remarks

Some improvements (in size) on this on-the-fly automaton are still possible. For example (as it is mentioned in [89] and [60]) not all labels in the generated locations are relevant after the construction of the new locations is finished. For instance, if in a location p is required to hold, then the formula $p \vee q$ holds as well, and therefore there is no need to discriminate the locations $\{p\}$ and $\{p, p \vee q\}$. If these irrelevant requirements are removed, some locations might turn out to be identical. The only relevant labels are propositional labels, Until or Release formulas and formulas that occur as right-hand sides of Until formulas (since they are needed to define the acceptance conditions). Another observation is that it is best to pick maximal unmarked formulas during the normal form procedure, i.e. formulas that are not a subformula of another unmarked formula. This prevents one from processing the same formula twice, possibly resulting in more terms in the normal form.
It might also be useful to prevent the splitting of entirely propositional formulas. For example if the formula $p \vee q \wedge r$ occurs, it is not necessary to split it into two terms leading to two locations. Only the 'temporal part' of the formula needs to be changed to normal form. Whether this would lead to smaller automata remains to be seen. This strategy will be used in section 7.3 for the construction of deterministic tableaux. Other strategies to obtain smaller automata include preprocessing of the original formula according to some set of rewriting rules in order to obtain simpler formulas [112, 166, 73] and postprocessing of the obtained automaton in order to reduce the number of locations or acceptance sets [166, 73]. Table 7.2 displays a few example formulas and the sizes of the corresponding tableaux for different construction methods.
If the number of locations is the paramount criterium, we can also construct a tableau automaton in which edges rather than locations are labelled with symbols. We observe from the use of temporal consistency requirements that the only 'memory' required by the automaton resides in the Until and Release formulas that need to be propagated. Thus, locations only need to be labelled with Until and Release formulas. We do not give the construction of such an automaton. It is relatively straightforward to obtain it from our on-the-fly automata, letting the transitions take the role of our locations and use locations to represent the temporal consistency constraints. For example, the edge-labelled automaton of figure 7.9 is obtained in this way from the tableau of figure 7.1. Along the edges we have written the entire labelling of

φ	Non-complete	Complete	On-the-fly	$\lvert 2^{cl(\varphi)} \rvert$
$p \mathsf{U} q$	42 / 1440	5 / 20	3 / 4	64
$(p \mathsf{U} q) \mathsf{U} (q \mathsf{U} p)$	264 / 41664	7 / 28	12 / 24	1024
$p \mathsf{U} (q \mathsf{U} r)$	384 / 98176	12 / 96	6 / 10	1024
$(p \mathsf{U} q) \mathsf{U} r$	432 / 118560	13 / 104	8 / 15	1024
$p \mathsf{U} (q \vee r \vee s)$	1040 / 941248	17 / 272	5 / 8	4096
$p \mathsf{U} (q \wedge r \wedge s)$	1232 / 1104640	23 / 368	3 / 4	4096
$\Box (p \Rightarrow (q \mathsf{U} r))$	1496 / 1174848	18 / 144	5 / 17	16384
$(p \mathsf{U} q) \mathsf{U} (r \mathsf{U} s)$	3936 / 8256000	31 / 496	17 / 38	16384

Table 7.2: Numbers of states / transitions of different tableaux

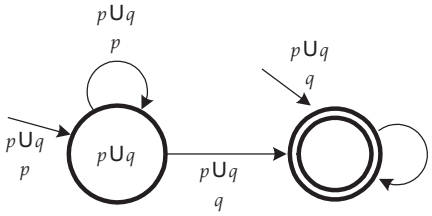


Figure 7.9: On-the-fly tableau automaton of the formula $p \mathsf{U} q$ with labels on edges

the corresponding locations in the original automaton. In order to accept a state sequence, the states must satisfy the propositional formulas along the edges.

7.3 Efficient Fragments of LTL

In this section we will discuss a fragment of LTL that can be translated into a deterministic automaton and the dual fragment which also has an efficient tableau construction. With a deterministic tableau automaton, the tableau automaton's transition is completely determined from the corresponding transition of the system and thus the number of transitions of the combined location of the product automaton depends only on the number of outgoing edges of the location of the system. This leads to less backtracking and possibly fewer locations during model-checking. The dual fragment is also interesting. It can be translated into an automaton the size of which grows only quadratically with the length of the formula (and linearly if one constructs edge-labelled automata).

7.3.1 Modified Syntax and Semantics of LTL

To be able to define the fragments of LTL we are interested in, we redefine the syntax of LTL in terms of two layers, propositional logic and temporal operators. Neither the syntax nor the semantics of LTL formulas essentially changes. Propositional logic is interpreted on states σ over a set *Prop* of atomic propositions.

7.3.1 DEFINITION *The* syntax of PL *is defined by the following grammar, where* $p \in Prop$

$$\pi ::= \text{true} \mid p \mid \neg\pi \mid \pi_1 \vee \pi_2$$

Formulas of PL will be ranged over by π.
The semantics of PL is standard and is not repeated here. We also use false and $\wedge$ to write PL formulas in positive form. In the syntax of LTL, atomic propositions are replaced by propositional formulas.

7.3.2 DEFINITION *The* syntax of LTL *is defined by the following grammar*

$$\varphi ::= \pi \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi' \mid \varphi_1 \mathsf{U} \varphi_2$$

The semantics of LTL formulas remains the same. Propositional formulas are interpreted in the first state of the sequence:

$$\overline{\sigma} \models \pi \text{ iff } \overline{\sigma}(0) \models \pi.$$

Note that the presented grammar is ambiguous (more so than the original syntax was already). The semantics is invariant under the different interpretations of a formula, we will assume that the common operators $\neg$ and $\vee$ are interpreted as operators of PL whenever possible. We adjust the definition of the normal form accordingly.

7.3.3 DEFINITION *An* LTL *formula is said to be in* disjunctive temporal normal form *if it is of the form*

$$(\pi_1 \wedge \bigcirc\varphi_1) \vee (\pi_2 \wedge \bigcirc\varphi_2) \vee \ldots (\pi_k \wedge \bigcirc\varphi_k)$$

where all π_i *are* PL *formulas and* φ_i *are* LTL *formulas.*

Note that in contrast with traditional definitions this normal form does not require disjunctions to be removed from PL formulas and therefore computes the tableau "modulo PL". For application to LTL model checking this is good enough since the PL formulas can be evaluated in the current system state to determine the next locations and it allows the proper definition of the acceptance conditions as well.

7.3.2 A Deterministic Fragment of LTL

We now define a fragment of LTL and a way to rewrite formulas in this fragment into a form that will lead to deterministic tableau automata. Certain types of LTL formulas allow an adaptation to the normal form procedure that results in tableau automata having deterministic transitions. This means that for any location of the automaton and any state, there is at most one outgoing edge leading to a location labelled with that state. In terms of the labelling of locations with propositional formulas, this means that those formulas are mutually exclusive, if one outgoing edge leads to a location labelled with π_1 and another to a location labelled with π_2, then $\pi_1 \wedge \pi_2$ is not satisfiable. This gives us deterministic tableau automata without an extra exponential blowup in size that would be the result of a determinisation afterwards, if possible at all.

	Case	$P \cup \{\Psi \cup \{\psi\}\}$ reduces to:
1	$\psi = \pi$	$P \cup \{\Psi \cup \{\psi^*\}\}$
2	$\psi = \pi \vee \psi'$	$P \cup \{\Psi \cup \{\psi^*, \pi\}, \Psi \cup \{\psi^*, \neg\pi, \psi'\}\}$
3	$\psi = \psi_1 \wedge \psi_2$	$P \cup \{\Psi \cup \{\psi^*, \psi_1, \psi_2\}\}$
4	$\psi = \psi' \mathsf{U} \pi$	$P \cup \{\Psi \cup \{\psi^*, \pi\}, \Psi \cup \{\psi^*, \neg\pi, \psi'\}\}$
5	$\psi = \pi \mathsf{V} \psi'$	$P \cup \{\Psi \cup \{\psi^*, \pi, \psi'\}, \Psi \cup \{\psi^*, \neg\pi, \psi'\}\}$

Table 7.3: Deterministic local consistency procedure

7.3.4 DEFINITION *The* syntax of deterministic LTL *is defined by the following grammar*

$$\varphi ::= \pi \mid \pi \vee \varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc \varphi \mid \varphi \mathsf{U} \pi \mid \pi \mathsf{V} \varphi.$$

Note that the syntax is given in positive form. The Until and Release operators are restricted to have only propositional formulas on one side. Multiple transitions from a location of the tableau automaton arise from the $\vee$ operator in the normal form. The transitions can be non-deterministic if the propositional formulas in the terms of the normal form are not mutually exclusive. We use rewriting rules for the presented fragment that will lead to mutually exclusive terms and thus to deterministic automata.
The normal form procedure to determine the next locations is modified such that the choice for the next location is deterministic. This is achieved by using the following equivalences as rewrite rules from left to right.

- $\pi \vee \varphi \equiv \pi \vee (\neg\pi \wedge \varphi)$
- $\varphi \mathsf{U} \pi \equiv \pi \vee (\neg\pi \wedge \varphi \wedge \bigcirc \varphi \mathsf{U} \pi)$
- $\pi \mathsf{V} \varphi \equiv \varphi \wedge (\pi \vee (\neg\pi \wedge \bigcirc \pi \mathsf{V} \varphi))$

The corresponding reductions to be used in the algorithm of definition 7.2.5 are given in table 7.3.
For example, the formula $\varphi = (p \mathsf{U} q) \mathsf{U} r$ can be rewritten as follows.

$$\begin{aligned} & (p \mathsf{U} q) \mathsf{U} r \\ \equiv\ & r \vee \neg r \wedge ((p \mathsf{U} q) \wedge \bigcirc \varphi) \\ \equiv\ & r \vee \neg r \wedge ((q \vee \neg q \wedge p \wedge \bigcirc (p \mathsf{U} q)) \wedge \bigcirc \varphi) \\ \equiv\ & r \vee (\neg r \wedge q \wedge \bigcirc \varphi) \vee (\neg r \wedge \neg q \wedge p \wedge \bigcirc (p \mathsf{U} q \wedge \varphi)). \end{aligned}$$

The propositional parts of the terms of this formula are mutually exclusive. The entire deterministic tableau of this formula is shown in figure 7.10. The three terms in the normal form above manifest themselves in the three initial locations of the automaton. In this example there are two acceptance sets, one for the entire formula and one for the $p \mathsf{U} q$ subformula. The location belonging to the former are indicated using bold lines and the locations belonging to the latter are represented using a double circle.
It is straightforward to demonstrate that the terms of the normal form are mutually exclusive, by induction on the procedure of construction. Whenever a set is split, the new sets are mutually exclusive (and still mutually exclusive to the rest of the sets as well).

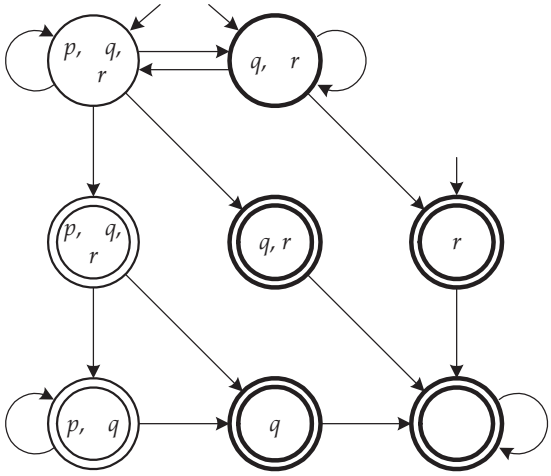


Figure 7.10: Deterministic tableau of the formula $(p\mathsf{U}q)\mathsf{U}r$

7.3.3 Quadratic / Linear Fragment

Another interesting fragment of LTL arises if one changes the restrictions in the following ways. It allows only conjunction with propositional formulas (compared to disjunctions in the previous fragment). It allows on the left side of Until and the right side of Release formulas only propositional formulas (as opposed to right and left respectively in the previous fragment).

7.3.5 DEFINITION *The* syntax of linear LTL *is defined by the following grammar*

$$\varphi ::= \pi \mid \varphi_1 \vee \varphi_2 \mid \pi \wedge \varphi \mid \bigcirc \varphi' \mid \pi \mathsf{U} \varphi \mid \varphi \mathsf{V} \pi.$$

Note that this fragment is not determinisable, since it includes the formula $\Diamond\Box p$, which is known to have no corresponding deterministic Büchi automaton [171]. If we apply the normal form rewriting rules, we observe that there will be at most one Until or Release operator behind the Next operator of any term. For example after the rewriting of

$$\pi \mathsf{U} \varphi \equiv \varphi \vee (\pi \wedge \bigcirc \pi \mathsf{U} \varphi)$$

the Next operator arising in the right term has only one Until operator and it is already in normal form. The same is true for the unfolding of the Release operator.

$$\varphi \mathsf{V} \pi \equiv (\pi \wedge \varphi) \vee (\pi \wedge \bigcirc \varphi \mathsf{V} \pi)$$

Here, the right term is immediately in normal form.
The corresponding reductions to be used in the algorithm of definition 7.2.5 are given in table 7.4. In fact they are exactly the same as in the original algorithm, except for the use of propositional formulas rather than atomic propositions. Note that a partially marked set Ψ contains, at any point during the construction of the normal form, at most one unmarked formula. It is started with only a single formula and at every reduction case in table 7.4, the unmarked formula ψ is marked and in every

	Case	$P \cup \{\Psi \cup \{\psi\}\}$ reduces to:
1	$\psi = \pi$	$P \cup \{\Psi \cup \{\psi^*\}\}$
2	$\psi = \psi_1 \vee \psi_2$	$P \cup \{\Psi \cup \{\psi^*, \psi_1\}, \Psi \cup \{\psi^*, \psi_2\}\}$
3	$\psi = \pi \wedge \psi'$	$P \cup \{\Psi \cup \{\psi^*, \pi^*, \psi'\}\}$
4	$\psi = \pi \mathsf{U} \psi'$	$P \cup \{\Psi \cup \{\psi^*, \psi'\}, \Psi \cup \{\psi^*, \pi^*\}\}$
5	$\psi = \psi' \mathsf{V} \pi'$	$P \cup \{\Psi \cup \{\psi^*, \pi^*, \psi'\}, \Psi \cup \{\pi^*, \psi'\}\}$

Table 7.4: Quadratic local consistency procedure

set added to P, there is at most one unmarked formula. Furthermore the size of the unmarked formula decreases with every step. The number of terms increases by at most one in every step and if a term is split in two, then so is the size of the respective unmarked formulas.
To derive a bound on the number of generated locations, we define a measure of the size of the set P. Let

$$|P_k| = \sum_{\Psi \in P_k} |\Psi|$$

where

$$|\Psi| = \begin{cases} 1 & \text{if } \Psi \text{ has no unmarked formula} \\ |\psi| & \text{if the unmarked formula in } \Psi \text{ is } \psi. \end{cases}$$

Then it follows from the rules of the rewriting procedure that $|P_{k+1}| \leq |P_k|$. After the procedure is finished, $|P_k|$ is the number of terms in the normal form and $|P_0| = |\psi|$, thus the number of terms in $P(\psi)$ is at most $|\psi|$.
It is easy to show that for every term Ψ in $P(\psi)$, there is at most one subformula of ψ in $\mathrm{Next}(\Psi)$ and thus there are at most $|\psi|$ different invocations of the normal form procedure during a construction of an on-the-fly tableau. Thus, the number of locations of the tableau automaton of the formula φ is bounded by $|\varphi|^2$. In fact, the number of locations is linear in the size of φ if we label edges rather than locations. The locations then only store consistency constraints and every location contains at most one of those.

7.4 Automata for Prefixes

In chapter 6 a methodology for the verification of safety properties during simulations was discussed. To this end, we needed so-called automata for bad prefixes, i.e. automata on finite words that recognise prefixes of state sequences that cannot be extended into state sequences that satisfy the property. In this section we discuss how the tableau method can be adapted to this purpose. First, we define the notions of good prefix and bad prefix.

7.4.1 DEFINITION *[123] A finite word $u \in \Sigma^*$ is called a* good prefix for the language $\mathcal{L} \subseteq \Sigma^\omega$ *iff for every infinite word $\bar{w} \in \Sigma^\omega$, $u \cdot \bar{w} \in \mathcal{L}$.*

Similarly, a *bad prefix for* $\mathcal{L}$ is a prefix of which no extension is a member of the language $\mathcal{L}$. Thus there are also prefixes that are neither good not bad.

For prefixes of state sequences and properties expressed in LTL, we use the notion of an *informative* good (bad) prefix from [123]. Informative refers to the fact that it is possible to demonstrate, by means of the semantics of LTL, why this prefix must be a good (bad) prefix for the property defined by the formula. We use $\overline{\xi}$ to range over finite state sequences.

7.4.2 DEFINITION *[123] Let $\overline{\xi}$ be a prefix of a state sequence. $\overline{\xi}$ is informative for φ iff there exists a finite sequence $\overline{IS} \in (2^{\mathrm{LTL}})^*$, say of length $n+1 \leq |\overline{\xi}|+1$, such that* [5]

- $\varphi \in \overline{IS}(0)$;
- $\overline{IS}(n) = \varnothing$;
- *for all* $0 \leq i < n$ *and* $\psi \in \overline{IS}(i)$,
  - *if ψ is a propositional formula, then $\overline{\xi}(i) \models \psi$;*
  - *$\overline{IS}(i)$ is locally consistent;*
  - *$\overline{IS}(i+1)$ is a temporally consistent successor of $\overline{IS}(i)$.* □

We call such a sequence $\overline{IS}$ an *informative sequence*. If such an informative sequence exists, it tells us why φ holds for any extension of the prefix $\overline{\xi}$. It indicates what formulas hold at what moment of the prefix and why. Since $\overline{IS}(i)$ is at some point empty, this reasoning is complete and thus applies to any extension of the prefix. For instance, if $\psi_1 \wedge \psi_2 \in \overline{IS}(i)$, then by the consistency requirements, both $\psi_1 \in \overline{IS}(i)$ and $\psi_2 \in \overline{IS}(i)$, which tells us that $\psi_1 \wedge \psi_2$ holds for any extension of $\overline{\xi}^i$ (the part of $\overline{\xi}$ from state i to the end) since both ψ_1 and ψ_2 hold for any extension of $\overline{\xi}^i$. If $\psi_1 \mathsf{U} \psi_2 \in \overline{IS}(i)$, $\psi_1 \in \overline{IS}(i)$, and $\psi_2 \notin \overline{IS}(i)$, then according to temporal consistency, $\psi_1 \mathsf{U} \psi_2 \in \overline{IS}(i+1)$. This signifies that $\psi_1 \mathsf{U} \psi_2$ must hold for any extension of $\overline{\xi}^i$, because ψ_1 holds for any extension of $\overline{\xi}^i$ and $\psi_1 \mathsf{U} \psi_2$ holds for any extension of $\overline{\xi}^{i+1}$. Since $\overline{IS}(n) = \varnothing$, such a reasoning is complete and "tells the whole story"[123].

It is possible to effectively construct an automaton on finite words that accepts all bad prefixes for a given formula. We concentrate, however, on automata that recognise informative prefixes only, for two reasons. Firstly, the construction of automata for all bad prefixes is doubly exponential in the length of the formula, whereas the construction of automata for informative prefixes is only singly exponential [123]. Secondly, the informative bad prefixes can be considered as the only proper counterexamples, since they demonstrate why the formula does not hold. Other bad prefixes depend on some peculiarity of the formula. For example, if ψ is a formula that is not satisfiable, then every finite state sequence is a bad prefix of the formula $\Diamond\psi$, but this tells us nothing about that finite state sequence itself.

The idea behind the construction is very simple. One creates the on-the-fly tableau automaton of the formula φ, but interprets it as an automaton on finite words. The original acceptance conditions can be forgotten, since they refer to infinite state sequences. If a finite state sequence $\overline{\xi}$ is an informative bad prefix, then there is no finite run on the automaton that matches it. If it is an informative good prefix, then

[5] We rephrase the definition of [123] in terms of our notions of consistency.

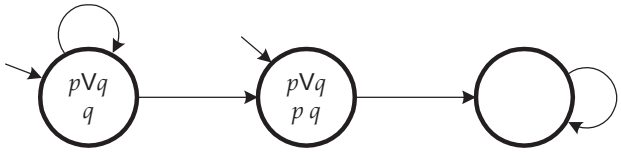


Figure 7.11: Automaton for prefixes of the formula $p\mathsf{V}q$

there is a run to the location $\varnothing$. To be precise, for any extension of the prefix, longer than the prefix itself, there is a matching run, the last location of which is $\varnothing$. Thus, if an automaton does not have a location $\varnothing$ then the formula does not have any informative good prefixes.

7.4.3 DEFINITION *Let $A = (L, \Sigma, L_0, Q, E, F)$ be an ω-automaton, then $[A_\varphi]$ is the automaton $(L, \Sigma, L_0, Q, E, L)$ on finite words, i.e. the same automaton interpreted as a safety automaton (all locations are final) on finite words.*

Example

Figure 7.11 shows the automaton on prefixes of the formula $p\mathsf{V}q$. $\{q\}\{q\}\{p,q\}$ is an (informative) good prefix of $p\mathsf{V}q$. The corresponding run to the location $\varnothing$ (the right location) is $\{p\mathsf{V}q, q\}\{p\mathsf{V}q, q\}\{p\mathsf{V}q, p, q\}\varnothing$. The run itself forms the informative sequence that establishes this. An informative bad prefix is $\{q\}\{p\}$. It can be verified that this sequence has no matching finite run on the automaton. A corresponding informative sequence demonstrating that the prefix is informative for $\neg(p\mathsf{V}q)$ is $\{\neg(p\mathsf{V}q), \neg p\}\{\neg(p\mathsf{V}q), \neg q\}\varnothing$. The informative sequence can be interpreted as follows. It claims $(\neg(p\mathsf{V}q) \in \overline{IS}(0))$ that there is no matching run starting from any location containing the formula $p\mathsf{V}q$ (and all initial locations of the automaton contain it). The reason for this is that the first state of the prefix does not satisfy p $(\neg p \in \overline{IS}(0))$ and the remainder does not satisfy $p\mathsf{V}q$ $(\neg(p\mathsf{V}q) \in \overline{IS}(1))$. There is no matching run starting from the middle location, since it contains p. Any successor location of the left location contains $p\mathsf{V}q$ again. According to the informative sequence, a run from such a successor location (left and middle) for the remainder $\{p\}$ does not exist since the second state of the prefix does not satisfy q $(\neg q \in \overline{IS}(1)$. This immediately rules out both locations as possible locations for a matching run and thus a matching run does not exist.

Automata for good and bad prefixes

Similarly to the automaton rejecting bad prefixes, we can also define an automaton on finite words accepting the good prefixes of φ. This is achieved by making only the location $\varnothing$ (if it exists) accepting. An automaton accepting bad prefixes is obtained by constructing such an automaton for $\neg\varphi$. Thus we can use the same automaton (the same locations and edges, but different acceptance conditions) to detect both the good and the bad prefixes of a formula φ.

Correctness

The above example illustrated that for an informative bad prefix, there is no matching run on the tableau automaton. Vice versa, if there is no matching run for a prefix on an automaton $[A_\varphi]$, then the prefix is informative for $\neg\varphi$. This relationship between a finite state sequence being an informative bad prefix and the existence of a matching run is formalised in this section, in theorem 7.4.7.
The example also showed the relationship between good prefixes and finite runs on the tableau automaton ending in the location $\varnothing$. Every finite run on $[A_\varphi]$ ending in $\varnothing$ constitutes an informative sequence matching informative good prefixes. Conversely, for any informative good prefix such a run can be found, this is demonstrated with theorem 7.4.9.
Remember that the formula associated with a set of formulas is the conjunction of the formulas in the set and the implicit constraints ($\bigcirc$ formulas) on the remainder of a state sequence. The normal form procedure preserves informative bad prefixes. If a prefix is informatively bad for a normal form of some formula, then it is also informatively bad for the formula itself.

7.4.4 LEMMA *If $\overline{\xi}$ is an informative bad prefix for $\mathrm{NF}(\Phi)$, then $\overline{\xi}$ is an informative bad prefix for Φ.*

For the proof, see appendix B.1.3. Next follows the main lemma showing that prefixes for which there is no matching run on the tableau automaton starting from some location Φ, are informatively bad for the formula corresponding to the location Φ.

7.4.5 LEMMA *Let A_φ be a tableau automaton, let Φ be a location of A_φ and let $\overline{\xi}$ be a prefix of a state sequence for which there is no (accepting) run on A_φ starting from Φ. Then $\overline{\xi}$ is an informative bad prefix for $\Phi \wedge \bigcirc\mathrm{Next}(\Phi)$.*

PROOF By induction on the length of the prefix $\overline{\xi}$.

- If $|\overline{\xi}| = 1$ then there is some $\psi \in \Phi$, either an atomic proposition or the negation of an atomic proposition, such that $\overline{\xi}(0) \not\models \psi$ and thus $\{\neg\psi, \neg\Phi, \neg(\Phi \wedge \bigcirc\mathrm{Next}(\Phi))\}$ is an informative sequence for $\neg(\Phi \wedge \bigcirc\mathrm{Next}(\Phi))$ on $\overline{\xi}$.
- If $|\overline{\xi}| > 1$ then either
  - the first symbol does not match, which is similar to the first case, or
  - there is no successor location for which there is a run. By induction we have that $\overline{\xi}^1$ is an informative bad prefix for every successor location Φ_i, and thus for $\bigvee \mathrm{NF}(\mathrm{Next}(\Phi))$, and by lemma 7.4.4 it is an informative bad prefix for $\mathrm{Next}(\Phi)$. From this it follows that $\overline{\xi}$ is an informative bad prefix for $\Phi \wedge \bigcirc\mathrm{Next}(\Phi)$. □

The following lemma is the main ingredient to show the converse, i.e. that informative bad prefixes have no matching run on the tableau automaton.

7.4.6 LEMMA *Let $\psi \in \Phi$ and let $\overline{IS}$ be an informative sequence demonstrating $\neg\psi$ for $\overline{\xi}$. Then there is no run for $\overline{\xi}$ on $[A_\varphi]$ starting from Φ.*

This lemma is proved by induction on the length of $\overline{\xi}$ and the structure of ψ. The proof is given in appendix B.1.4.

Now we can show that our tableau automata accept all finite sequences except the ones that are informative for $\neg\phi$ (Kupferman and Vardi show a similar result for alternating automata in [123]).

7.4.7 THEOREM *Let φ be an* LTL *formula and let A_φ be a tableau automaton for φ. Then $[A_\varphi]$ accepts finite state sequence $\overline{\xi}$ iff $\overline{\xi}$ is not an informative bad prefix of φ.*

PROOF ($\Rightarrow$) Assume towards a contradiction that $\overline{\xi}$ is an informative bad prefix for φ. Any initial run starts from a location Φ such that $\varphi \in \Phi$. But by lemma 7.4.6 such a run cannot exist.
($\Leftarrow$) Again by contradiction. Assume that $\overline{\xi}$ is not accepted by $[A_\varphi]$. Then by lemma 7.4.5, for every $\Phi \in \mathrm{NF}(\{\varphi\})$ (the initial locations of the automaton), $\overline{\xi}$ is an informative bad prefix for $\Phi \wedge \bigcirc \bigwedge \mathrm{Next}(\Phi)$. Thus by lemma 7.4.4, $\overline{\xi}$ is an informative bad prefix for φ. □

Next, we show that informative *good* prefixes can be recognised by the existence of a run to the location $\varnothing$.

7.4.8 LEMMA *Let Φ be a set of formulas and let $\overline{IS}$ be an informative sequence with $\Phi \subseteq \overline{IS}(0)$. Then there is some $\Phi' \in \mathrm{NF}(\Phi)$ such that $\Phi' \subseteq \overline{IS}(0)$ and $\mathrm{Next}(\Phi') \subseteq \overline{IS}(1)$.*

The proof is given in appendix B.1.5. As a consequence, a finite state sequence is an informative good prefix iff there is a matching run leading to the location $\varnothing$.

7.4.9 THEOREM *Let A_φ be the on-the-fly tableau automaton of the formula φ. A finite state sequence $\overline{\xi}$ is an informative good prefix of φ iff there is a finite initial run on $[A_\varphi]$ matching $\overline{\xi}$ ending in the location $\varnothing$.*

PROOF ($\Rightarrow$) Let $\overline{IS}$ be an informative sequence with $\varphi \in \overline{IS}(0)$. By lemma 7.4.8, there is some $\Phi \in \mathrm{NF}(\varphi)$ such that $\Phi \subseteq \overline{IS}(0)$ and $\mathrm{Next}(\Phi) \subseteq \overline{IS}(1)$. Repeating the argument, we can show that there is a run $\overline{\Phi}$ such that $\overline{\Phi}(k) \subseteq \overline{IS}(k)$ for all $0 \leq k \leq |\overline{IS}|$. Thus $\overline{\Phi}(|\overline{IS}|) = \varnothing$.
($\Leftarrow$) Let $\overline{\Phi}$ be such a run. Then $\overline{\Phi}$ itself is an informative sequence for φ. □

7.5 Related Work

Tableau algorithms

Tableau algorithms are essential parts of automata-based temporal-logic verification programs (see chapter 6). They translate temporal-logic properties into the realm of automata and allow the use of procedures on automata for model-checking. As automated verification techniques are limited by the size of the state space of the automata they have to deal with, there is a strong pressure to find tableau algorithms that yield smaller automata.
The connection between LTL formulas and Büchi automata was first established by Wolper, Vardi and Sistla [184]. It was done by constructing two automata, one to check state-to-state consistency (safety) and one to check that all eventualities (liveness requirements) are satisfied. The tableau automaton was then created by taking the product of these two automata. The size of tableau automata is in worst case

exponential in the length of the formula and their construction always realised automata of that size. The more efficient 'on-the-fly' constructions were introduced in [89]. Now many formulas in practice could be translated to moderately sized automata and this has made LTL model checking practical. Further optimisations have been made by others, [60, 166, 73]. Other work on tableau constructions for LTL includes, [67] with an improved proof of correctness of the algorithm in [89], [118] with a tableau for a logic that comprises both future-time and past-time temporal logic operators and [112] describing an efficient satisfiability checker.
Although tableau algorithms are usually presented with a proof of their correctness, the use of optimisations and efficient implementation techniques make that their correct operation in practice is sometimes doubtful. This is observed for instance in [168], which describes a testing methodology for tableau implementations and its application.

Fragments of linear temporal logic

Tableau constructions and model-checking algorithms are complex and computationally intensive. Sometimes things can be simplified for particular subclasses of problems. Liveness properties and fairness constraints, for instance, require the use of acceptance sets of the underlying automata. If properties are restricted to safety properties (and fairness constraints are not applied) then the model-checking problem can be simplified to a reachability analysis [123]. In such a reachability analysis, one can reason about finite executions of a system rather than infinite ones. We have seen in chapter 6 that this is essential to using the technique in the context of simulations.
In [3, 2] a fragment of a modal μ-calculus-like timed temporal logic is used, that is similar to the deterministic fragment presented in this chapter. It allows for the use of a (timed) automaton construction and reachability analysis for model-checking of this logic.
In [133], Maidl defines a 'deterministic' fragment of ACTL, and shows that this fragment captures exactly those properties expressible in ACTL that are also expressible in LTL. This is achieved by requiring all choices to be resolved locally by making them deterministic. It has for instance the following restriction on the disjunction: only disjunctive formulas of the form $(\pi \wedge \psi_1) \vee (\neg\pi \wedge \psi_2)$ are allowed. It is also observed that an automaton can be constructed that recognises the negation of such a property, the size of which is linear in the size of the formula. This observation corresponds to our dual (quadratic) fragment.
Another similar fragment of a temporal logic is the flat logic flatCTL(*) discussed in [58, 59]. It is called flat, since the use of the Until operator is restricted to have on its left side only propositional formulas. The equivalence relations induced by such logics are investigated for the use of partition-refinement methods. In comparison with our fragment, it can be noted that only the Until operator is restricted, not the disjunction. Also the negation is used normally, giving a dual of the Until operator that is also restricted to propositional formulas on the left side, in contrast with our fragment, where the Release is restricted on the right side. It is observed that the distinguishing power of flatCTL* differs from that of flatCTL, this in contrast with other fragments for which there was no such difference. It would be interesting to see what would happen in the case of a fragment like in this chapter or in case

of Maidl's fragment, since the deterministic choice leaves no room to exploit path quantification.

7.6 Conclusions and Future Work

In this chapter an overview was given of the tableau method for linear temporal logic that comprises both the complete and on-the-fly approaches and a description of the algorithms for producing tableau automata was given. We have also discussed two particular fragments of LTL that lead to particularly efficient tableaux. One fragment allowing the construction of deterministic tableaux and one leading to tableaux that are at most quadratic in size compared to the size of the formula instead of exponential. Moreover, these fragments were introduced since similar fragments will turn out to be useful for the effective analysis of timed temporal logic in the next chapter. It has also been shown how automata for informative prefixes can be constructed. These can be used to simplify the model checking of particular safety formulas [123] and to apply model-checking techniques in the context of simulations as described in chapter 6 and [82]. It has been described how in practice informative good and bad prefixes can be detected using a single automaton.
For simplicity, we have described our methods for LTL, even without the $\bigcirc$ operator. It is possible and useful to include other operators as well. Past-time operators for instance, reasoning backwards in time instead of forward, would be valuable for the specification of temporal properties. They can be included as it is done in [163] or [118].
In our automata for prefixes, the acceptance sets of the original tableau automata were discarded, as they were formally only meaningful for infinite computations. In practice however, it might be possible to extract valuable information from it. It could be interesting to observe the difference between a run that does not pass any accepting locations and one that does pass accepting locations every now and then.
Our notion of informative good and bad prefixes is closely related to the notions of local and temporal consistency that define the tableau automata. It would be interesting to see how this relates to different kinds of optimisations proposed by various authors, [89, 60, 166, 73]. In a sense, the informative sequences are proofs that the state sequence does or does not satisfy the property. The steps of these proofs are given by the local and temporal consistency rules that guide the on-the-fly construction. It may be so that more sophisticated rules for constructing the on-the-fly tableau give rise to informative proofs that use more sophisticated proof rules.

Chapter 8

Tableaux for a Real-Time Temporal Logic

In the previous chapter, tableau constructions for linear temporal logic have been described. Formulas in LTL allow the specification of particular orders of events. In the analysis of real-time systems, we are often interested in quantitative timing properties. Extensions of LTL exist that aim to express such properties. Instead of state sequences of the untimed systems we will now look at timed state sequences that include timing information.
In LTL we can express for instance that every a event is at some point followed by a b event, but we cannot express constraints on the amount of time between these events. The timed extensions of LTL aim to provide such expressive power.
We proceed along the same lines as the previous chapter, first introducing the complete tableaux and then on-the-fly tableaux, and see how we have to extend the method to deal with qualitative temporal properties. To keep matters simple we will initially only consider a particular type of timed state sequences. We only allow one type of intervals, namely those that are left-closed and right-open, i.e. that are of the form $[a, b)$ for $b > a$. Tableaux for this type of timed state sequences are introduced in section 8.1. Subsequently, section 8.2 introduces tableau constructions for arbitrary interval types. With some extra bookkeeping, the same principles can be applied.

8.1 Restricted Real-Time Tableaux

We used Büchi automata for tableaux of LTL properties. In order to capture the quantitative timing properties, we employ timed automata (see section 2.6.2) for our timed tableaux. If we want to define a tableau construction for real-time temporal logic, we make use of timers to remember the distance in time to important events in the past. This information by itself however, may not be enough to determine the resulting constraints on the remaining timed state sequence. For instance, suppose we are interested in the formula $\Diamond_{\leq 5} p$ and we want to verify that the formula holds for every moment in some timed state sequence $\overline{\rho}$. If we know that at moment t, the

state changed from $\{p\}$ (p is true) to $\varnothing$ (p is false), then this information is insufficient to determine constraints on the future of the state sequence. For let $\overline{\rho}_1 = (\overline{\sigma}, \overline{I}_1)$ and $\overline{\rho}_2 = (\overline{\sigma}, \overline{I}_2)$ be timed state sequences, partly depicted below

$$\begin{array}{lccc} \overline{\sigma}: & \{p\} & \varnothing & \{p\} \\ \overline{I}_1: & \ldots t) & [t, t+5] & (t+5, \ldots \\ \overline{I}_2: & \ldots t] & (t, t+5] & (t+5, \ldots \end{array}$$

then it is clear that in $\overline{\rho}_1$, $\Diamond_{\leq 5} p$ does not hold at time t, whereas it does hold at t in $\overline{\rho}_2$[1].

In order to successfully construct a tableau automaton for MITL formulas, one needs to memorise both the moment of a transition (using a timer) and the type of the transition (from a right-open to a left-closed interval or vice versa).

In this section we start with defining our real-time temporal logic, MITL$_{\leq}$, and a tableau construction for MITL$_{\leq}$, interpreted over a restricted class of timed state sequences, namely those whose intervals are always left-closed and right-open. Then the type of transition is known a priori and we can concentrate on the use of timers to introduce the quantitative timing constraints. In section 8.2 we discuss the general case, building upon the restricted version and adding the required bookkeeping to keep track of the types of transitions.

8.1.1 Preliminaries

If we use interval sequences in this section, we refer to interval sequences as defined in section 2.1.5, but only those that are composed of intervals that are all left-closed and right-open. Also, the timed automata in this section have only runs that use this kind of interval sequences and thus only accept timed state sequences with this kind of intervals.

Moreover, we will use a restricted version of MITL ([5], see section 2.5.3). In the construct $\varphi_1 \mathsf{U}_I \varphi_2$, the interval I is restricted to be of the form $[0, d]$. The formula is denoted as $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$. It is not that hard to extend the results to include also intervals such as $[0, \infty)$, $[0, d)$ or $(0, d)$. The use of arbitrary (rational) lower bounds to the interval however, makes the construction too complex for practical use (EXPSPACE as opposed to PSPACE).

8.1.2 Real-Time Temporal Logic

In this section MITL$_{\leq}$ is defined as a restricted version of Metric Interval Temporal Logic MITL [5] (see section 2.5.3). The syntax of MITL$_{\leq}$ is defined by the following grammar ($d \in \mathbb{N}$):

$$\varphi ::= \mathsf{true} \mid p \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathsf{U}_{\leq d} \varphi_2.$$

The semantics of MITL$_{\leq}$ remains unchanged from the semantics of MITL as defined in section 2.5.3 (using $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ as shorthand notation for $\varphi_1 \mathsf{U}_{[0,d]} \varphi_2$). Notice that

[1] This is true even for the MITL formula $\Diamond_{<5} p$, a case that, we suspect, is not dealt with correctly in [5, 9]. Although they observe that requiring p to hold at some time where the associated clock is smaller than 5 is too strong, their solution, changing this to the clock being at most 5, is still too strong. (See also section 8.2.2.)

$\text{MITL}_{\leq}$ cannot discriminate between equivalent timed state sequences: if $\overline{\rho}_1 \equiv \overline{\rho}_2$ then $\overline{\rho}_1 \models \varphi$ iff $\overline{\rho}_2 \models \varphi$ for any $\text{MITL}_{\leq}$ formula φ.

8.1.1 DEFINITION *(After [5]) Let $\varphi \in MITL_{\leq}$. An interval sequence $\overline{I}$ is called φ-fine for timed state sequence $\overline{\rho}$ if for all $\psi \in sub(\varphi)$, all $k \geq 0$ and all $t_1, t_2 \in \overline{I}(k)$, $\overline{\rho}^{t_1} \models \psi$ iff $\overline{\rho}^{t_2} \models \psi$. A timed state sequence $\overline{\rho} = (\overline{\sigma}, \overline{I})$ is called φ-fine if $\overline{I}$ is φ-fine for $\overline{\rho}$.* □

Note that due to the choice of $[0, d]$ as the interval of the Until formulas, φ-fine interval sequences also consist of left-closed and right-open intervals. Such a φ-fine interval sequence exists for any φ and $\overline{\rho}$.

8.1.2 LEMMA *[83, 5] Let φ be an $MITL_{\leq}$ formula and let $\overline{\rho}$ be a timed state sequence. Then there exists a φ-fine interval sequence for $\overline{\rho}$.*

PROOF One can show by induction on the structure of φ that this is the case. The only non-trivial case is the formula $\varphi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2$. Let $\overline{I}_1$ and $\overline{I}_2$ be φ_1 respectively φ_2-fine interval sequences for $\overline{\rho}$. Then we can construct a φ-fine interval sequence $\overline{I}$ by letting our new interval sequence have a transition to a new interval at every point in time where $\overline{I}_1$ has a transition or $\overline{I}_2$ has a transition, and at all points at distance d in time before such transitions (if larger than 0). Then $\overline{I}$ is a φ-fine interval sequence for $\overline{\rho}$. □

For our tableaux we need a suitable discretisation of the time domain. In the untimed case, the state sequences were inherently discrete. For timed state sequences, the φ-fine interval sequences provide the necessary discretisation.
The *language* $\mathcal{L}(\varphi)$ associated with the $\text{MITL}_{\leq}$ formula φ (w.r.t. a set *Prop* of propositions) is the set of all timed state sequences over *Prop* that satisfy φ,

$$\mathcal{L}(\varphi) = \{\overline{\rho} \mid \overline{\rho} \models \varphi\}.$$

8.1.3 Tableaux for Real-time Temporal Logic

In this section we define the construction of a timed automaton that accepts precisely all the models that satisfy an $\text{MITL}_{\leq}$ formula φ. We first describe the ideas behind the construction and later give a precise definition.

Intuition behind the construction

The real-time tableau automaton is very similar to the untimed tableaux for LTL formulas. The added quantitative timing constraints are dealt with by timers of the timed automaton. Consistency constraints are applied the same way as in the untimed case. Temporal consistency is modified to include validity of the timer manipulations.
Figure 8.1 shows an example of a tableau automaton for the formula $\Box_{\leq 5} p$. For the time of stay in the initial location, the formula $\Box_{\leq 5} p$ must hold and so must p. After leaving the initial location, a timer y is set to 5 and until the time has decreased to 0, p should remain satisfied. After that, a transition to the upper right location is possible. At this location, p need no longer hold and the run continues to the lower

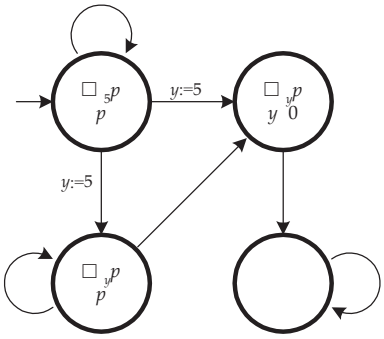


Figure 8.1: Example tableau of the formula $\Box_{\leq 5} p$

right location, where it can remain forever. Note that the transition from the upper left to the upper right location labelled with the timer setting $y := 5$ leading to a location where y is required to be at most 0 is redundant[2], but it is produced by the on-the-fly algorithm to be presented later.
The general strategy for checking that timed state sequences satisfy an Until formula or its negation when they match a run that passes through a location labelled accordingly, is done in two parts. First, we look at checking the positive Until formulas and later we look at the structure for checking negations of Until formulas.
The strategy for checking the formula $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2$ is depicted in figure 8.2. As in the previous chapter, the large circles labelled with formulas denote *classes* of locations, namely those that are labelled with at least those formulas. In the figure, only the edges of the locations labelled with $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ are fully The edges Notice that the transitions are almost identical to those of the untimed version of figure 7.2. There is an extra transition from the non-trivial $\neg(\varphi_1 \mathsf{U}_{\leq d} \varphi_2)$ sets to the non-trivial $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ sets. Whereas this situation was impossible in the untimed case, it may occur in the timed case if a moment when φ_2 holds comes within the reach of d units of time. For instance, if p holds for all $t \geq 2$ and does not hold for any $t < 2$, then for $t < 1$, the formula $\mathrm{true}\mathsf{U}_{\leq 1} p$ (non-trivially) does not hold and for all $1 \leq t < 2$ it (non-trivially) does hold.

- If a location is labelled with $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ and with φ_2 at the same time, it is trivial for $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$. The Until formula is immediately satisfied and no other constraints are required (no timers need to be applied yet).

- If a location is labelled with $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$, but not with φ_2, then the Until formula is not satisfied immediately and using temporal consistency constraints as in the untimed case, one can verify that at some later time φ_2 will occur and φ_1 remains true in the meantime. But besides that, we now have to verify that this φ_2 occurs within d time units from the moment we first encountered such a 'non-trivial' Until location. This is accomplished by setting a corresponding timer, say x, to the value d at the moment the non-trivial location is entered and requiring that the value of this timer remains larger than zero as long

[2]Although it would not be redundant if, for instance, the timed automaton was interpreted on the discrete time domain consisting of the time points $\{0, 5, 10, 15, \ldots\}$.

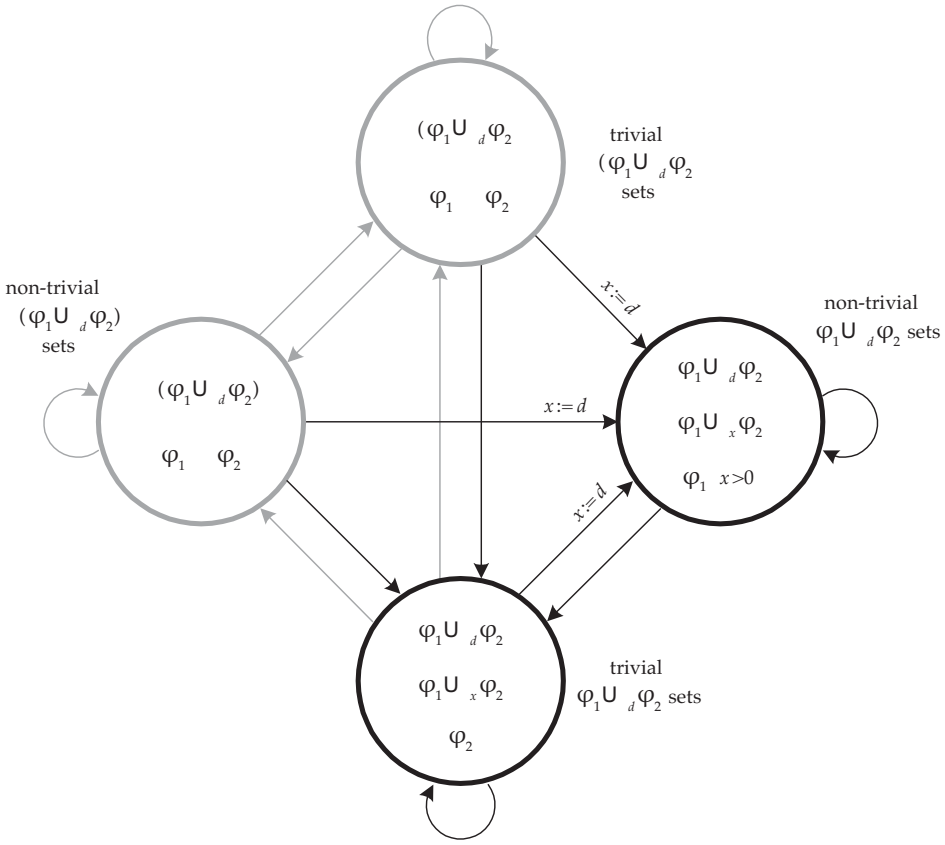


Figure 8.2: Pattern for checking the formula $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$

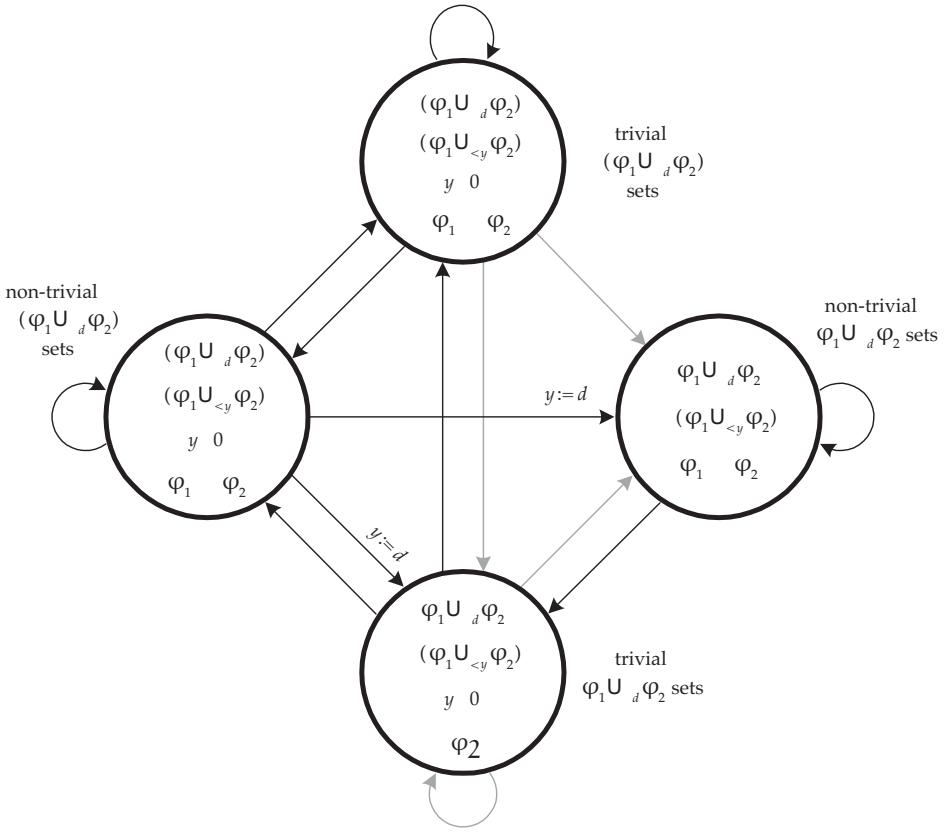


Figure 8.3: Pattern for checking the formula $\neg\ (\varphi_1 \mathsf{U}_{\leq d} \varphi_2)$

as φ_2 is not yet encountered (remember that we let timers decrease as time advances). As soon as φ_2 is encountered, the timer is 'deactivated', i.e. the constraint $x > 0$ disappears. In order to signify that the timer x is active in a location and 'guards' the formula $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$, we label the location with $\varphi_1 \mathsf{U}_{\leq x} \varphi_2$. The intuitive meaning of the label is that whenever the automaton is in an extended location (ℓ, ν) (the automaton resides in location ℓ and the timer valuation equals ν) then in order to accept the timed state sequence, the 'formula' $\varphi_1 \mathsf{U}_{\leq x} \varphi_2$ must hold, where $\nu(x)$ is substituted for x. Note that we can use a single timer to verify the formula for all instants the automaton resides in the non-trivial ψ sets, because if it holds for the first instant, then it must also hold for later instants.

The second part of the strategy deals with the verification of a Release (negated Until) formula $\psi = \neg\,(\varphi_1 \mathsf{U}_{\leq d} \varphi_2)$ $(= (\neg\varphi_1) \mathsf{V}_{\leq d} (\neg\varphi_2))$. We apply the following rules (illustrated in figure 8.3).

- If a location is labelled with ψ and with both $\neg\varphi_1$ and $\neg\varphi_2$, then it is trivial for ψ and the formula follows immediately (the locations at the top of figure 8.3).

- If it is not labelled with $\neg\varphi_1$, then temporal consistency constraints are necessary to constrain the remainder of the sequence. Now there are two possible ways in which ψ can be true. Either because φ_2 remains false and φ_1 remains true until some point is reached where both φ_1 and φ_2 are false, or φ_2 remains false and φ_1 remains true until some point in time is reached, *more than d units of time away*. Only the latter case requires the use of timers to check the quantitative constraint 'further than d time units away'. This is done in the following way. We need to check the constraint for all instants we reside in the non-trivial $\neg(\varphi_1 \mathsf{U}_{\leq d} \varphi_2)$ locations. It suffices in this case, to verify this for the last instant (compared to the first instant for the non-trivial $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ sets), since this immediately implies the satisfaction of the constraint for all previous instants. However, because of the left-open and right-closed shape of the interval there is no last instant. Hence we need to check a slightly modified constraint represented by the label $\neg(\varphi_1 \mathsf{U}_{<y} \varphi_2)$ (notice the constraint $< y$ rather than $\leq y$). Any location containing this label shall not contain φ_2 or contains the constraint $y \leq 0$, signifying that enough time has passed and the constraints have been met. Then we have verified that the φ_2 location did not occur at a distance smaller than or equal to d from any point in time that the automaton was in a location labelled ψ. Note that even a transition exists from the non-trivial $\neg(\varphi_1 \mathsf{U}_{\leq d} \varphi_2)$ locations to the trivial $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ locations (containing φ_2), but taking this transition is only possible if d equals 0.

In order to conclude this introductory description of a tableau for $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$, we put figures 8.2 and 8.3 together, resulting in figure 8.4. One can see that both timers are either not active at the same time, or have identical values (x is active in the non-trivial $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ locations and y is active after a transition from a non-trivial $\neg(\varphi_1 \mathsf{U}_{\leq d} \varphi_2)$ location to a $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ location and remains active until a location containing $y \leq 0$ is reached). Indeed in general, it is possible to use only one timer per Until formula. In order to keep things simple in the discussion of the tableau however, we shall use two separate timers per Until formula. Also in the on-the-fly version of the tableaux to be discussed later (in section 8.1.6), we need only one timer per Until formula.

Definition of the Tableau Automaton

The definition of tableau automata for real-time temporal logic follows the same structure as the untimed tableaux in the previous chapter. Adapted versions of local and temporal consistency constraints are defined, now including constraints on timer manipulations and timer values as well.
The set $sub(\varphi)$ of *subformulas* of an $\text{MITL}_{\leq}$ formula φ is again the set of all syntactic subformulas of φ (in positive form). Next, the closure of an $\text{MITL}_{\leq}$ formula is defined. It contains its subformulas, their negations, but also for all Until and Release formulas, the corresponding labels.

8.1.3 DEFINITION *The* closure $cl(\varphi)$ *of an* $\text{MITL}_{\leq}$ *formula* φ *is the smallest set* Φ*, such that*

- $sub(\varphi) \subseteq \Phi$*;*
- *if* $\psi \in sub(\varphi)$*, then* $\neg\psi \in \Phi$*;*

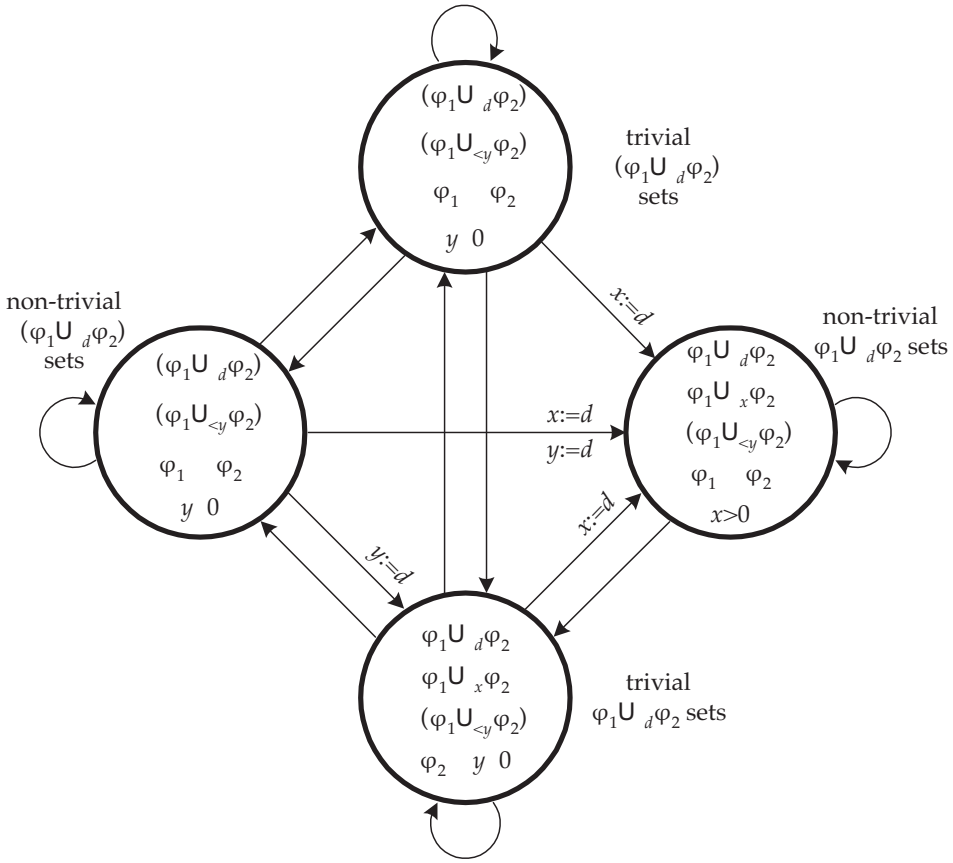


Figure 8.4: Pattern for checking the formulas $\neg\ (\varphi_1 \mathsf{U}_{\leq d} \varphi_2)$ and $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ at the same time

- *for every Until formula* $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in \Phi$*, the* labels
$$\{\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2, x_\psi > 0, \varphi_1 \mathsf{V}_{< y_\psi} \varphi_2, y_\psi \leq 0\} \subseteq \Phi. \qquad \square$$

Note that (for now) the labels are not considered to be formulas; they just help to define the locations of the automata. In section 8.1.6 we do interpret them as formulas to define a normal form for the on-the-fly construction. Similar to the negated formulas and formulas in positive form we use negated labels as well as labels 'in positive form'. Whenever convenient, we use $\neg \left(\varphi_1 \mathsf{U}_{< y_\psi} \varphi_2\right)$ to denote $(\neg\varphi_1)\mathsf{V}_{< y_\psi}(\neg\varphi_2)$. The set $\mathit{Timers}(\varphi)$ of timers for an MITL$_{\leq}$ formula φ is the set containing two timers for every Until formula $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2$ in $cl(\varphi)$ (even if there are multiple occurrences of the Until formula in φ). These timers are called x_ψ and y_ψ:

$$\mathit{Timers}(\varphi) = \{x_\psi, y_\psi \mid \psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in cl(\varphi)\}.$$

We need to extend the notions of local and temporal consistency of LTL to include the new timing related labels.

8.1.4 DEFINITION *A set* Φ *of MITL*$_{\leq}$ *formulas and labels is called* locally consistent *if*

1. false $\notin \Phi$*;*
2. *if* $\psi_1 \vee \psi_2 \in \Phi$ *then* $\psi_1 \in \Phi$ *or* $\psi_2 \in \Phi$*;*
3. *if* $\psi_1 \wedge \psi_2 \in \Phi$ *then* $\psi_1 \in \Phi$ *and* $\psi_2 \in \Phi$*;*
4. *if* $\varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in \Phi$ *then* $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \in \Phi$*;*
5. *if* $\varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2 \in \Phi$ *then* $\varphi_2 \in \Phi$ *or both* $x_{\psi'} > 0 \in \Phi$ *and* $\varphi_1 \in \Phi$*;*
6. *if* $\varphi_1 \mathsf{V}_{\leq d} \varphi_2 \in \Phi$ *then* $\varphi_2 \in \Phi$*;*
7. *if* $\varphi_1 \mathsf{V}_{< y_{\psi'}} \varphi_2 \in \Phi$ *then* $y_{\psi'} \leq 0 \in \Phi$ *or* $\varphi_2 \in \Phi$*.* $\square$

Note that requirements 1, 2, 3 are identical to the requirements in the untimed case and the others are adapted to the bounded Until formula and its labels. Non-trivial sets are defined for the new labels as well (compare definition 7.1.4).

8.1.5 DEFINITION *If* ψ *is a formula or label of one of the following forms:* $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$*,* $\varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$*,* $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$ *or* $\varphi_1 \mathsf{V}_{< y_{\psi'}} \varphi_2$*, then a* ψ*-set* Φ *is called non-trivial for* ψ

- *if* $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2$ *and* $\varphi_2 \notin \Phi$*;*
- *if* $\psi = \varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$ *and* $\varphi_2 \notin \Phi$*;*
- *if* $\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2$ *and* $\varphi_1 \notin \Phi$*;*
- *if* $\psi = \varphi_1 \mathsf{V}_{< y_{\psi'}} \varphi_2$*,* $\varphi_1 \notin \Phi$ *and* $y_{\psi'} \leq 0 \notin \Phi$*.* $\square$

Again, for such a formula or label ψ, a ψ-set is called trivial for ψ if it is not non-trivial for ψ. Completeness of a set now requires that it is fully decorated with the appropriate labels (corresponding to figure 8.4).

8.1.6 DEFINITION *A set Φ is called* complete *w.r.t. the set L of formulas and labels if*

1. *for every* formula $\varphi \in L$, *either* $\varphi \in \Phi$ *or* $\neg\varphi \in \Phi$ *(but not both).*
2. *for every* $\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2 \in L$, $\varphi_1 \mathsf{V}_{< y_{\neg\psi}} \varphi_2 \in \Phi$;
3. *for every* $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in L$, $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \in \Phi$ *iff* $\psi \in \Phi$;
4. *for every* $\psi = \varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2 \in L$, $x_{\psi'} > 0 \in \Phi$ *iff* Φ *is non-trivial for* ψ;
5. *for every* $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in L$, $y_\psi \leq 0 \in \Phi$ *iff* Φ *is not non-trivial for* $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$. □

A set Φ is called *complete* w.r.t. the formula φ if Φ is complete w.r.t. $cl(\varphi)$. Temporal consistency is also defined and includes constraints on the added labels as well as on the timer settings used on edges.

8.1.7 DEFINITION *A set Φ' of formulas is a* temporally consistent successor *of the set Φ of formulas under the timer setting TS if for every bounded Until formula $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2$,*

1. *if Φ is non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$, then $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \in \Phi'$ and x_ψ is undefined in TS;*
2. *if Φ is not non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ and Φ' is, then $TS(x_\psi) = d$;*

and if for every bounded Release formula $\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2$,

3. *if Φ is non-trivial for ψ, then $\psi \in \Phi'$ or both $TS(y_\psi) = d$ and $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2 \in \Phi'$;*
4. *if Φ is non-trivial for $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2$ then $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2 \in \Phi'$.*

Temporal consistency is denoted as $\Phi \xrightarrow{TS} \Phi'$. □

To be used later for an on-the-fly construction, we describe temporal consistency in terms of temporal consistency constraints similar to the untimed case. The temporal consistency constraints of a location now determine not only formulas or labels of the next location, but also place constraints upon the update of timer values.

8.1.8 DEFINITION *Let Φ be a set of formulas. Then the pair* $\mathrm{Next}(\Phi)$ *of temporal consistency constraints, is the pair (TS, Φ') where TS and Φ' are the smallest sets such that*

- *if Φ is non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ then Φ' contains $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$;*
- *if Φ is non-trivial for $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$ then Φ' contains $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2$ and TS contains $y_\psi := d$;*
- *if Φ is non-trivial for $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2$ then Φ' contains $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2$.* □

Now the complete tableau can be defined using these consistency requirements as the following automaton.

8.1.9 DEFINITION *The (restricted) complete* tableau automaton A_φ *of MITL*$_\leq$ *formula* φ *is the timed automaton* $(L, \Sigma, \Theta, L_0, Q, TC, E)$ *where*

- $L \subseteq 2^{cl(\varphi)}$ *contains all sets that are locally consistent and complete w.r.t.* φ*;*
- $\Sigma = 2^{Prop(\varphi)}$*;*
- $\Theta = Timers(\varphi)$*;*
- L_0 *is the set of all extended locations* (Φ_0, ν_0)*, such that* $\Phi_0 \in L$*,* $\varphi \in \Phi_0$ *and* $\nu_0(x_\psi) = d$ *and* $\nu_0(y_\psi) = 0$ *for every bounded Until formula* $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ *in* $cl(\varphi)$*;*
- Q *is the mapping that assigns to the location* Φ*, the set of all states* $\sigma \in 2^{Prop}$ *such that*
  - $p \in \sigma$ *if* $p \in \Phi$*,*
  - $p \notin \sigma$ *if* $\neg p \in \Phi$*;*
- TC *is the mapping that assigns to the location* Φ*, the set of all timer conditions (see section 2.6.2)* $\chi \in \Phi$*;*
- E *is the set of all edges* $(\Phi, TS, \Phi') \in L \times TSet(\Theta) \times L$ *such that* $\Phi \xrightarrow{TS} \Phi'$ *(only considering timer settings of the form* $x_\psi := d$ *or* $y_\psi := d$ *for bounded Until formulas* $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in cl(\varphi)$*).* □

8.1.4 Example

As an example, we construct the tableau automaton of the formula $\varphi = \neg \Diamond_{\leq 5} p = \neg (\mathrm{true} \mathsf{U}_{\leq 5} p)$. The subformulas of φ, are

$$sub(\varphi) = \{\neg (\mathrm{true} \mathsf{U}_{\leq 5} p), \mathrm{true} \mathsf{U}_{\leq 5} p, p, \mathrm{true}\}.$$

The closure set is

$$cl(\varphi) = \{\neg (\mathrm{true} \mathsf{U}_{\leq 5} p), \mathrm{true} \mathsf{U}_{\leq 5} p, p, \neg p, \mathrm{true}, \neg \mathrm{true}, \neg \left(\mathrm{true} \mathsf{U}_{<y} p\right), \mathrm{true} \mathsf{U}_{\leq x} p, x > 0, y \leq 0\}.$$

Inspecting all the subsets of $cl(\varphi)$, we find only three complete locally consistent subsets:

1. $\{\mathrm{true} \mathsf{U}_{\leq 5} p, p, \mathrm{true}, \neg \left(\mathrm{true} \mathsf{U}_{<y} p\right), \mathrm{true} \mathsf{U}_{\leq x} p, y \leq 0\}$;
2. $\{\mathrm{true} \mathsf{U}_{\leq 5} p, \neg p, \mathrm{true}, \neg \left(\mathrm{true} \mathsf{U}_{<y} p\right), \mathrm{true} \mathsf{U}_{\leq x} p, x > 0\}$;
3. $\{\neg (\mathrm{true} \mathsf{U}_{\leq 5} p), \neg p, \mathrm{true}, \neg \left(\mathrm{true} \mathsf{U}_{<y} p\right), y \leq 0\}$.

We see that set number 2 is non-trivial for $\mathrm{true} \mathsf{U}_{\leq 5} p$ and set number 3 is non-trivial for $\neg (\mathrm{true} \mathsf{U}_{\leq 5} p)$. We can now determine the temporally consistent pairs of sets and appropriate timer settings. Set 3 is the (only) initial location, since it contains φ. This leads to the automaton represented in figure 8.5.

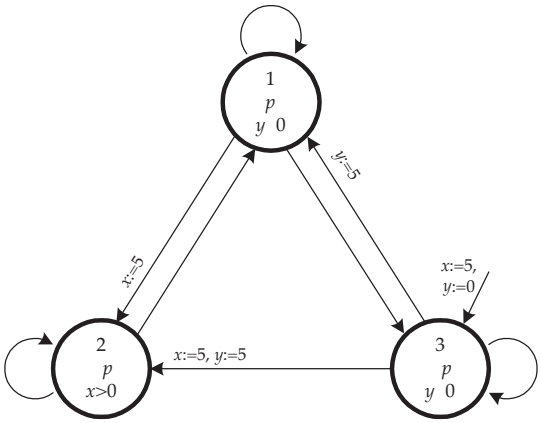


Figure 8.5: Timed tableau automaton of the formula $\neg\Diamond_{\leq 5}p$

8.1.5 Correctness

We will now prove that the tableau A_φ accepts precisely the timed state sequences that satisfy the formula φ. The proof is very similar to the corresponding proof in the untimed case, but a little more intricate.

8.1.10 THEOREM *Let φ be an MITL$_\leq$ formula and let A_φ be its tableau automaton. Then $\mathcal{L}(A_\varphi) = \mathcal{L}(\varphi)$.*

The theorem follows immediately from the lemmas 8.1.17 and 8.1.21, which prove soundness and completeness respectively.

Soundness

To show that the tableau is sound we need to demonstrate that every timed state sequence accepted by the automaton A_φ satisfies the formula φ. As in the untimed case we show that the formulas in the locations of the automaton are satisfied by any timed state sequence matching a run starting in that location. The following two lemmas deal with the Until formulas and are illustrated by figure 8.6. The first lemma says that timers and temporal consistency constraints are applied as intended to verify Until formulas.

8.1.11 LEMMA *Let $\bar{r} = (\overline{\Psi}, \overline{I}, \overline{\nu})$ be a timed run for $\overline{\rho}$ on A_φ and $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in cl(\varphi)$. For any $k \geq 0$ such that $\overline{\Psi}(k)$ is non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$, we have (i) $\varphi_1 \in \overline{\Psi}(k)$, (ii) $x_\psi > 0 \in \overline{\Psi}(k)$, (iii) $\overline{\nu}(k+1)(x_\psi) = \overline{\nu}(k)(x_\psi) - |\overline{I}(k)|$ and (iv) either $\varphi_2 \in \overline{\Psi}(k+1)$ or $\overline{\Psi}(k+1)$ is non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$.*

PROOF $\varphi_1 \in \overline{\Psi}(k)$ and $x_\psi > 0 \in \overline{\Psi}(k)$ follow from the fact that $\overline{\Psi}(k)$ is non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ and local consistency requirement 5. From temporal consistency (1) it follows that timer x_ψ is not set in the transition from $\overline{\Psi}(k)$ to $\overline{\Psi}(k+1)$ and thus $\overline{\nu}(k+1)(x_\psi) = \overline{\nu}(k)(x_\psi) - |\overline{I}(k)|$. Finally, it follows from temporal consistency (1) that either $\varphi_2 \in \overline{\Psi}(k+1)$ or $\overline{\Psi}(k+1)$ is non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$. □

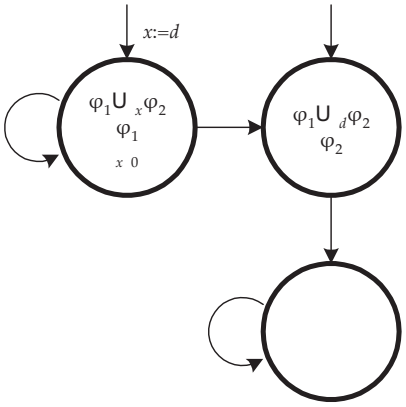


Figure 8.6: Soundness of the verification of $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$

We can now prove the lemma that tells us that Until formulas are checked correctly (notice the correspondence between the following lemma and lemma 7.1.9).

8.1.12 LEMMA *Let $\bar{r}$ be a timed run for $\overline{\rho}$ on A_φ and let $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in \bar{r}(0)$. Then there is some $t \leq d$, such that $\varphi_2 \in \bar{r}(t)$ and for all $0 \leq t' < t$, $\varphi_1 \in \bar{r}(t')$ or $\varphi_2 \in \bar{r}(t')$.*

PROOF Let $\bar{r} = (\overline{\Psi}, \overline{I}, \overline{\nu})$. If $\psi \in \overline{\Psi}(0)$ then either of the following is true:

- $\overline{\Psi}(0)$ is trivial for ψ, then $\varphi_2 \in \overline{\Psi}(0)$ by local consistency and the lemma follows trivially, taking $t = 0$.
- $\overline{\Psi}(0)$ is non-trivial for ψ (and for $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ by local consistency) then one can show using lemma 8.1.11 and knowing that $\overline{\nu}(0)(x_\psi) \leq d$, that there is some $k \geq 0$, such that $l(\overline{I}(k)) \leq d$, $\varphi_2 \in \overline{\Psi}(k)$ and for all $0 \leq m < k$, $\varphi_1 \in \overline{\Psi}(m)$. Thus taking $t = l(\overline{I}(k))$, $\varphi_2 \in \bar{r}(t)$ and for all $0 \leq t' < t$, $\varphi_1 \in \bar{r}(t')$ or $\varphi_2 \in \bar{r}(t')$, the lemma follows. □

The next two lemmas help us to show the same for Release formulas (illustrated by figure 8.7). From a location labelled with $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$ (and thus also with φ_2), a run leads to a trivial $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$ location containing φ_1 and φ_2 or to a location containing $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2$ (and thus also containing φ_2). From there, a run must proceed until a trivial $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2$ location is reached (either with $y_\psi \leq 0$ or with φ_1).

8.1.13 LEMMA *Let $\bar{r} = (\overline{\Psi}, \overline{I}, \overline{\nu})$ be a timed run for $\overline{\rho}$ on A_φ and let $\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2 \in cl(\varphi)$. For any $k \geq 0$ such that $\overline{\Psi}(k)$ is non-trivial for ψ, one of the following is true: (i) $\overline{\Psi}(k+1)$ is trivial for ψ, (ii) $\overline{\Psi}(k+1)$ is non-trivial for ψ or (iii) $\overline{\nu}(k+1)(y_\psi) = d$ and $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2 \in \overline{\Psi}(k+1)$.*

PROOF From temporal consistency (3) it follows that $\overline{\Psi}(k+1)$ is a ψ set or $\overline{\Psi}(k+1)$ contains $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2$. In the former case it can be either trivial (i) or non-trivial (ii) for ψ. In the latter case (iii), $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2 \in \overline{\Psi}(k+1)$ and $\overline{\nu}(k+1)(y_\psi) = d$. □

The three options in the previous lemma correspond to the outgoing edges of the left location at the top in figure 8.7. The following lemma deals with the middle location in the middle row of the figure.

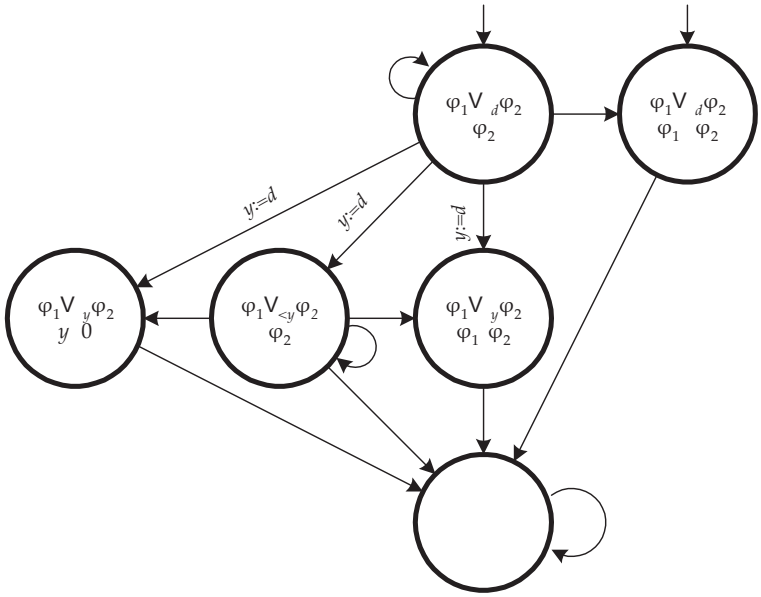


Figure 8.7: Soundness of the verification of $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$

8.1.14 LEMMA *Let $\bar{r} = (\overline{\Psi}, \overline{I}, \overline{\nu})$ be a timed run for $\overline{\rho}$ on let A_φ and $\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2 \in cl(\varphi)$. If $\overline{\Psi}(0)$ is non-trivial for $\varphi_1 \mathsf{V}_{<y_\psi} \varphi_2$ and $\overline{\nu}(k)(y_\psi) = d' \geq 0$ then for all $0 \leq t < d'$, $\varphi_2 \in \bar{r}(t)$ or there is some t', $0 \leq t' < t$ such that $\varphi_1 \in \bar{r}(t')$ and $\varphi_2 \in \bar{r}(t')$.*

PROOF If for some $i \geq 0$, $\overline{\Psi}(i)$ is non-trivial for $\varphi_1 \mathsf{V}_{<y_\psi} \varphi_2$ then from temporal consistency (4) it follows that $\varphi_1 \mathsf{V}_{<y_\psi} \varphi_2 \in \overline{\Psi}(i+1)$. Local consistency (7) says that when $\varphi_1 \mathsf{V}_{<y_\psi} \varphi_2 \in \overline{\Psi}(i+1)$, then $\varphi_2 \in \overline{\Psi}(i+1)$ or $y_\psi \leq 0 \in \overline{\Psi}(i+1)$. From this it follows that the first (if any) m such that $\varphi_2 \notin \overline{\Psi}(m)$ occurs either for $l(\overline{I}(m)) \geq d'$ and for all $0 \leq n < m$, $\varphi_2 \in \overline{\Psi}(n)$ or if there is some $0 \leq n < m$ such that $\varphi_1 \in \overline{\Psi}(n)$ and $\varphi_2 \in \overline{\Psi}(n)$. □

Using the previous two lemmas together, we can now show that the Release formula is checked correctly (corresponding to lemma 7.1.10 in the untimed case).

8.1.15 LEMMA *Let $\bar{r}$ be a timed run for $\overline{\rho}$ on A_φ, let $\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2 \in cl(\varphi)$ and let $\psi \in \bar{r}(0)$. Then for all $0 \leq t \leq d$, $\varphi_2 \in \bar{r}(t)$ or there is some $0 \leq t' < t$, $\varphi_1 \in \bar{r}(t')$ and $\varphi_2 \in \bar{r}(t')$.*

PROOF Let $\bar{r} = (\overline{\Psi}, \overline{I}, \overline{\nu})$. Either of the following is true:

- $\overline{\Psi}(0)$ is trivial for ψ, then the lemma follows trivially, since $\varphi_2 \in \bar{r}(0)$ and $\varphi_1 \in \bar{r}(0)$.
- $\overline{\Psi}(0)$ is non-trivial for ψ, then from lemma 8.1.13 it follows that the first (if any) $\overline{\Psi}(i)$ that is *not* non-trivial for ψ, is either trivial for ψ or contains $\varphi_1 \mathsf{V}_{<y_\psi} \varphi_2$ and $\overline{\nu}(i)(y_\psi) = d$,
  - if $\overline{\Psi}(i)$ is non-trivial for ψ for all $i \geq 0$ then for all $t \geq 0$, $\varphi_2 \in \bar{r}(t)$ by local consistency (6);
  - if the first is trivial for ψ then there is some $t \geq 0$ such that $\varphi_1 \in \bar{r}(t)$, $\varphi_2 \in \bar{r}(t)$ and $\varphi_2 \in \bar{r}(t')$ for all $0 \leq t' < t$, which satisfies the lemma;

– if the first, say $\overline{\Psi}(k)$, contains $\varphi_1 \mathsf{V}_{<y_\psi} \varphi_2$ and $\overline{\nu}(k)(y_\psi) = d$, then from lemma 8.1.14 it follows that for all $l(\overline{I}(k)) \leq t < l(\overline{I}(k)) + d$, $\varphi_2 \in \bar{r}(t)$ or there is some $l(\overline{I}(k)) \leq t' < t$ such that $\varphi_1, \varphi_2 \in \bar{r}(t')$ and since $l(\overline{I}(k)) > 0$, the lemma follows. □

Having the results for Until and Release formulas, we can now show that whenever a location is labelled with a formula ψ, then this formula is checked correctly (see lemma 7.1.11 in the untimed case).

8.1.16 LEMMA *Let $\bar{r}$ be a run on A_φ for the (restricted) timed state sequence $\overline{\rho}$ and let ψ be an MITL$_\leq$ formula such that $\psi \in \bar{r}(0)$. Then $\overline{\rho} \models \psi$.*

PROOF By structural induction on ψ

- if $\psi = \text{true}$, then it is trivial, since any timed state sequence satisfies true;
- $\psi = \text{false}$ cannot occur in any locally consistent set;
- if $\psi \in \mathit{Prop}(\varphi)$ and $\psi \in \bar{r}(0)$, then $\psi \in \overline{\rho}(0)$ and thus $\overline{\rho} \models \psi$;
- if $\psi = \neg p$ for some $p \in \mathit{Prop}(\varphi)$ and $\psi \in \bar{r}(0)$, then $p \notin \overline{\rho}(0)$ and thus $\overline{\rho} \models \psi$;
- if ψ is of the form $\psi_1 \vee \psi_2$ and $\psi \in \bar{r}(0)$, then (by local consistency) either ψ_1 or ψ_2 in $\bar{r}(0)$ and thus (by the induction hypothesis) $\overline{\rho} \models \psi$;
- if ψ is of the form $\psi_1 \wedge \psi_2$ and $\psi \in \bar{r}(0)$, then (by local consistency) both ψ_1 and ψ_2 are in $\bar{r}(0)$ and thus (by the induction hypothesis) $\overline{\rho} \models \psi$;
- if $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2$, then for all $t \geq 0$, $\bar{r}^t$ is a run on A_φ for $\overline{\rho}^t$ and if $\psi \in \bar{r}(0)$, then by lemma 8.1.12, there is some $0 \leq t \leq d$ such that $\varphi_2 \in \bar{r}(t)$ and $\varphi_1 \in \bar{r}(t')$ or $\varphi_2 \in \bar{r}(t')$ for all $0 \leq t' < t$. By the induction hypothesis, it follows that $\overline{\rho} \models \psi$;
- if $\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2$, then for all $t \geq 0$, $\bar{r}^t$ is a run on A_φ for $\overline{\rho}^t$ and if $\psi \in \bar{r}(0)$, then by lemma 8.1.15, for all $0 \leq t \leq d$, $\varphi_2 \in \bar{r}(t)$ or $\varphi_1 \in \bar{r}(t')$ and $\varphi_2 \in \bar{r}(t')$ for some $0 \leq t' < t$. By the induction hypothesis it follows that $\overline{\rho} \models \psi$. □

The previous result assures us that only state sequences that satisfy the formula φ are accepted by the tableau automaton.

8.1.17 LEMMA *If A_φ accepts $\overline{\rho}$ then $\overline{\rho} \models \varphi$.*

PROOF Follows from lemma 8.1.16 and the fact that there is an initial run $\bar{r}$ on A_φ for $\overline{\rho}$ and thus $\varphi \in \bar{r}(0)$. □

Completeness

We now show that any timed state sequence that satisfies the formula φ is accepted by our automaton. Using the following definitions we construct an accepting run for a timed state sequence $\overline{\rho}$.

8.1.18 DEFINITION *Let $\overline{\rho}$ be a timed state sequence, let φ be an MITL$_\leq$ formula and let $\overline{\rho}' = (\overline{\sigma}, \overline{I})$ be equivalent to $\overline{\rho}$ where $\overline{I}$ is φ-fine. Let Φ_k contain all $\psi \in cl(\varphi)$ that are true for all $t \in \overline{I}(k)$ and $\neg\psi$ for all $\psi \in cl(\varphi)$ that are false for all $t \in \overline{I}(k)$. For an Until formula $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in cl(\varphi)$,*

- *the timer x_ψ is said to be* active *during interval $\overline{I}(k)$, if Φ_k is non-trivial for ψ;*

- *the timer y_ψ is said to be* active *during interval $\overline{I}(k)$, if Φ_k contains ψ and the closest interval before $\overline{I}(k)$ that does not contain ψ was non-trivial for $\neg\psi$.* □

Notice that the moments t at which a timer is active are the same for equivalent timed state sequences and are independent of the chosen φ-fine interval sequence.

8.1.19 DEFINITION *Given a timed state sequence $\overline{\rho}$ and an MITL$_{\leq}$ formula φ, let for all $t \geq 0$ and every $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in cl(\varphi)$, μ_t, the* φ-fine timer valuation *for $\overline{\rho}$ at moment t, be defined as follows.*

- *$\mu_t(x_\psi)$ equals d minus the time, elapsed since the last time x_ψ was activated ($\mu_t(x_\psi)$ equals d at the moment of activation itself). If x_ψ has never been activated before, then $\mu_t(x_\psi)$ equals d minus the time since $t = 0$.*
- *$\mu_t(y_\psi)$ equals d minus the time, elapsed since the last time y_ψ was activated. If y_ψ has never been activated before, then $\mu_t(y_\psi)$ equals 0 minus the time since $t = 0$.* □

8.1.20 DEFINITION *Given an MITL$_{\leq}$ formula φ and a timed state sequence $\overline{\rho}$. Then a φ-fine timed run $\bar{r} = (\overline{\Psi}, \overline{I}, \overline{\nu})$ is constructed as follows: let $\overline{I}$ be a φ-fine interval sequence for $\overline{\rho}$, μ_t the φ-fine timer valuation for $\overline{\rho}$ at time t, then $\overline{\nu}(k) = \mu_{l(\overline{I}(k))}$ and $\overline{\Psi}(k)$ contains*

- *all $\psi \in cl(\varphi)$ that are true for all $t \in \overline{I}(k)$ and $\neg\psi$ for all $\psi \in cl(\varphi)$ that are false;*

and for every $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in cl(\varphi)$

- *$\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ if $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ is true during $\overline{I}(k)$;*
- *$\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2$ always;*
- *$x_\psi > 0$ if it is non-trivial for ψ;*
- *$y_\psi \leq 0$ if it is not non-trivial for ψ.* □

Now we show that the φ-fine timed run for $\overline{\rho}$ is an initial run on $A\varphi$ if $\overline{\rho} \models \varphi$.

8.1.21 LEMMA *If $\overline{\rho} \models \varphi$ then A_φ accepts $\overline{\rho}$.*

PROOF Let $\bar{r} = (\overline{\Psi}, \overline{I}, \overline{\nu})$ be a φ-fine timed run for $\overline{\rho}$. Then $\bar{r}$ is a run for $\overline{\rho}$ on A. This follows from the following facts:

- $\overline{\Psi}$ is a sequence of locally consistent and complete sets w.r.t. φ. This can easily be checked by the definitions of local consistency completeness and the definition of a φ-fine timed run.
- [Symbol match] For every $t \geq 0$, $\overline{\rho}(t) \in Q(\bar{r}(t))$. This is obvious since $\bar{r}(t)$ contains precisely the propositions that are true at moment t.
- [Consecution] Let $k \geq 0$ and let every timer x be set in TS_k if x is activated at moment $l(I_{k+1})$. Then $\overline{\Psi}(k) \xrightarrow{TS_k} \overline{\Psi}(k+1)$ and thus there is an edge $(\overline{\Psi}(k), TS_k, \overline{\Psi}(k+1)) \in E$. One can show using the definition of the φ-fine timer valuations that $\overline{\nu}(k+1) = TS_k[\overline{\nu}(k) - |\overline{I}(k)|]$.
- [Timing] For all $k \geq 0$, every $t \in \overline{I}(k)$ and every $\chi \in TC(\overline{\Psi}(k)), \overline{\nu}(k) - (t - l(\overline{I}(k))) \models \chi$. This can be verified by the semantics of the logic and the definition of a φ-fine timed run.
- [Initiality] Since $\overline{\rho} \models \varphi$, $\varphi \in \overline{\Psi}(0)$. Thus $(\overline{\Psi}(0), \overline{\nu}(0)) \in L_0$. □

8.1.6 Other Tableaux

As in the untimed case we proceed to non-complete and on-the-fly constructions to obtain smaller automata that can be used for efficient analyses.

Non-complete tableaux

As in the untimed case, we can drop the completeness condition and allow also non-complete locations without changing the set of timed state sequences accepted by the automaton.

8.1.22 DEFINITION *The* non-complete real-time tableau automaton B_φ *of MITL*$_\leq$ *formula* φ *is the timed automaton* $(L, \Sigma, \Theta, L_0, Q, TC, E)$*, where* L *consists of all (also non-complete) locally consistent sets* $\Phi \subseteq 2^{cl(\varphi)}$ *of formulas and labels and the rest is defined exactly the same as the complete tableau automaton in definition 8.1.9.* □

Using these non-complete sets one can show that lemma 8.1.16 still holds since (as in the untimed case) completeness of the sets is never used.

8.1.23 THEOREM *Let* B_φ *be the non-complete tableau automaton of MITL*$_\leq$ *formula* φ *and let* A_φ *be the complete tableau automaton. Then* $\mathcal{L}(A_\varphi) = \mathcal{L}(B_\varphi)$*.*

PROOF The two inclusions are shown separately.

- $\mathcal{L}(A_\varphi) \subseteq \mathcal{L}(B_\varphi)$. It can be shown that any run on A_φ is still a run on B_φ, since $L_A \subseteq L_B$, $E_A \subseteq E_B$ and for every $\ell \in L_A$, $Q_A(\ell) = Q_B(\ell)$ and $TC_A(\ell) = TC_B(\ell)$.
- $\mathcal{L}(B_\varphi) \subseteq \mathcal{L}(A_\varphi)$. Let $\overline{\rho}$ be a timed state sequence and let $\overline{\rho} \in \mathcal{L}(B_\varphi)$. Then it follows from lemma 8.1.16 that $\overline{\rho} \models \varphi$ and thus that $\overline{\rho} \in \mathcal{L}(A_\varphi)$. □

On-the-fly tableaux

To define an on-the-fly version of the tableau construction for real-time temporal logic, we introduce a normal form similar to the one used for the untimed tableaux. To deal with the quantitative timing constraints, we need to extend the logic with explicit timers and formulas constrained by timers. We will take the extra labels of the complete tableaux and interpret them as formulas in the extended logic.
Essential to the on-the-fly tableau construction in the untimed case was the separation of constraints on the current state from constraints on the remainder of the state sequence using the $\bigcirc$ operator. In timed state sequences there is no notion of a next moment and there is no $\bigcirc$ operator. We have already seen that an appropriate discretisation is provided by the φ-fine timed state sequences. We introduce a form of $\bigcirc$ operator that can be used for timed state sequences and provides the intended meaning for those timed state sequences that are φ-fine.
Formulas in the extended logic are interpreted within an environment ν that assigns a value to the individual timers. The syntax of the extended logic is given by the following grammar ($\varphi \in \text{MITL}_\leq$, $\chi \in \mathit{TCond}(\Theta)$, $x \in \Theta$, $d \in \mathbb{N}$ for a set Θ of timers):

$$\psi ::= \varphi \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2 \mid \chi \mid \mathit{TS}.\psi \mid \varphi_1 \mathsf{U}_{\leq x} \varphi_2 \mid \varphi_1 \mathsf{V}_{<x} \varphi_2 \mid \bigcirc \psi.$$

We use ψ to range over formulas of the extended logic. We use the notation $\{x := d_1, y := d_2, \ldots\}$ to denote the timer setting $\{(x, d_1), (y, d_2), \ldots\}$, the function assigning the value d_1 to the timer x, d_2 to the timer y, etcetera. The interpretation of the extended logic is the following. We define a relation $\psi \models_\nu \overline{\rho}$ expressing when a timed state sequence $\overline{\rho} = (\overline{\sigma}, \overline{I})$ satisfies a formula ψ in the context of a timer valuation ν:

$\overline{\rho} \models_\nu \varphi$	if $\overline{\rho} \models \varphi$;
$\overline{\rho} \models_\nu \psi_1 \vee \psi_2$	if $\overline{\rho} \models_\nu \psi_1$ or $\overline{\rho} \models_\nu \psi_2$;
$\overline{\rho} \models_\nu \psi_1 \wedge \psi_2$	if $\overline{\rho} \models_\nu \psi_1$ and $\overline{\rho} \models_\nu \psi_2$;
$\overline{\rho} \models_\nu \chi$	if $\nu \models \chi$;
$\overline{\rho} \models_\nu TS.\psi$	if $\overline{\rho} \models_{TS[\nu]} \psi$;
$\overline{\rho} \models_\nu \varphi_1 \mathsf{U}_{\leq x} \varphi_2$	if $\overline{\rho} \models_\nu \varphi_2$ or there is some $0 \leq t \leq \nu(x)$, such that $\overline{\rho}^t \models_{\nu - t} \varphi_2$ and for all $0 \leq t' < t$, $\overline{\rho}^{t'} \models_{\nu - t'} \varphi_1 \vee \varphi_2$ (note that $\nu(x)$ may be smaller than 0, see below);
$\overline{\rho} \models_\nu \varphi_1 \mathsf{V}_{< y} \varphi_2$	if for all $0 \leq t < \nu(y)$, $\overline{\rho}^t \models_{\nu - t} \varphi_2$ or there is some $0 \leq t' < t$, such that $\overline{\rho}^{t'} \models_{\nu - t'} \varphi_1 \wedge \varphi_2$;
$\overline{\rho} \models_\nu \bigcirc \psi$	iff $\overline{\rho}^{\overline{I}(0)} \models_{\nu - \overline{I}(0)} \psi$.

The semantics of the extended formulas is quite straightforward. MITL$_{\leq}$ formulas and $\vee$ and $\wedge$ operators are interpreted as usual. Timer conditions (formulas of the form $x > 0$ or $y \leq 0$) are interpreted by substituting the timers with their values in the timer environment ν. A timer setting TS in the formula $TS.\psi$ binds free timers in ψ to particular values given by the function TS.

The semantics of the Until formula $\varphi_1 \mathsf{U}_{\leq x} \varphi_2$ is similar to the bounded Until formula of MITL$_{\leq}$, but the bound is determined by the value of x in ν. Moreover it holds whenever φ_2 holds, independent of the value of x, which simplifies the treatment in the remainder of this section. $\varphi_1 \mathsf{V}_{< y} \varphi_2$ and $\varphi_1 \mathsf{V}_{< d} \varphi_2$ are similar, but note the strict upper bound in contrast with the MITL$_{\leq}$ Release formula. There is also a $\bigcirc$ operator that is used to take advantage of the discretisation required for the tableau construction and provided by the interval sequence. The meaning of $\bigcirc\psi$ is that ψ holds for the timed state sequence obtained by removing the first interval. Notice that in general, this operator is sensitive to the choice of interval sequence among equivalent timed state sequences. We will only use it with timed state sequences that are fine for the formulas under consideration and in formulas that are insensitive to refinements of interval sequences, i.e. the timed equivalent of stutter-closed formulas. Therefore, this will not be a problem.

Normal form and rewrite rules. The construction is again based upon separating what has to be true now from what has to be true in the future. Since time is dense however, we cannot just separate the current moment from the rest of the time domain. In order to arrive at an appropriate discretisation of the timed states sequence, we look at φ-fine interval sequences. Within the intervals of such a sequence, the valuations of individual subformulas do not change. Now we can unfold our formulas into current (in the current φ-fine interval) and future requirements (from the next interval onwards).

An extended MITL$_{\leq}$ formula is said to be in *disjunctive temporal normal form* (DTNF)

if it is of the form

$$\bigvee_{i=1}^{k} TS_i.(\pi_i \wedge \bigcirc\psi_i).$$

The rewriting rules of the untimed case still apply in the timed case. Additionally we need the following equivalences. $\psi_1 \equiv \psi_2$ denotes that for every timed state sequence $\overline{\rho}$ and timer valuation ν, $\psi_1 \models_\nu \overline{\rho}$ iff $\psi_2 \models_\nu \overline{\rho}$. If conditions are noted along with the equivalence, it means that this is only true for $\overline{\rho}$ and ν that meet those conditions.

- Distribution rules of $\vee$, $\wedge$ and $\bigcirc$ apply as in the untimed case.
- Distribution of timer settings.
  - $(TS_1.\psi_1) \wedge (TS_2.\psi_2) \equiv (TS_1 \cup TS_2).(\psi_1 \wedge \psi_2)$ (if the domains of TS_1 and TS_2 are disjoint)
  - $TS.(\psi_1 \vee \psi_2) \equiv (TS.\psi_1) \vee (TS.\psi_2)$
- Timer instantiation
  - $\varphi_1 \mathsf{U}_{\leq d} \varphi_2 \equiv \{x := d\}.(\varphi_1 \mathsf{U}_{\leq x} \varphi_2)$
- Unfolding
  - $\varphi_1 \mathsf{U}_{\leq x} \varphi_2 \equiv \varphi_2 \vee (x > 0 \wedge \varphi_1 \wedge \bigcirc(\varphi_1 \mathsf{U}_{\leq x} \varphi_2))$ (if $\overline{\rho}$ is both φ_1-fine and φ_2-fine and $\nu(x) \geq 0$)
  - $\varphi_1 \mathsf{V}_{\leq d} \varphi_2 \equiv \varphi_2 \wedge (\varphi_1 \vee \bigcirc(\{y := d\}.\varphi_1 \mathsf{V}_{< y} \varphi_2))$ (if $\overline{\rho}$ is $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$-fine)
  - $\varphi_1 \mathsf{V}_{< y} \varphi_2 \equiv y \leq 0 \vee (\varphi_2 \wedge (\varphi_1 \vee \bigcirc(\varphi_1 \mathsf{V}_{< y} \varphi_2)))$ (if $\overline{\rho}$ is both φ_1- and φ_2-fine)

These equivalences, when interpreted as rewrite rules from left to right, can be used to convert any extended MITL$_\leq$ formula into disjunctive temporal normal form. The formula $p\mathsf{U}_{\leq 5} q$ for instance can be written in normal form via $\{x := 5\}.(p\mathsf{U}_{\leq x} q)$ as $\{x := 5\}.q \vee \{x := 5\}.(x > 0 \wedge p \wedge \bigcirc(p\mathsf{U}_{\leq x} q))$.
However, the unfolding rules introduce new timers. If we want to use this normal form for the construction of a timed automaton we have to make sure that the total number of timers remains finite. To achieve this we make use of the fact that $\varphi_1 \mathsf{U}_{\leq x} \varphi_2 \wedge \varphi_1 \mathsf{U}_{\leq y} \varphi_2 \equiv \varphi_1 \mathsf{U}_{\leq x} \varphi_2$ in any environment ν where $\nu(x) \leq \nu(y)$. Similarly, in such an environment ν, $\varphi_1 \mathsf{V}_{\leq x} \varphi_2 \wedge \varphi_1 \mathsf{V}_{\leq y} \varphi_2 \equiv \varphi_1 \mathsf{V}_{\leq y} \varphi_2$.
Timers related to the Until formula $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ are introduced with a timer setting that assigns to such a timer the value d. If another timer related to this Until formula is active, we know that its value is at most d and thus we can forget about the new timer. Similarly, we can forget about the existing timer in case of $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$. This is expressed by the following equivalences.

- Absorbtion
  - $\varphi_1 \mathsf{U}_{\leq x} \varphi_2 \wedge (\{y := d\}.(\varphi_1 \mathsf{U}_{\leq y} \varphi_2)) \equiv \varphi_1 \mathsf{U}_{\leq x} \varphi_2$ if $\nu(x) \leq d$
  - $\varphi_1 \mathsf{V}_{< x} \varphi_2 \wedge (\{y := d\}.(\varphi_1 \mathsf{V}_{< y} \varphi_2)) \equiv \{y := d\}.(\varphi_1 \mathsf{V}_{< y} \varphi_2)$ if $\nu(x) \leq d$

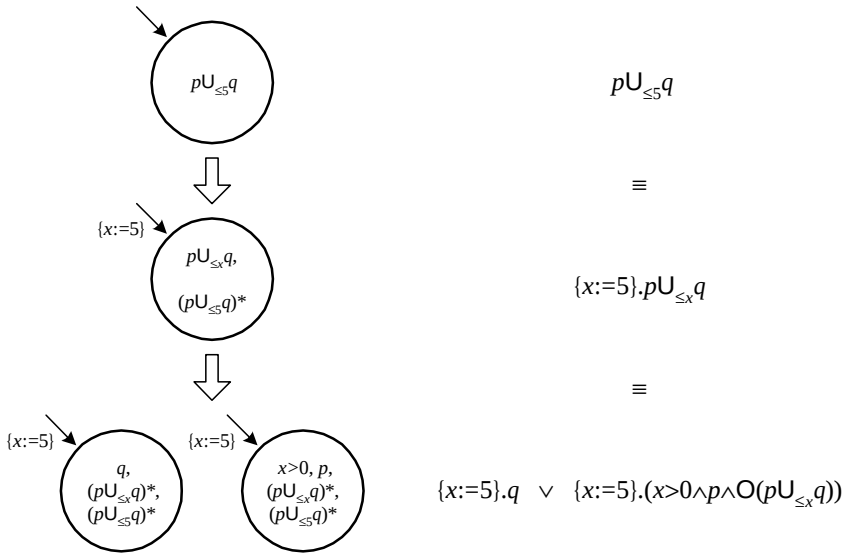


Figure 8.8: Example of the normal form reduction procedure for the timer-setting formula pair $(\varnothing, p\mathsf{U}_{\leq 5}q)$

Constructing the Automaton. Based on the normal form rules and the local and temporal consistency constraints, we can again define a procedure to construct a timed tableau on-the-fly. The main ingredient is, like in the previous chapter, a procedure to obtain from the temporal consistency constraints of a location, a set of edges and new locations that are reachable from that location. In this case we deal not just with sets of formulas that become the new locations, but also with timer settings. These requirements are represented as a set of pairs (TS, Φ) consisting of a required timer update TS and a set Φ of formulas (as in the untimed case).
As an example, figure 8.8 shows the procedure performed on the formula $p\mathsf{U}_{\leq 5}q$ and empty timer setting, i.e. on the pair $(\varnothing, p\mathsf{U}_{\leq 5}q)$. In the first step the timer x is introduced and in the second step, the Until formula is unfolded. As the formulas are processed, they are marked (indicated by an asterisk), as further explained below.

8.1.24 DEFINITION *The* local consistency procedure *that generates a set of consistent terms from a pair (TS, Ψ) is defined as follows.*

- *Let $P_0 = \{(TS, \Psi)\}$.*
- *Then as long as P_n contains a pair with an unmarked formula apply one of the rules of table 8.1 to produce a new set P_{n+1}.*

We use $\mathrm{NF}((TS, \Psi))$ *to denote a result of the procedure*[3] *on* (TS, Ψ), $\mathrm{NF}(\Psi)$ *to denote a result of the procedure performed on the pair* $(\varnothing, \Psi)$ *and* $\mathrm{NF}(\varphi)$ *to denote a result of the procedure performed on the pair* $(\varnothing, \{\varphi\})$. □

[3]The procedure is non-deterministic as it is described here. We assume the existence of a deterministic implementation NF of this procedure.

	Case	$P \cup \{(TS, \Psi \cup \{\psi\})\}$ reduces to:
1	$\psi = \text{false}$	P
2	$\psi = \text{true}, p, \neg p,$ $x > 0, x \leq 0$	$P \cup \{(TS, \Psi \cup \{\psi^*\})\}$
3	$\psi = \psi_1 \vee \psi_2$	$P \cup \{(TS, \Psi \cup \{\psi^*, \psi_1\}), (TS, \Psi \cup \{\psi^*, \psi_2\})\}$
4	$\psi = \psi_1 \wedge \psi_2$	$P \cup \{(TS, \Psi \cup \{\psi^*, \psi_1, \psi_2\})\}$
5	$\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2$	$P \cup \{\Big(\{x_\psi := d\}[TS], \Psi \cup \{\psi^*, \varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2\}\Big)\}$ if $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \notin \Psi$ $P \cup \{\Big(TS, \Psi \cup \{\psi^*, \varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2\}\Big)\}$ if $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \in \Psi$
6	$\psi = \varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$	$P \cup \{(TS, \Psi \cup \{\psi^*, \varphi_2\}), (TS, \Psi \cup \{\psi^*, x_{\psi'} > 0, \varphi_1\})\}$
7	$\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2$	$P \cup \{(TS, \Psi \cup \{\psi^*, \varphi_2\}), (TS, \Psi \cup \{\psi^*, \varphi_1, \varphi_2\})\}$
8	$\psi = \varphi_1 \mathsf{V}_{< y_{\psi'}} \varphi_2$	$P \cup \{(TS, \Psi \cup \{\psi^*, y_{\psi'} \leq 0\}), (TS, \Psi \cup \{\psi^*, \varphi_1, \varphi_2\}),$ $(TS, \Psi \cup \{\psi^*, \varphi_2\})\}$

Table 8.1: Local consistency procedure. If an unmarked formula is added to a set already containing that formula, the formula will be unmarked in the new set. We assume that $\psi \notin \Psi$ and that $(TS, \Psi \cup \{\psi\}) \notin P$

The reduction rules of the procedure correspond closely to the equivalences used for rewriting an extended formula in temporal normal form. The first absorption rule for instance, can be recognised in reduction rule 5; a new timer setting is only introduced if the location does not yet contain $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ (in which case the timer x_ψ is already active and cannot be larger than the corresponding bound d).
The on-the-fly tableau automaton of the MITL$_\leq$ formula φ is the automaton $(L, \Sigma, \Theta, L_0, Q, TC, E)$. The locations (L, L_0) and edges (E) of the automaton are constructed by the algorithm presented in figure 8.9. Furthermore, Σ, Q and TC are defined as in the maximal tableau. Θ is also similar, but contains only one timer per Until or Release formula occurring in φ.

Example OTF Tableau

As an example we construct a tableau for the MITL$_\leq$-formula $\varphi = \Box_{\leq 100} \Diamond_{\leq 5} p$, illustrated with table 8.2 and figure 8.10. We start with the formula $\Box_{\leq 100} \Diamond_{\leq 5} p$; unfolding leads to the pairs:

$$(\{x := 5\}, \{\Box_{\leq 100} \Diamond_{\leq 5} p, \Diamond_{\leq 5} p, \Diamond_{\leq x} p, x > 0\}),$$
$$(\{x := 5\}, \{\Box_{\leq 100} \Diamond_{\leq 5} p, \Diamond_{\leq 5} p, \Diamond_{\leq x} p, p\})$$

These give rise to the initial locations 1 and 2 respectively (the second set of each pair). The temporal consistency requirements for location number 1 are: $(\{y := 100\}, \{\Box_{< y} \Diamond_{\leq 5} p, \Diamond_{\leq x} p\})$. Unfolding these requirements leads to the new pairs

$$(\{y := 100\}, \{\Box_{< y} \Diamond_{\leq 5} p, \Diamond_{\leq x} p, y \leq 0, x > 0\}),$$
$$(\{y := 100\}, \{\Box_{< y} \Diamond_{\leq 5} p, \Diamond_{\leq x} p, y \leq 0, p\}),$$
$$(\{y := 100\}, \{\Box_{< y} \Diamond_{\leq 5} p, \Diamond_{\leq 5} p, \Diamond_{\leq x} p, x > 0\}),$$
$$(\{y := 100\}, \{\Box_{< y} \Diamond_{\leq 5} p, \Diamond_{\leq 5} p, \Diamond_{\leq x} p, p\})$$

```
L0 := {(TS[0], Φ) | (TS, Φ) ∈ NF(φ)}
L := ∅, Edges := ∅
New := {Φ | (ν, Φ) ∈ L0}
while New ≠ ∅ do
  Let Φ ∈ New
  New := New\{Φ}
  L := L∪{Φ}
  for every (TS, Φ') ∈ NF(Next(Φ)) do
    Edges := Edges∪{(Φ, TS, Φ')}
    if Φ' ∉ L then New := New∪{Φ'}
  od
od
```

Figure 8.9: Algorithm for constructing the locations and edges of the on-the-fly tableau automaton

which yield the locations 3, 4, 5 and 6 respectively.
The only consistency requirement of location 2 is, $(\{y := 100\}, \{\Box_{<y}\Diamond_{\leq 5}p\})$, which leads to the pairs

$$\begin{gathered}(\{y := 100\}, \{\Box_{<y}\Diamond_{\leq 5}p, y \leq 0\}), \\ (\{x := 5, y := 100\}, \{\Box_{<y}\Diamond_{\leq 5}p, \Diamond_{\leq 5}p, \Diamond_{\leq x}p, x > 0\}), \\ (\{x := 5, y := 100\}, \{\Box_{<y}\Diamond_{\leq 5}p, \Diamond_{\leq 5}p, \Diamond_{\leq x}p, p\})\end{gathered}$$

the first of which introduces a new location, number 7, and the last two correspond to the locations 5 and 6.
Location 3 has temporal consistency requirements $(\varnothing, \{\Diamond_{\leq x}p\})$, leading to new locations 8 and 9:

$$(\varnothing, \{\Diamond_{\leq x}p, x > 0\}), (\varnothing, \{\Diamond_{\leq x}p, p\}).$$

Location 4 has temporal consistency requirements $(\varnothing, \varnothing)$ and therefore has an edge to location 10, the empty set $\varnothing$. Location 5 has the requirements $(\varnothing, \{\Box_{<y}\Diamond_{\leq 5}p, \Diamond_{\leq x}p\})$ resulting in edges to locations 3, 4, 5 and 6. Location 6 has similar requirements as location 2, $(\varnothing, \{\Box_{<y}\Diamond_{\leq 5}p\})$. Location 7 has no requirements, location 8 the same as 3 and the requirements of locations 9 and 10 are empty. These temporal consistency requirements are summarised in table 8.2. Together, this leads to the timed automaton depicted in figure 8.10.
Some information on the numbers of locations, transitions and timers of tableau automata for different formulas has been collected in table 8.3.

Correctness

Correctness is established along the same lines as in the untimed case in the previous chapter. The automaton is sound since all its locations and edges are consistent. To show that it is complete we demonstrate how an initial run can be constructed for any timed state sequence satisfying φ.

Location	Consistency Requirements	Successor Locations
1	$(\{y := 100\}, \{\Box_{<y}\Diamond_{\leq 5}p, \Diamond_{\leq x}p\})$	3, 4, 5, 6
2	$(\{y := 100\}, \{\Box_{<y}\Diamond_{\leq 5}p\})$	5, 6, 7
3	$(\varnothing, \{\Diamond_{\leq x}p\})$	8, 9
4	$(\varnothing, \varnothing)$	10
5	$(\varnothing, \{\Box_{<y}\Diamond_{\leq 5}p, \Diamond_{\leq x}p\})$	3, 4, 5, 6
6	$(\varnothing, \{\Box_{<y}\Diamond_{\leq 5}p\})$	5, 6, 7
7	$(\varnothing, \varnothing)$	10
8	$(\varnothing, \{\Diamond_{\leq x}p\})$	8, 9
9	$(\varnothing, \varnothing)$	10
10	$(\varnothing, \varnothing)$	10

Table 8.2: Consistent successor locations for the tableau of $\Box_{\leq 100}\Diamond_{\leq 5}p$

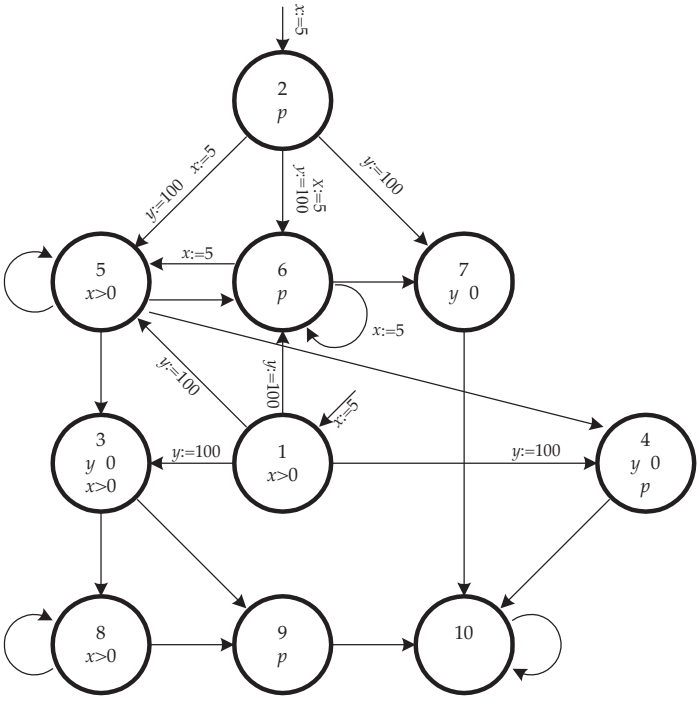


Figure 8.10: Example of on-the-fly tableau for the formula $\Box_{\leq 100}\Diamond_{\leq 5}p$

	Non-Complete	Complete	On-the-fly
$\neg\Diamond_{\leq 5}p$	198/28028/2	3/8/2	4/6/1
$\Box_{\leq 100}\Diamond_{\leq 5}p$	3632/?/4	5/19/4	10/22/2
$\Diamond_{\leq 5}(\Box_{\leq 1}p \vee \Box_{\leq 1}q)$	?/?/6	?/?/6	11/21/3
$p\mathsf{U}_{\leq 1}(q\mathsf{U}_{\leq 1}(r\mathsf{U}_{\leq 1}s))$	?/?/6	?/?/6	14/30/3
$p \Rightarrow (\Box_{\leq 5}(q \Rightarrow \Box_{\leq 1}r))$	?/?/4	?/?/4	15/48/2
$(p \Rightarrow \Diamond_{\leq 5}q)\,\mathsf{U}_{\leq 100}\Box_{\leq 5}\neg p$	?/?/6	?/?/6	21/64/3
$(((p\mathsf{U}_{\leq 4}q)\,\mathsf{U}_{\leq 3}r)\,\mathsf{U}_{\leq 2}s)\,\mathsf{U}_{\leq 1}t$	?/?/8	?/?/8	60/271/4

Table 8.3: Numbers of locations / transitions / timers of different tableaux. We were unable to determine certain figures with the prototype implementation, those have been replaced with '?'.

8.1.25 THEOREM *Let A_φ be the 'on-the-fly' tableau automaton of φ then $\mathcal{L}(A_\varphi) = \mathcal{L}(\varphi)$.*

Soundness and completeness are proved in lemmas 8.1.27 and 8.1.30 respectively. We introduce some shorthand notation. We write $\overline{\rho} \models_\nu \Phi$ instead of " for all $\psi \in \Phi$, $\overline{\rho} \models_\nu \psi$ " and we use $\overline{\rho} \models_\nu (TS, \Phi)$ to denote " for all $\psi \in \Phi$, $\overline{\rho} \models_\nu TS.\psi$ ". Thus, the following is a valid statement: $\overline{\rho} \models_\nu \mathrm{Next}(\Phi)$.

Soundness We can establish that the on-the-fly automaton is sound by checking that its locations and edges are consistent (compare to lemma 7.2.7 of the previous chapter).

8.1.26 LEMMA *Let Φ be a set of extended MITL$_\leq$ formulas and let $(TS', \Phi') \in \mathrm{NF}(\Phi)$. Then Φ' is locally consistent. Moreover, $\Phi \xrightarrow{TS'} \Phi'$ for any $(TS', \Phi') \in \mathrm{NF}(\mathrm{Next}(\Phi))$.*

For the proof, see appendix B.2.1.

8.1.27 LEMMA *Let A_φ be the 'on-the-fly' tableau automaton of φ. Then $\mathcal{L}(A_\varphi) \subseteq \mathcal{L}(\varphi)$.*

PROOF Let B_φ be the non-complete tableau automaton of φ. By lemma 8.1.26, every initial run on A_φ is also an initial run on B_φ and thus every timed state sequence accepted by A_φ satisfies φ. □

Completeness We now show the converse, i.e. that the automaton is also complete. Given a timed state sequence that satisfies φ, we can incrementally construct a run for $\overline{\rho}$ on A_φ. Intuitively, a location corresponds to an extended formula. If a timed state sequence satisfies this formula, there must be a run for this timed state sequence starting from this location. This is so, since all the outgoing edges of the location correspond to a term in the normal form of the 'Next' part of the location. If we know it satisfies the formula, then it must satisfy some term in the normal form, which is the edge that can be taken for the run.
The following lemma tells us that we can find a suitable initial location for the run we want to construct.

8.1.28 LEMMA *Let φ be an MITL$_{\leq}$ formula and $\Phi \subseteq cl(\varphi)$ be a set of extended formulas. Let tsseq be a timed state sequence such that $\overline{\rho} \models_{\nu} \Phi$. Then for any φ-fine interval sequence $\overline{I}$, there is some $(TS, \Phi') \in \mathrm{NF}(\Phi)$ such that $\overline{\rho}^t \models_{TS[\nu]-t} \Phi'$ for all $t \in \overline{I}(0)$ and $\overline{\rho}^{|\overline{I}(0)|} \models_{TS[\nu]-|\overline{I}(0)|} \mathrm{Next}(\Phi')$.*

This lemma is proved in appendix B.2.2.
From this initial location we can continue to find transitions to new locations. If the remainder of the timed state sequence satisfies the temporal consistency constraints of the location, then there is an edge to a new location, where it satisfies the local constraints during the entire interval, as well as the new temporal consistency constraints.

8.1.29 LEMMA *Let Φ be a location in the on-the-fly tableau automaton A_{φ}, let $\overline{\rho}$ and ν be such that $\overline{\rho} \models_{\nu} \mathrm{Next}(\Phi)$ and let $\overline{I}$ be a φ-fine interval sequence. Then there is an edge (Φ, TS', Φ') of A_{φ} such that $\overline{\rho}^t \models_{TS'[\nu]-t} \Phi'$ for all $t \in \overline{I}(0)$ and $\overline{\rho}^{|\overline{I}(0)|} \models_{TS'[\nu]-|\overline{I}(0)|} \mathrm{Next}(\Phi')$.*

PROOF It follows from lemma 8.1.28 and the construction of the on-the-fly tableau. □

This demonstrates that the tableau automaton is complete.

8.1.30 LEMMA *Let A_{φ} be the 'on-the-fly' tableau automaton of φ. Then $\mathcal{L}(\varphi) \subseteq \mathcal{L}(A_{\varphi})$.*

PROOF Let $\overline{\rho}$ be a timed state sequence such that $\overline{\rho} \models \varphi$. It can be shown using lemmas 8.1.28 and 8.1.29 that there is a run for $\overline{\rho}$ on A_{φ}. □

8.2 Unrestricted Real-Time Tableaux

In this section we extend the real-time logic and its tableaux of the previous section to unrestricted interval sequences. The basic principles remain the same, but a lot of extra bookkeeping is required to deal with them. Definitions that are identical to the restricted case are not repeated.

8.2.1 Preliminaries

In the previous section we used timed automata with interval sequences that consist of left-closed and right-open intervals only. We now allow any kind of interval. We need to differentiate between locations that are entered in a left-closed interval and locations that are entered in a left-open interval, as explained in section 8.1. For this, we define a notion of timed automata that are unconventional in the sense that edges of the automaton are labelled with a type stating that the edge is allowed for a transition between a right-open and a left-closed interval only or between a right-open and left-closed interval only. Such a feature is not strictly necessary, but it simplifies things when defining the tableau automata.
Let the set TT of *transition types* be defined as the set $\{\#co, \#oc\}$. *#co* is intended to represent a transition from a right-closed to a left-open interval and *#oc* a transition from a right-open to a left-closed interval.

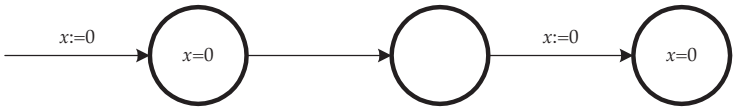


Figure 8.11: Alternating open and singular intervals

8.2.1 DEFINITION *A* timed automaton $A = (L, \Sigma, \Theta, L_0, Q, TC, E)$ *consists of*

- *a finite set* L *of locations;*
- *a finite alphabet* Σ*;*
- *a set* Θ *of timers;*
- *a finite set* L_0 *of initial extended locations* $(\ell_0, \nu_0) \in L \times TVal(\Theta)$*;*
- *a mapping* $Q : L \to 2^{\Sigma}$ *labelling every location with a set of symbols of the alphabet;*
- *a mapping* $TC : L \to 2^{TCond(\Theta)}$*, labelling every location with a set of timer constraints over timers in* Θ*;*
- *a set* $E \subseteq L \times TSet(\Theta) \times \mathrm{TT} \times L$ *of edges.* $(\ell, TS, tt, \ell') \in E$ *denotes an edge from location* ℓ *to location* ℓ'*, labelled with a timer set operation* TS *and a transition type* tt*.* □

Note that the special semantics using transition types is only added for convenience. It can be removed by the same trick as applied in [5, 9]. An extra timer is used to enforce the interval sequence of a run to alternate open intervals ((a, b)) and singular intervals ($[a, a]$). For a singular location, the extra timer x is set to 0 when the location is entered and the location contains the timer constraint $x = 0$ (see figure 8.11). Then the automaton can only reside in such location for a single instant in time. Open-interval locations are constrained to occur between singular locations and can therefore only correspond to open intervals.
A run on such an automaton with transition types is now required to match the transitions of intervals to the transition types of the edges.

8.2.2 DEFINITION *A* timed run $\bar{r}$ *for timed word* $\bar{\upsilon}$ *on timed automaton* A *is a triple* $\left(\overline{\ell}, \overline{I}, \overline{\nu}\right)$ *consisting of a sequence of locations, an interval sequence and a sequence of timer valuations, such that*

- *[Symbol match] for all* $t \geq 0$*,* $\bar{\upsilon}(t) \in Q(\bar{r}(t))$*;*
- *[Consecution] for all* $k \geq 0$ *there is an edge* $\left(\overline{\ell}(k), TS_k, tt_k, \overline{\ell}(k+1)\right) \in E$ *such that* $\overline{\nu}(k+1) = TS_k[\overline{\nu}(k) - |\overline{I}(k)|]$ *and* tt_k *matches the transition from* $\overline{I}(k)$ *to* $\overline{I}(k+1)$*;*
- *[Timing] for all* $k \geq 0$*, every* $t \in \overline{I}(k)$ *and every* $\chi \in TC(\overline{\ell}(k))$*,* $\overline{\nu}(k) - (t - l(\overline{I}(k)) \models \chi$*.* □

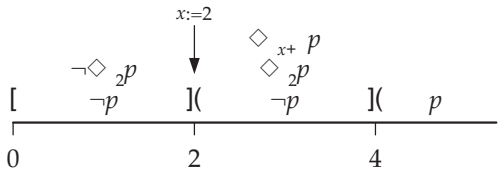


Figure 8.12: Timed state sequence requiring $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2$-label

Also these automata cannot discriminate between equivalent timed words. Furthermore, it is still true that any suffix of a timed run is also a timed run (see section 2.6.2).

Drawing conventions In the illustrations of these automata we use the same conventions as before. Moreover, we denote the type of a transition (*#oc* or *#co*) alongside of the arrow. If no transition type is shown along the arrow, then it represents two edges, both a corresponding edge labelled with *#oc* and an edge labelled with *#co*.

8.2.2 Tableaux for Real-time Logic

In this section we define the construction of a timed automaton from an MITL$_{\leq}$ formula φ that accepts precisely all models that satisfy φ, interpreted over unrestricted interval sequences. This is done by extending the method applied in the restricted case.

Intuition behind the construction

The principles behind the real-time tableau are the same as in the case of the 'left-closed tableau'. Due to the problem of open and closed intervals, we need to distinguish between two types of labels: those that are started at open and those that are started at closed intervals.
In the restricted case, we used a label $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ to denote that $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ must hold for the instant t when the timer was started. Suppose now that $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ must hold in some left-open interval and the timer x_ψ was set to d at the start of this interval. Then requiring that $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ holds is too strong, as demonstrated by the timed state sequence in figure 8.12[4]. If we want to check that $\Diamond_{\leq 2} p$ holds during the second interval $((2, 4])$ then we set the timer x to 2 at $t = 2$. One can check that in the presented timed state sequence, $\Diamond_{\leq 2} p$ holds for every instant in the second interval, but p holds only after timer x expires.

[4]The tableau of [5, 9] is not complete. Assume (using notation of [5, 9]) we have the *MITL* formula $\psi = \Diamond_{(0,1]} p$. Consider an incoming edge to an open-interval location containing ψ from a location not containing x_ψ. Then the timer is allowed to be reset upon entering the location. From that point onwards, subsequent locations contain x_ψ and $x_\psi \in (0, 1]$ until a location is reached containing p and the timer x_ψ is not reset in the mean time. Thus by the time when $x_\psi = 1$, a location containing p must have been reached, otherwise the run is stuck. But the formula holds in the initial open interval if p becomes true immediately after this moment.

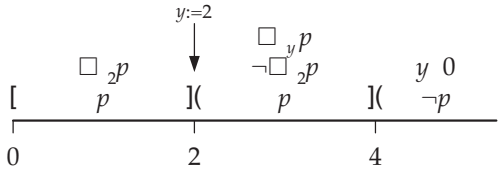


Figure 8.13: Timed state sequence requiring $\neg\ (\varphi_1 \mathsf{U}_{\leq y_\psi} \varphi_2)$-label

In general, φ_2 must become true at the latest immediately after the timer expires. To be precise, for every $\epsilon > 0$, φ_2 must become true within $x_\psi + \epsilon$ units of time. Hence, to denote this we use the following label: $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2$. Similarly, we need the extra label $\neg \left(\varphi_1 \mathsf{U}_{\leq y_\psi} \varphi_2\right)$ for timing constraints on Release formulas if the interval during which $\neg (\varphi_1 \mathsf{U}_{\leq d} \varphi_2)$ must be checked is right-closed. For instance, in the timed state sequence shown in figure 8.13, $\Box_{\leq 2} p$ must be checked during the first interval. At $t = 2$ the timer y is set to 2. Then p must remain satisfied until $y < 0$ (as opposed to $y \leq 0$ in the restricted case). This is expressed by the new label $\Box_{\leq y} p$ (and $\neg \left(\varphi_1 \mathsf{U}_{\leq y_\psi} \varphi_2\right)$ in general). The new labels bring forth the new timer constraints $x_\psi \geq 0$ and $y_\psi < 0$.
Figure 8.14 shows the familiar pattern for checking the Until formula $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$. Notice that the pattern is very similar to the restricted case with the exception of the duplication of the non-trivial $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ sets. These sets contain either the label $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ or the label $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2$ and which one of these locations is entered depends on the type of transition, as explained above. For the rest of the pattern, the types of transitions are irrelevant.
The pattern for Release formulas is shown in figure 8.15. Also this pattern is very similar to the restricted case. As the label $\neg \left(\varphi_1 \mathsf{U}_{< y_\psi} \varphi_2\right)$ occurs in all locations in the restricted setting, all locations are now duplicated containing either $\neg \left(\varphi_1 \mathsf{U}_{< y_\psi} \varphi_2\right)$ or $\neg \left(\varphi_1 \mathsf{U}_{\leq y_\psi} \varphi_2\right)$. The corresponding timer is activated on a transition from a non-trivial $\neg (\varphi_1 \mathsf{U}_{\leq d} \varphi_2)$ location to a non-trivial $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ location. If this transition is from a right-open to a left-closed interval then the edge leads to a location labelled with $\neg \left(\varphi_1 \mathsf{U}_{< y_\psi} \varphi_2\right)$, otherwise it leads to a location labelled with $\neg \left(\varphi_1 \mathsf{U}_{\leq y_\psi} \varphi_2\right)$. For the other transitions, the type is irrelevant.
We do not attempt to draw the combined pattern for $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ and $\neg \varphi_1 \mathsf{U}_{\leq d} \varphi_2$ (such as figures 7.2 for the untimed case and figure 8.4 for the restricted timed case) for the resulting image would be too cluttered to be readable.

Definition of the Tableau Automaton

We proceed with the precise definition of the complete tableau automaton, by extending the familiar concepts further to deal with the extra labels and transition types. These definitions are extended after the patterns displayed in figures 8.14 and 8.15 and discussed in the previous section. The closure set includes the new labels and timer constraints.

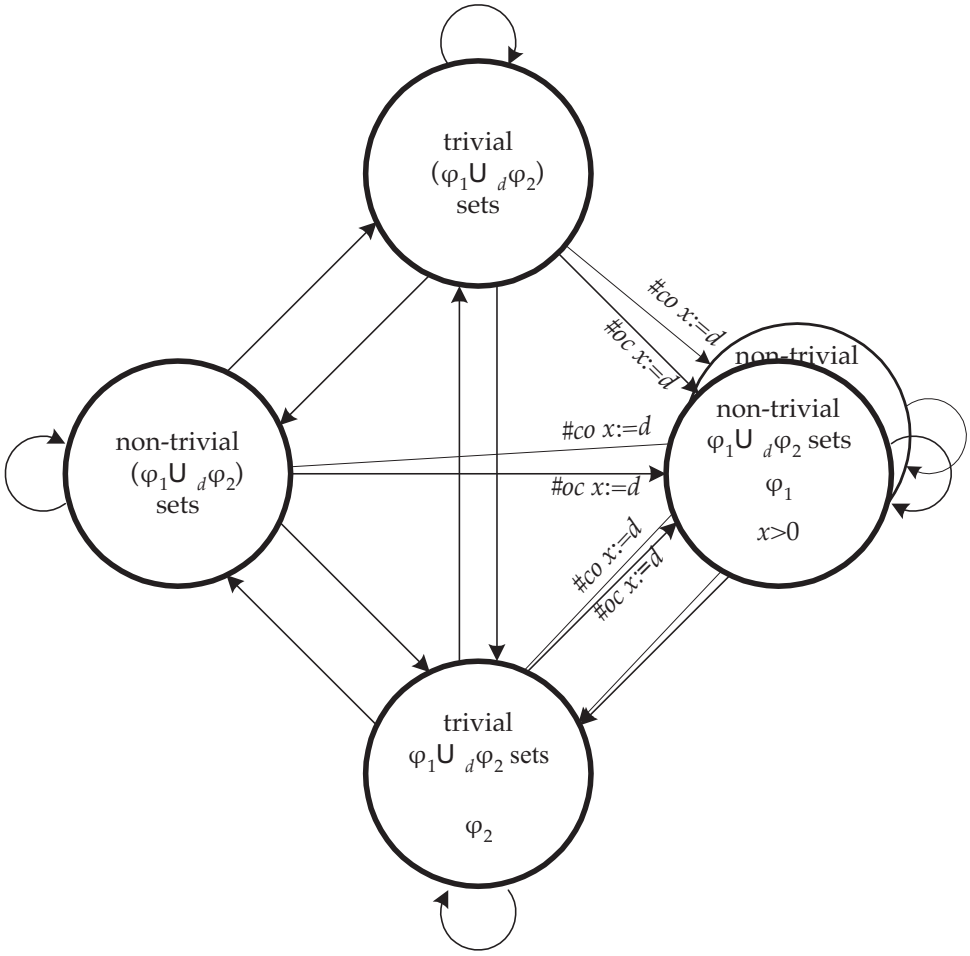


Figure 8.14: Pattern for the verification of $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$

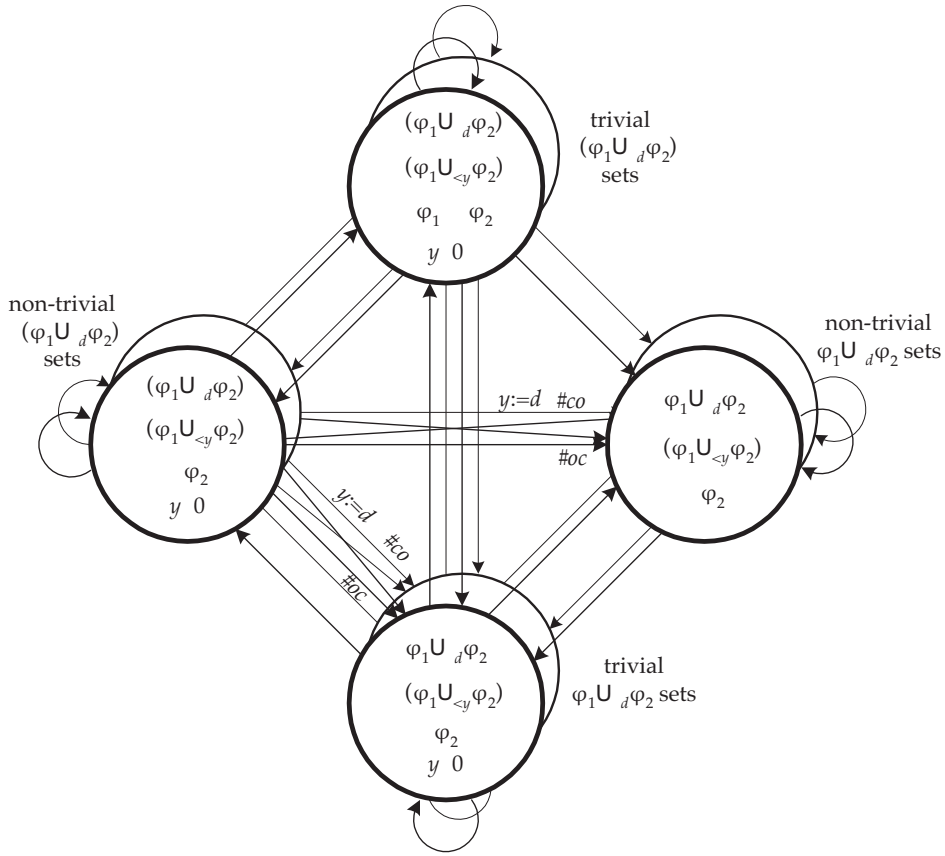


Figure 8.15: Pattern for the verification of $\neg\ (\varphi_1 \mathsf{U}_{\leq d} \varphi_2)$

8.2.3 DEFINITION *The* closure $cl(\varphi)$ *of an MITL*$_{\leq}$ *formula* φ *is the smallest set* Φ *of formulas and labels, such that*

- $sub(\varphi) \subseteq \Phi$*;*
- *if* $\psi \in sub(\varphi)$*, then* $\neg\psi \in \Phi$*;*
- *for every Until formula* $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in \Phi$*, the* labels

$$\{\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2, x_\psi > 0, \varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2, x_\psi \geq 0,$$

$$\neg\left(\varphi_1 \mathsf{U}_{<y_\psi} \varphi_2\right), y_\psi \leq 0, \neg\left(\varphi_1 \mathsf{U}_{\leq y_\psi} \varphi_2\right), y_\psi < 0\} \subseteq \Phi. \qquad \square$$

Local consistency is redefined for the different types of labels. Only the rules 4, 5.(b) and 7.(b) are changed or new, compared to the corresponding definition (8.1.4) in the restricted case.

8.2.4 DEFINITION *A set* Φ *is called* locally consistent *if*

1. $\text{false} \notin \Phi$*;*
2. *if* $\psi_1 \vee \psi_2 \in \Phi$ *then* $\psi_1 \in \Phi$ *or* $\psi_2 \in \Phi$*;*
3. *if* $\psi_1 \wedge \psi_2 \in \Phi$ *then* $\psi_1 \in \Phi$ *and* $\psi_2 \in \Phi$*;*
4. *if* $\varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in \Phi$ *then either* $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \in \Phi$ *or* $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2 \in \Phi$ *(but not both);*
5. *(a) if* $\varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2 \in \Phi$ *then* $\varphi_2 \in \Phi$ *or both* $x_{\psi'} > 0 \in \Phi$ *and* $\varphi_1 \in \Phi$*;*
   (b) if $\varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2 \in \Phi$ *then* $\varphi_2 \in \Phi$ *or both* $x_{\psi'} \geq 0 \in \Phi$ *and* $\varphi_1 \in \Phi$*;*
6. *if* $\varphi_1 \mathsf{V}_{\leq d} \varphi_2 \in \Phi$ *then* $\varphi_2 \in \Phi$*;*
7. *(a) if* $\varphi_1 \mathsf{V}_{<y_{\psi'}} \varphi_2 \in \Phi$ *then* $y_{\psi'} \leq 0 \in \Phi$ *or* $\varphi_2 \in \Phi$*;*
   (b) if $\varphi_1 \mathsf{V}_{\leq y_{\psi'}} \varphi_2 \in \Phi$ *then* $y_{\psi'} < 0 \in \Phi$ *or* $\varphi_2 \in \Phi$*.* $\square$

The notion of non-trivial set is extended to cover the new labels in a way that is very similar to the existing labels.

8.2.5 DEFINITION *If* ψ *is a formula or label of one of the forms* $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$*,* $\varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$*,* $\varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2$*,* $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$*,* $\varphi_1 \mathsf{V}_{<y_{\psi'}} \varphi_2$ *or* $\varphi_1 \mathsf{V}_{\leq y_{\psi'}} \varphi_2$ *then a* ψ*-set* Φ *is called non-trivial for* ψ

- *if* $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2$ *and* $\varphi_2 \notin \Phi$*;*
- *if* $\psi = \varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$ *and* $\varphi_2 \notin \Phi$*;*
- *if* $\psi = \varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2$ *and* $\varphi_2 \notin \Phi$*;*
- *if* $\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2$ *and* $\varphi_1 \notin \Phi$*;*
- *if* $\psi = \varphi_1 \mathsf{V}_{<y_{\psi'}} \varphi_2$*,* $\varphi_1 \notin \Phi$ *and* $y_{\psi'} \leq 0 \notin \Phi$*;*

- *if* $\psi = \varphi_1 \mathsf{V}_{\leq y_{\psi'}} \varphi_2$, $\varphi_1 \notin \Phi$ *and* $y_{\psi'} < 0 \notin \Phi$. □

Completeness of sets is redefined as well in order to deal with the added labels.

8.2.6 DEFINITION *A set* Φ *is called* complete *w.r.t. the set* L *of formulas and labels if*

1. *for every MITL*$_{\leq}$ formula $\psi \in L$, *precisely one of* ψ *and* $\neg\psi$ *is in* Φ;
2. *exactly one of* $\varphi_1 \mathsf{V}_{< y_{\neg\psi}} \varphi_2$ *and* $\varphi_1 \mathsf{V}_{\leq y_{\neg\psi}} \varphi_2$ *is in* Φ *for every* $\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2 \in L$;
3. *exactly one of* $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ *and* $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2$ *is in* Φ *if* $\psi \in \Phi$, *and neither of them if* $\psi \notin \Phi$, *for every* $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in L$;
4. (a) $x_{\psi'} > 0$ *is in* Φ *iff* Φ *is non-trivial for* $\psi = \varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2 \in L$;
   (b) $x_{\psi'} \geq 0$ *is in* Φ *iff* Φ *is non-trivial for* $\psi = \varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2 \in L$;
5. (a) $y_\psi \leq 0$ *is in* Φ *iff* Φ *is not non-trivial for* $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in L$ *and* $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2$ *in* Φ;
   (b) $y_\psi < 0$ *is in* Φ *iff* Φ *is not non-trivial for* $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in L$ *and* $\varphi_1 \mathsf{V}_{\leq y_\psi} \varphi_2$ *in* Φ. □

Temporal consistency between sets of formulas and labels and timer settings depends now not only on the sets of formulas and labels and the timer setting, but also on the type of transition (*#oc* or *#co*) as explained earlier in this section. If a particular label is introduced, the required version of the label depends on the type of the transition.

8.2.7 DEFINITION *A set* Φ *is* temporally consistent *with the set* Φ' *under the timer setting TS and transition type tt if for every bounded Until formula* $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2$

1. (a) *if* Φ *is non-trivial for* $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ *then* $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \in \Phi'$ *and* x_ψ *is undefined in TS;*
   (b) *if* Φ *is non-trivial for* $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2$ *then* $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2 \in \Phi'$ *and* x_ψ *is undefined in TS;*
2. (a) *if* Φ *is not non-trivial for* $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ *and* Φ' *is, then* $TS(x_\psi) = d$ *and* $tt =$ *#oc;*
   (b) *if* Φ *is not non-trivial for* $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2$ *and* Φ' *is, then* $TS(x_\psi) = d$ *and* $tt =$ *#co;*

and for every bounded Release formula $\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2$

3. *if* Φ *is non-trivial for* $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$ *then*
   (a) $\varphi_1 \mathsf{V}_{\leq d} \varphi_2 \in \Phi'$ *or*
   (b) $tt =$ *#oc*, $TS(y_{\neg\psi}) = d$ *and* $\varphi_1 \mathsf{V}_{< y_{\neg\psi}} \varphi_2 \in \Phi'$ *or*
   (c) $tt =$ *#co*, $TS(y_{\neg\psi}) = d$ *and* $\varphi_1 \mathsf{V}_{\leq y_{\neg\psi}} \varphi_2 \in \Phi'$;
4. (a) *if* Φ *is non-trivial for* $\varphi_1 \mathsf{V}_{< y_{\neg\psi}} \varphi_2$ *then* $\varphi_1 \mathsf{V}_{< y_{\neg\psi}} \varphi_2 \in \Phi'$;

(b) if Φ *is non-trivial for* $\varphi_1 \mathsf{V}_{\leq y_{\neg\psi}} \varphi_2$ *then* $\varphi_1 \mathsf{V}_{\leq y_{\neg\psi}} \varphi_2 \in \Phi'$.

Temporal consistency is denoted $\Phi \xrightarrow{TS,tt} \Phi'$. □

Finally, we have to refine the definition of temporal consistency constraints, by means of the function Next. The set of consistency constraints is now sensitive to the type of the transition to the next interval as well.

8.2.8 DEFINITION *Let* Φ *be a set of formulas and labels and let* tt *be a transition type. Then* $\mathrm{Next}(\Phi, tt)$ *is the triple* (TS, tt, Φ') *where* TS *and* Φ' *are the smallest sets such that*

- *if* Φ *is non-trivial for* $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$, *then* Φ' *contains* $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$;
- *if* Φ *is non-trivial for* $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2$, *then* Φ' *contains* $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2$;
- *if* Φ *is non-trivial for* $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$ *and* $tt = \#oc$, *then* Φ' *contains* $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2$ *and* TS *contains* $y_\psi := d$;
- *if* Φ *is non-trivial for* $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$ *and* $tt = \#co$, *then* Φ' *contains* $\varphi_1 \mathsf{V}_{\leq y_\psi} \varphi_2$ *and* TS *contains* $y_\psi := d$;
- *if* Φ *is non-trivial for* $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2$, *then* Φ' *contains* $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2$;
- *if* Φ *is non-trivial for* $\varphi_1 \mathsf{V}_{\leq y_\psi} \varphi_2$, *then* Φ' *contains* $\varphi_1 \mathsf{V}_{\leq y_\psi} \varphi_2$. □

We use $\mathrm{Next}(\Phi)$ to denote the consistency constraints of both types together,

$$\mathrm{Next}(\Phi) = \{\mathrm{Next}(\Phi, \#oc), \mathrm{Next}(\Phi, \#co)\}.$$

Again, $\mathrm{Next}(\Phi, tt)$ yields the necessary timer setting and formulas to be a temporally consistent successor of Φ under the timer setting and the transition type tt. The normal form procedure in the on-the-fly construction that will follow, maintains this temporal consistency. All the following definitions are analogous to the restricted case, but are now possibly sensitive to the transition type. With these new definitions we can define the tableau automaton for the MITL$_\leq$ formula φ for arbitrary interval sequences.

8.2.9 DEFINITION *The complete* tableau automaton A_φ *of MITL*$_\leq$ *formula* φ *is the timed automaton* $(L, \Sigma, \Theta, L_0, Q, TC, E)$ *where*

- L *contains all sets that are locally consistent and complete w.r.t.* φ;
- $\Sigma = 2^{Prop(\varphi)}$;
- L_0 *is the set of all extended locations* (Φ_0, ν_0) *such that* $\Phi_0 \in L$, $\varphi \in \Phi_0$, $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2 \notin \Phi_0$ *and* $\nu_0(x_\psi) = d$ *and* $\nu_0(y_\psi) = 0$ *for every bounded Until formula* $\varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in cl(\varphi)$;
- Q *is the mapping that assigns to the location* Φ, *the set of all states* $\sigma \in 2^{Prop}$ *such that*

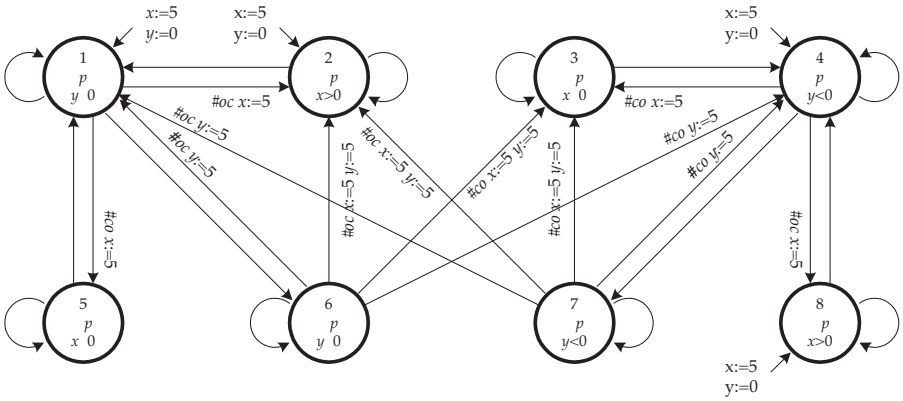


Figure 8.16: Example of the unrestricted tableaux of the formula $\Diamond_{\leq 5} p$

- $p \in \sigma$ *if* $p \in \Phi$,
- $p \notin \sigma$ *if* $\neg p \in \Phi$;

- *TC is the mapping that assigns to the location* Φ*, the set of all timer conditions* $\chi \in \Phi$*;*

- *E is the set of all edges* (Φ, TS, tt, Φ') *such that* $\Phi \xrightarrow{TS,tt} \Phi'$ *(only considering timer settings of the form* $x_\psi := d$ *or* $y_\psi := d$ *for bounded Until formulas* $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2$ *in* $cl(\varphi)$*).* □

This definition is straightforward and similar to the definition of the previous complete tableau automata. Note that if the initial location is non-trivial for some Until formula $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$, then it contains $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ and not $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2$, since every timed state sequence starts with a left-closed interval.

8.2.3 Example

As an example, the unrestricted complete tableaux automaton for the formula $\Diamond_{\leq 5} p$ is shown in figure 8.16. The locations of the automaton are the following locally consistent and complete subsets of $cl(\Diamond_{\leq 5} p) = \{\Diamond_{\leq 5} p, \neg\Diamond_{\leq 5} p, \Diamond_{\leq x} p, \Diamond_{\leq x+\epsilon} p, \neg\Diamond_{<y} p, \neg\Diamond_{\leq y} p, p, \neg p, \text{true}, \text{false}\}$ (using x to denote $x_{\Diamond_{\leq 5} p}$ and y to denote $y_{\Diamond_{\leq 5} p}$).

$$
\begin{aligned}
\ell_1 &= \{\Diamond_{\leq 5} p, \neg\Diamond_{<y} p, y \leq 0, p, \text{true}\} \\
\ell_2 &= \{\Diamond_{\leq 5} p, \Diamond_{\leq x} p, \neg\Diamond_{<y} p, x > 0, \neg p, \text{true}\} \\
\ell_3 &= \{\Diamond_{\leq 5} p, \Diamond_{\leq x+\epsilon} p, \neg\Diamond_{\leq y} p, x \geq 0, \neg p, \text{true}\} \\
\ell_4 &= \{\Diamond_{\leq 5} p, \neg\Diamond_{\leq y} p, p, y < 0, \text{true}\} \\
\ell_5 &= \{\Diamond_{\leq 5} p, \Diamond_{\leq x+\epsilon} p, \neg\Diamond_{<y} p, x \geq 0, \neg p, \text{true}\} \\
\ell_6 &= \{\neg\Diamond_{\leq 5} p, \neg\Diamond_{<y} p, y \leq 0, \neg p, \text{true}\} \\
\ell_7 &= \{\neg\Diamond_{\leq 5} p, \neg\Diamond_{\leq y} p, y < 0, \neg p, \text{true}\} \\
\ell_8 &= \{\Diamond_{\leq 5} p, \Diamond_{\leq x} p, \neg\Diamond_{\leq y} p, x > 0, \neg p, \text{true}\}
\end{aligned}
$$

The locations show only those formulas and labels that determine their labelling. The edges are determined by temporal consistency constraints. The symmetrical

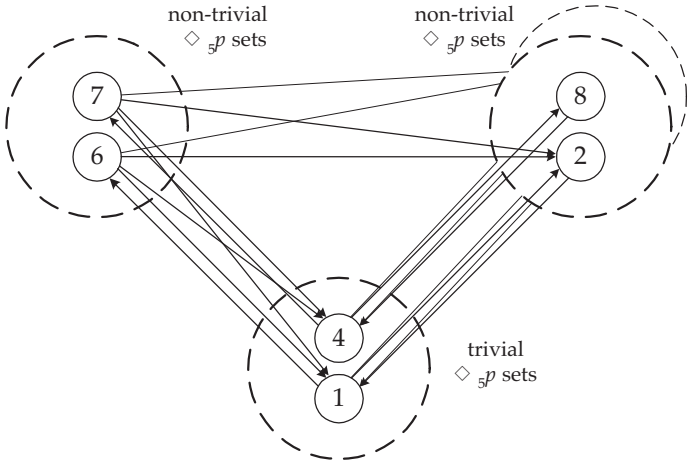


Figure 8.17: Correspondence between the tableau of formula $\Diamond_{\leq 5}p$ and the general pattern

parts on the left and on the right represent the locations labelled with $\neg\Diamond_{<y}p$ and $\neg\Diamond_{\leq y}p$ respectively. Upon leaving location 6 or 7, a choice is made between the two sides, depending on the type of transition. The initial extended locations are for every $k \in \{1,2,4,8\}$: $(\{x := 5, y := 0\}, \ell_k)$. In figure 8.17 the locations of the tableau are arranged according to the general pattern related to $\Diamond_{\leq 5}p$ (see figure 8.14). There are no trivial $\neg\Diamond_{\leq 5}p$ sets, since they would contain false. The labelling of the edges and locations has been left out and can be found in figure 8.16.

8.2.4 Correctness

8.2.10 THEOREM *The tableau automaton A_φ accepts precisely the models of MITL$_\leq$ formula φ.*

PROOF Follows immediately from the lemmas 8.2.19 and 8.2.21 showing soundness and completeness of the tableau respectively. □

Soundness We start by showing that Until formulas are verified correctly. The general idea is displayed in figure 8.18. Any location that is labelled with $\varphi_1 \mathsf{U}_{\leq d}\varphi_2$ corresponds to one of the three kinds of locations on the left. It is trivial (the middle one), it is non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi}\varphi_2$ (the top one if the non-trivial situation was entered in a left-closed fashion, covered by lemma 8.2.11) or it is non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon}\varphi_2$ (the bottom one if it was entered in a left-open fashion, covered by lemma 8.2.12). The next two lemmas correspond to lemma 8.1.11 in the restricted case.

8.2.11 LEMMA *Let $\bar{r} = (\overline{\Psi}, \overline{I}, \overline{\nu})$ be a timed run for $\overline{\rho}$ on A_φ and let $\psi = \varphi_1 \mathsf{U}_{\leq d}\varphi_2 \in cl(\varphi)$. If $\overline{\Psi}(k)$ is non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi}\varphi_2$, then we have (i) $\varphi_1 \in \overline{\Psi}(k)$, (ii) $x_\psi > 0 \in \overline{\Psi}(k)$, (iii) $\overline{\nu}(k+1)(x_\psi) = \overline{\nu}(k)(x_\psi) - |\overline{I}(k)|$ and (iv) either $\varphi_2 \in \overline{\Psi}(k+1)$ or $\overline{\Psi}(k+1)$ is non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi}\varphi_2$.*

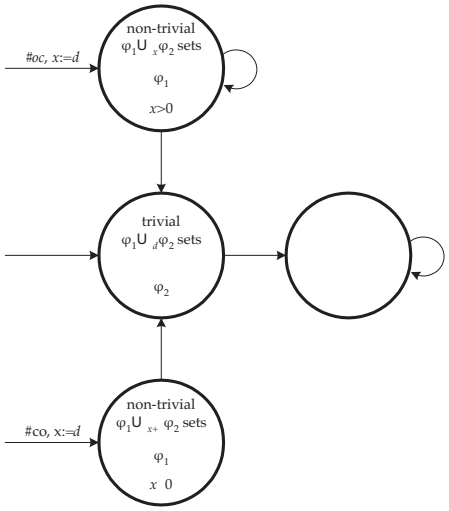


Figure 8.18: Verification of $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$

PROOF That $\varphi_1 \in \overline{\Psi}(k)$ and $x_\psi > 0 \in \overline{\Psi}(k)$ follows from the fact that $\overline{\Psi}(k)$ is non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ and local consistency (5a). From temporal consistency (1a) it follows that timer x_ψ is not set in the transition from $\overline{\Psi}(k)$ to $\overline{\Psi}(k+1)$ and thus $\overline{\nu}(k+1)(x_\psi) = \overline{\nu}(k)(x_\psi) - |\overline{I}(k)|$. Finally, it follows from temporal consistency (1a) that either $\varphi_2 \in \overline{\Psi}(k+1)$ or $\overline{\Psi}(k+1)$ is non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$. □

Almost identical to the previous lemma, but now for $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2$ we demonstrate the following lemma.

8.2.12 LEMMA *Let $\bar{r} = (\overline{\Psi}, \overline{I}, \overline{\nu})$ be a timed run for $\overline{\rho}$ on A_φ and let $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in cl(\varphi)$. If $\overline{\Psi}(k)$ is non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2$, then we have (i) $\varphi_1 \in \overline{\Psi}(k)$, (ii) $x_\psi \geq 0 \in \overline{\Psi}(k)$, (iii) $\overline{\nu}(k+1)(x_\psi) = \overline{\nu}(k)(x_\psi) - |\overline{I}(k)|$ and (iv) either $\varphi_2 \in \overline{\Psi}(k+1)$ or $\overline{\Psi}(k+1)$ is non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2$.*

PROOF That $\varphi_1 \in \overline{\Psi}(k)$ and $x_\psi \geq 0 \in \overline{\Psi}(k)$ follows from the fact that $\overline{\Psi}(k)$ is non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2$ and local consistency (5b). From temporal consistency (1b) it follows that timer x_ψ is not set in the transition from $\overline{\Psi}(k)$ to $\overline{\Psi}(k+1)$ and thus $\overline{\nu}(k+1)(x_\psi) = \overline{\nu}(k)(x_\psi) - |\overline{I}(k)|$. Finally, it follows from temporal consistency (1b) that either $\varphi_2 \in \overline{\Psi}(k+1)$ or $\overline{\Psi}(k+1)$ is non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2$. □

We can now prove the lemma that tells us that Until formulas are verified correctly.

8.2.13 LEMMA *Let $\bar{r}$ be a timed run for $\overline{\rho}$ on A_φ and let $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in \bar{r}(0)$. Then there is some $t \leq d$ such that $\varphi_2 \in \bar{r}(t)$ and for all $0 \leq t' < t$, $\varphi_1 \in \bar{r}(t')$ or $\varphi_2 \in \bar{r}(t')$.*

PROOF Let $\bar{r} = (\overline{\Psi}, \overline{I}, \overline{\nu})$. If $\psi \in \overline{\Psi}(0)$ then one of the following is true:

- $\overline{\Psi}(0)$ is trivial for ψ. Then $\varphi_2 \in \overline{\Psi}(0)$ and the lemma follows trivially, taking $t = 0$.

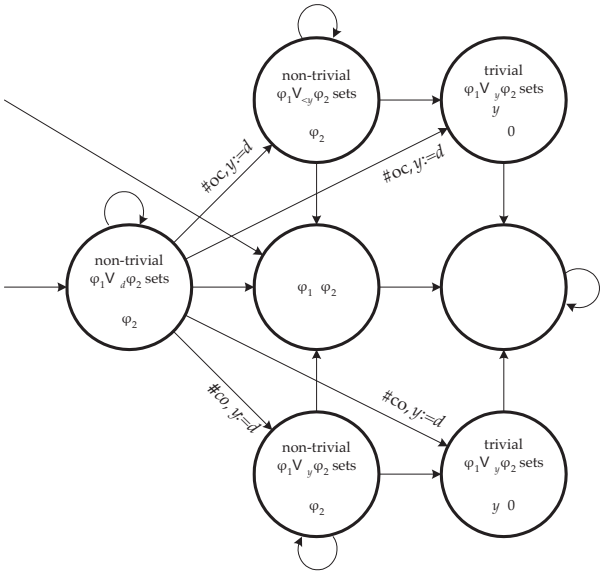


Figure 8.19: Verification of $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$

- $\overline{\Psi}(0)$ is non-trivial for ψ and $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \in \overline{\Psi}(0)$. Then one can show using lemma 8.2.11 and knowing that $\overline{\nu}(0)(x_\psi) \leq d$, that there are some $k \geq 0$ and $t \in \overline{I}(k)$ such that $t \leq d$, $\varphi_2 \in \overline{\Psi}(k)$ and for all $0 \leq m < k$, $\varphi_1 \in \overline{\Psi}(m)$. Thus, $\varphi_2 \in \bar{r}(t)$ and for all $0 \leq t' < t$, $\varphi_1 \in \bar{r}(t')$ or $\varphi_2 \in \bar{r}(t')$, and the lemma follows.
- $\overline{\Psi}(0)$ is non-trivial for ψ and $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2 \in \overline{\Psi}(0)$. Then one can show using lemma 8.2.12 and knowing that $\overline{\nu}(0)(x_\psi) < d$, that there are some $k \geq 0$ and $t \in \overline{I}(k)$ such that $t < d$, $\varphi_2 \in \overline{\Psi}(k)$ and for all $0 \leq m < k$, $\varphi_1 \in \overline{\Psi}(m)$. Thus, $\varphi_2 \in \bar{r}(t)$ and for all $0 \leq t' < t$, $\varphi_1 \in \bar{r}(t')$ or $\varphi_2 \in \bar{r}(t')$, and the lemma follows. □

The next three lemmas help us to show the same for Release formulas along the lines of figure 8.19. A location labelled as a $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$ set is either trivial (the centre locations of the figure) or non-trivial (the left-most locations). Lemma 8.2.14 will show that a run, starting from a non-trivial $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$ location, ends up in a trivial $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$ location or in either a non-trivial $\varphi_1 \mathsf{V}_{<y_\psi} \varphi_2$ or a non-trivial $\varphi_1 \mathsf{V}_{\leq y_\psi} \varphi_2$ location with timer y_ψ set to d. Lemmas 8.2.15 and 8.2.16 respectively will take it from there for the respective cases.

8.2.14 LEMMA *Let $\bar{r} = (\overline{\Psi}, \overline{I}, \overline{\nu})$ be a timed run for $\overline{\rho}$ on A_φ and let $\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2 \in cl(\varphi)$. If $\overline{\Psi}(k)$ is non-trivial for ψ, then one of the following is true: (i) $\overline{\Psi}(k+1)$ is trivial for ψ, (ii) $\overline{\Psi}(k+1)$ is non-trivial for ψ or (iii) $\overline{\nu}(k+1)(y_{\neg\psi}) = d$ and $\varphi_1 \mathsf{V}_{<y_{\neg\psi}} \varphi_2 \in \overline{\Psi}(k+1)$ if $\overline{I}(k)$ is right-open and $\varphi_1 \mathsf{V}_{\leq y_{\neg\psi}} \varphi_2 \in \overline{\Psi}(k+1)$ if $\overline{I}(k)$ is right-closed.*

PROOF From temporal consistency (3a,b,c) it follows that $\overline{\Psi}(k+1)$ is a ψ set or $\overline{\Psi}(k+1)$ contains either $\varphi_1 \mathsf{V}_{<y_{\neg\psi}} \varphi_2$ or $\varphi_1 \mathsf{V}_{\leq y_{\neg\psi}} \varphi_2$ depending on the type of transition. In the former case it can be either trivial (i) or non-trivial (ii) for ψ. In the latter case, $\overline{\nu}(k+1)(y_{\neg\psi}) = d$ and

$\varphi_1 \mathsf{V}_{<y_{\neg\psi}} \varphi_2 \in \overline{\Psi}(k+1)$ if $\overline{I}(k)$ is right-open (3b) and $\varphi_1 \mathsf{V}_{\leq y_{\neg\psi}} \varphi_2 \in \overline{\Psi}(k+1)$ if $\overline{I}(k)$ is right-closed (3c). □

8.2.15 LEMMA *Let $\bar{r} = (\overline{\Psi}, \overline{I}, \overline{\nu})$ be a timed run for $\overline{\rho}$ on A_φ and let $\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2 \in cl(\varphi)$. If $\varphi_1 \mathsf{V}_{<y_{\neg\psi}} \varphi_2 \in \overline{\Psi}(k)$ and $\overline{\nu}(k)(y_\psi) = d'$, then for all t such that $t \in \overline{I}(m)$ for some $m \geq k$ and $t < l(\overline{I}(k)) + d'$, $\varphi_2 \in \bar{r}(t)$ or there is some t' with $t' \in \overline{I}(m)$ for some $m \geq k$ and $t' < t$, such that $\varphi_1 \in \bar{r}(t)$ and $\varphi_2 \in \bar{r}(t)$.*

PROOF $\overline{\Psi}(k)$ is non-trivial for $\varphi_1 \mathsf{V}_{<y_{\neg\psi}} \varphi_2$ and from temporal consistency (4a) it follows that if for some $i \geq 0$, $\overline{\Psi}(i)$ is non-trivial for $\varphi_1 \mathsf{V}_{<y_{\neg\psi}} \varphi_2$, then $\varphi_1 \mathsf{V}_{<y_{\neg\psi}} \varphi_2 \in \overline{\Psi}(i+1)$. Local consistency (7a) says that when $\varphi_1 \mathsf{V}_{<y_{\neg\psi}} \varphi_2 \in \overline{\Psi}(i+1)$, then $\varphi_2 \in \overline{\Psi}(i+1)$ or $y_{\neg\psi} \leq 0 \in \overline{\Psi}(i+1)$. From this it follows that the first (if any) $\overline{\Psi}(m)$ that is not non-trivial for $\varphi_1 \mathsf{V}_{<y_{\neg\psi}} \varphi_2$, contains either both φ_1 and φ_2 or $y_{\neg\psi} \leq 0$. In the former case, the lemma holds, since $\varphi_2 \in \overline{\Psi}(n)$ for every $k \leq n \leq m$. In the latter case, the lemma follows from the fact that $l(\overline{I}(m)) \geq l(\overline{I}(k)) + d'$. □

8.2.16 LEMMA *Let $\bar{r} = (\overline{\Psi}, \overline{I}, \overline{\nu})$ be a timed run for $\overline{\rho}$ on A_φ and let $\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2 \in cl(\varphi)$. If $\varphi_1 \mathsf{V}_{\leq y_{\neg\psi}} \varphi_2 \in \overline{\Psi}(k)$ and $\overline{\nu}(k)(y_{\neg\psi}) = d'$, then for all t such that $t \in \overline{I}(m)$ for some $m \geq k$ and $t \leq l(\overline{I}(k)) + d'$, $\varphi_2 \in \bar{r}(t)$ or there is some t' with $t' \in \overline{I}(m)$ for some $m \geq k$ and $t' < t$ such that $\varphi_1 \in \bar{r}(t)$ and $\varphi_2 \in \bar{r}(t)$.*

PROOF $\overline{\Psi}(k)$ is non-trivial for $\varphi_1 \mathsf{V}_{\leq y_{\neg\psi}} \varphi_2$ and from temporal consistency (4b) it follows that if $\overline{\Psi}(i)$ is non-trivial for $\varphi_1 \mathsf{V}_{\leq y_{\neg\psi}} \varphi_2$ then $\varphi_1 \mathsf{V}_{\leq y_{\neg\psi}} \varphi_2 \in \overline{\Psi}(i+1)$. Local consistency (7b) says that when $\varphi_1 \mathsf{V}_{\leq y_{\neg\psi}} \varphi_2 \in \overline{\Psi}(i+1)$, then $\varphi_2 \in \overline{\Psi}(i+1)$ or $y_{\neg\psi} < 0 \in \overline{\Psi}(i+1)$. From this it follows that the first (if any) $\overline{\Psi}(m)$ that is not non-trivial for $\varphi_1 \mathsf{V}_{<y_{\neg\psi}} \varphi_2$, contains either both φ_1 and φ_2 or $y_{\neg\psi} < 0$. In the former case, the lemma holds, since $\varphi_2 \in \overline{\Psi}(n)$ for every $k \leq n \leq m$. In the latter case, the lemma follows from the fact that every $t \in \overline{I}(m)$, $t > l(\overline{I}(k)) + d'$. □

Using the previous three lemmas we can now show that Release formulas are verified correctly (see figure 8.19).

8.2.17 LEMMA *Let $\bar{r}$ be a timed run for $\overline{\rho}$ on A_φ and let $\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2 \in \bar{r}(0)$. Then for all $0 \leq t \leq d$, $\varphi_2 \in \bar{r}(t)$ or there is some $0 \leq t' < t$ such that $\varphi_1 \in \bar{r}(t')$ and $\varphi_2 \in \bar{r}(t')$.*

PROOF Let $\bar{r} = (\overline{\Psi}, \overline{I}, \overline{\nu})$. Either of the following is true:

- $\overline{\Psi}(0)$ is trivial for ψ. Then the lemma follows trivially, since $\varphi_2 \in \bar{r}(0)$ and $\varphi_1 \in \bar{r}(0)$.
- $\overline{\Psi}(0)$ is non-trivial for ψ. Then from lemma 8.2.14 if follows that the first (if any) $\overline{\Psi}(i)$ that is *not* non-trivial for ψ, is either trivial for ψ or contains $\varphi_1 \mathsf{V}_{<y_{\neg\psi}} \varphi_2$ or $\varphi_1 \mathsf{V}_{\leq y_{\neg\psi}} \varphi_2$ depending on the type of transition.
  - If $\overline{\Psi}(i)$ is non-trivial for ψ for all $i \geq 0$, then for all $t \geq 0$, $\varphi_2 \in \bar{r}(t)$.
  - If the first is trivial for ψ, then there is some $t \geq 0$ such that $\varphi_1 \in \bar{r}(t)$ and $\varphi_2 \in \bar{r}(t)$ and $\varphi_2 \in \bar{r}(t')$ for all $0 \leq t' < t$, which satisfies the lemma.
  - If the first, say $\overline{\Psi}(k)$, contains $\varphi_1 \mathsf{V}_{<y_{\neg\psi}} \varphi_2$ or $\varphi_1 \mathsf{V}_{\leq y_{\neg\psi}} \varphi_2$, then:
    - If $\varphi_1 \mathsf{V}_{<y_{\neg\psi}} \varphi_2 \in \overline{\Psi}(k)$, then $\overline{\nu}(k)(y_{\neg\psi}) = d$. From lemma 8.2.15 it follows that for all $l(\overline{I}(k)) \leq t < l(\overline{I}(k)) + d$, $\varphi_2 \in \bar{r}(t)$ or there is some $l(\overline{I}(k)) \leq t' < t$ such that $\varphi_1 \in \bar{r}(t)$ and $\varphi_2 \in \bar{r}(t)$. Further since $l(\overline{I}(k)) > 0$ ($\overline{I}(k)$ is left-closed and $k > 0$) the lemma follows.

* If $\varphi_1 \mathsf{V}_{\leq y_{\neg\psi}} \varphi_2 \in \overline{\Psi}(k)$, then $\overline{\nu}(k)(y_{\neg\psi}) = d$. From lemma 8.2.16 it follows that for all $l(\overline{I}(k)) \leq t \leq l(\overline{I}(k)) + d$, $\varphi_2 \in \bar{r}(t)$ or there is some $l(\overline{I}(k)) \leq t' < t$ such that $\varphi_1 \in \bar{r}(t)$ and $\varphi_2 \in \bar{r}(t)$ and the lemma follows. □

Having the results for Until and Release formulas, we can now show (in exactly the same manner as in the restricted case) that whenever a location is labelled with a formula ψ, this formula is verified correctly.

8.2.18 LEMMA *Let $\bar{r}$ be a run on A_φ for the timed state sequence $\overline{\rho}$ and let ψ be an MITL$_{\leq}$ formula such that $\psi \in \bar{r}(0)$. Then $\overline{\rho} \models \psi$.*

The proof of lemma 8.1.16, the corresponding lemma in the restricted case, applies to this lemma as well, with the references to lemmas 8.1.12 and 8.1.15 replaced by references to the corresponding lemmas 8.2.13 and 8.2.17 respectively.
The previous result assures us that only state sequences that satisfy the formula φ are accepted by the tableau automaton.

8.2.19 LEMMA *If A_φ accepts the timed state sequence $\overline{\rho}$ then $\overline{\rho} \models \varphi$.*

PROOF Follows from lemma 8.2.18 and the fact that there is some initial run $\bar{r}$ on A_φ and thus $\varphi \in \bar{r}(0)$. □

Completeness We now show that any timed state sequence that satisfies the formula φ is accepted by our automaton. Using the following definitions we construct an accepting run for a timed state sequence $\overline{\rho}$.

8.2.20 DEFINITION *Given an MITL$_{\leq}$ formula φ and a timed state sequence $\overline{\rho}$. Then a* φ-fine timed run $\bar{r} = \left(\overline{\Psi}, \overline{I}, \overline{\nu}\right)$ *is constructed as follows: let $\overline{I}$ be a φ-fine interval sequence for $\overline{\rho}$, let $\mu(t)$ be the φ-fine timer valuation for $\overline{\rho}$ at time t (see definition 8.1.19), let $\overline{\nu}(k) = \mu(l(\overline{I}(k)))$ and let $\overline{\Psi}(k)$ contain*

- *all $\psi \in cl(\varphi)$ that are true for all $t \in \overline{I}(k)$ and $\neg\psi$ for all $\psi \in cl(\varphi)$ that are false;*

and for every $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in cl(\varphi)$

- *$\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ and $x_\psi > 0$ if x_ψ is active during $\overline{I}(k)$ and x_ψ was activated in a left-closed way;*
- *$\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2$ and $x_\psi \geq 0$ if x_ψ is active during $\overline{I}(k)$ and x_ψ was activated in a left-open way;*
- *$\neg\left(\varphi_1 \mathsf{U}_{< y_\psi} \varphi_2\right)$ if y_ψ is active during $\overline{I}(k)$ and y_ψ was activated in a left-closed way;*
- *$\neg\left(\varphi_1 \mathsf{U}_{\leq y_\psi} \varphi_2\right)$ if y_ψ is active during $\overline{I}(k)$ and y_ψ was activated in a left-open way;*
- *$y_\psi \leq 0$ if y_ψ is not active after a left-closed active interval or no active interval has yet occurred;*
- *$y_\psi < 0$ if y_ψ is not active after a left-open active interval.* □

8.2.21 LEMMA *If $\overline{\rho} \models \varphi$ then A_φ accepts $\overline{\rho}$.*

PROOF Let $\bar{r} = (\overline{\Psi}, \overline{I}, \overline{\nu})$ be a φ-fine timed run for $\overline{\rho}$. Then $\bar{r}$ is a run for $\overline{\rho}$ on A. This follows from the following facts:

- $\overline{\Psi}$ is a sequence of locally consistent sets. This can easily be checked by the definitions of local consistency and the definition of a φ-fine timed run.
- [Symbol match] For every $t \geq 0$, $\overline{\rho}(t) \in Q(\bar{r}(t))$. This is obvious since $\bar{r}(t)$ contains precisely the propositions that are true at moment t.
- [Consecution] Let $k \geq 0$ and let every timer x be set in TS_k if x is activated at moment $l(\overline{I}(k+1))$ and let tt correspond to the transition $\overline{I}(k)$ to $\overline{I}(k+1)$. Then $\overline{\Psi}(k) \xrightarrow{TS_k, tt} \overline{\Psi}(k+1)$ and thus there is an edge $(\overline{\Psi}(k), TS_k, tt, \overline{\Psi}(k+1)) \in E$. One can show by definition of the φ-fine timer valuations that $\overline{\nu}(k+1) = TS_k[\overline{\nu}(k) - |\overline{I}(k)|]$.
- [Timing] For all $k \geq 0$, every $t \in \overline{I}(k)$ and every $\chi \in TC(\overline{\Psi}(k))$, $\overline{\nu}(k) - (t - l(\overline{I}(k))) \models \chi$. This can be verified by the semantics of the logic and the definition a φ-fine timed run.
- [Initiality] Since $\overline{\rho} \models \varphi$, $\varphi \in \overline{\Psi}(0)$ and by definition of the run $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2 \notin \overline{\Psi}(0)$. Thus $\overline{\Psi}(0) \in L_0$. □

8.2.5 Other Tableaux

Non-complete tableaux

As in the previous cases, one can leave out the completeness condition and allow also non-complete locations without changing the set of timed state sequences accepted by the automaton.

8.2.22 DEFINITION *The* non-complete real-time tableau automaton *B_φ of MITL$_\leq$ formula φ is the timed automaton $(L, \Sigma, \Theta, L_0, Q, TC, E)$ where L consists of all (also non-complete) locally consistent sets $\Phi \subseteq 2^{cl(\varphi)}$ of formulas and labels and the rest is defined exactly the same as the complete tableau automaton in definition 8.2.9.* □

Using these non-complete sets one can show that lemma 8.2.18 still holds since (as in the previous cases) completeness of the sets is never used.

8.2.23 THEOREM *Let B_φ be the non-complete tableau automaton of MITL$_\leq$ formula φ and let A_φ be the complete tableau automaton. Then $\mathcal{L}(A_\varphi) = \mathcal{L}(B_\varphi)$.*

PROOF The two inclusions are shown separately.

- $\mathcal{L}(A_\varphi) \subseteq \mathcal{L}(B_\varphi)$. It can be shown that any run on A_φ is still a run on B_φ, since $L_A \subseteq L_B$, $E_A \subseteq E_B$, $Q_A(\ell) = Q_B(\ell)$ and $TC_A(\ell) = TC_B(\ell)$ for all $\ell \in L_A$.
- $\mathcal{L}(B_\varphi) \subseteq \mathcal{L}(A_\varphi)$. Let $\overline{\rho}$ be a timed state sequence and let $\overline{\rho} \in \mathcal{L}(B_\varphi)$. Then it follows from lemma 8.2.18 that $\overline{\rho} \models \varphi$ and thus that $\overline{\rho} \in \mathcal{L}(A_\varphi)$. □

On-the-fly tableaux

We proceed towards the on-the-fly construction of the timed tableau automata. In the previous cases, the foundation for a normal form reduction procedure was found in the local consistency requirements and directly inspired by the normal form and

rewrite rules for formulas of the logic. Although a similar normal form can be defined for the unrestricted real-time case, it would become rather involved because of the differentiation transition types for both the previous and the next interval of time. Such a setting requires two sets of rewrite rules, one set for the case that the first interval is left-closed and one set for the left-open case. Similarly we need to introduce two types of Next operators, one (say $\bigcirc_{\#oc}$) that is in effect if the current interval is right-open (and the following interval thus left-closed) and one ($\bigcirc_{\#co}$) for the case that the current interval is right-closed. Then the normal form looks somewhat like this:

$$\bigvee_{i=1}^{k} TS_i.(\pi_i \wedge (\bigcirc_{\#oc}\psi_{i,1} \vee \bigcirc_{\#co}\psi_{i,2}))$$

The logical normal forms were introduced in the previous cases to enhance the insight in the reasons for constructing the normal form procedures the way they are. In this case, the normal form would become too complicated to add much to the intuitive understanding of the procedure. This exercise is therefore skipped in this section.

Constructing the Automaton In the on-the-fly tableau we use the same type of procedure as in the previous cases. This time we generate a set of triples (TS, tt, Φ) consisting of a timer setting TS, a transition type tt and a set Φ of formulas.
The transition type denotes whether the formulas in the set are to be interpreted at the beginning of a left-closed interval or at the beginning (some initial interval) of a left-open interval. Note that the two types of Next operators that would arise in a logical normal form for this case, can be avoided since the 'Next formulas' are defined implicitly by Φ.

8.2.24 DEFINITION *The* local consistency procedure *that generates a set of locally consistent sets from a timer setting TS and a set Φ of formulas is defined as follows:*

- *Let* $P_0 = \{(TS, \#oc, \Phi), (TS, \#co, \Phi)\}$.
- *Then as long as P_n contains a triple with an unmarked formula, apply one of the reductions of table 8.4.* □

A result of the procedure is denoted as $\mathrm{NF}((TS, \Phi))$. We use $\mathrm{NF}(\Phi)$ to denote $\mathrm{NF}((\varnothing, \Phi))$. An example of the application of the procedure is shown in figure 8.20 for the formula $p\mathsf{U}_{\leq 5}q$. There are separate reductions for the *#oc* and *#co* cases.
The locations (L, L_0) and edges (E) of the on-the-fly tableau automaton $(L, \Sigma, \Theta, L_0, Q, TC, E)$ are produced by the algorithm of figure 8.21. The alphabet Σ, the set Θ of timers and the labelling Q and TC are defined as in the restricted case.

Example Figure 8.22 shows an example of an on-the-fly tableau construction for the formula $p\mathsf{U}_{\leq 5}(\Diamond_{\leq 1}q)$. Note that this is the automaton as it is produced by the presented algorithm. Some straightforward optimisations along the lines of [89] and [60] would lead to a much smaller automaton. Indeed, it can be verified from figure 8.22 that all locations labelled q are equivalent, and more in general, all locations carrying the same labelling are equivalent, showing that the same could have been achieved with an automaton with 5 locations.

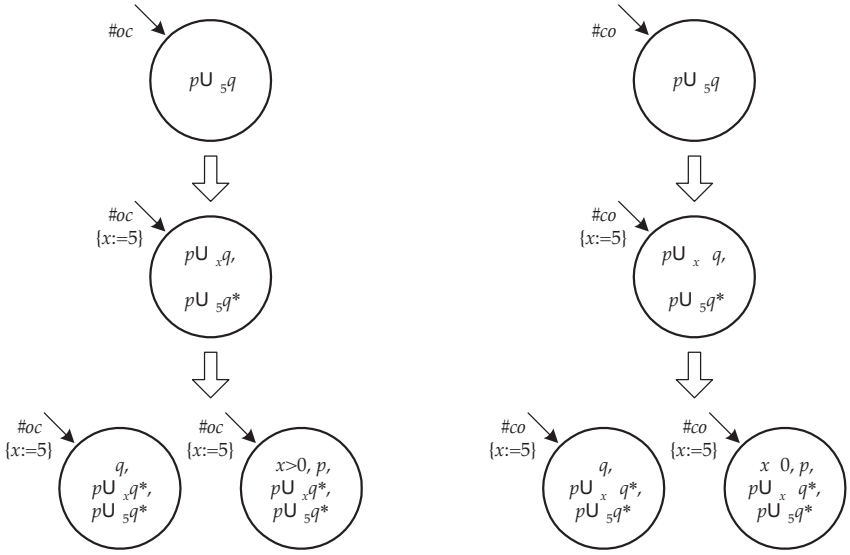


Figure 8.20: Example of the normal form reduction procedure for unrestricted intervals for the timer-setting formula pair $(\varnothing, p\mathsf{U}_{\leq 5}q)$

Correctness The correctness of the on-the-fly construction is expressed by the following theorem and is proved again by demonstrating its soundness and completeness in lemmas 8.2.27 and 8.2.31 respectively.

8.2.25 THEOREM *Let A_φ be the 'on-the-fly' tableau automaton of φ. Then $\mathcal{L}(A_\varphi) = \mathcal{L}(\varphi)$.*

Soundness We show that the automaton is sound since the procedure produces only consistent locations and edges.

8.2.26 LEMMA *Let $\Phi \subseteq cl(\varphi)$ be a set of MITL$_\leq$ formulas and labels and let $(TS', tt, \Phi') \in \mathrm{NF}(\Phi)$. Then Φ' is locally consistent. Moreover $\Phi \xrightarrow{TS', tt} \Phi'$ for any $(TS', tt, \Phi') \in \mathrm{NF}(\mathrm{Next}(\Phi))$.*

The proof of this lemma can be found in appendix B.2.4.

8.2.27 LEMMA *Let A_φ be the 'on-the-fly' tableau automaton of φ. Then $\mathcal{L}(A_\varphi) \subseteq \mathcal{L}(\varphi)$.*

PROOF Let B_φ be the non-complete tableau automaton of φ. By lemma 8.2.26, every initial run on A_φ is also an initial run on B_φ and thus every timed state sequence accepted by A_φ satisfies φ. □

	Case	$P \cup \{(TS, tt, \Psi \cup \{\psi\})\}$ reduces to:
1	$\psi = \text{false}$	P
2	$\psi = \text{true}, p, \neg p,$ $x > 0, x \geq 0,$ $y < 0, y \leq 0$	$P \cup \{(TS, tt, \Psi \cup \{\psi^*\})\}$
3	$\psi = \psi_1 \vee \psi_2$	$P \cup \{(TS, tt, \Psi \cup \{\psi^*, \psi_1\}), (TS, tt, \Psi \cup \{\psi^*, \psi_2\})\}$
4	$\psi = \psi_1 \wedge \psi_2$	$P \cup \{(TS, tt, \Psi \cup \{\psi^*, \psi_1, \psi_2\})\}$
5	$\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2$	$P \cup \{\left(\{x_\psi := d\}[TS], tt, \Psi \cup \{\psi^*, \varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2\}\right)\}$ if $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \notin \Psi$, $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2 \notin \Psi$ and $tt = \#oc$
		$P \cup \{\left(\{x_\psi := d\}[TS], tt, \Psi \cup \{\psi^*, \varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2\}\right)\}$ if $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \notin \Psi$, $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2 \notin \Psi$ and $tt = \#co$
		$P \cup \{\left(TS, tt, \Psi \cup \{\psi^*, \varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2\}\right)\}$ if $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \in \Psi$
		$P \cup \{\left(TS, tt, \Psi \cup \{\psi^*, \varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2\}\right)\}$ if $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2 \in \Psi$
6 a	$\psi = \varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$	$P \cup \{(TS, tt, \Psi \cup \{\psi^*, \varphi_2\}),$ $(TS, tt, \Psi \cup \{\psi^*, x_{\psi'} > 0, \varphi_1\})\}$
6 b	$\psi = \varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2$	$P \cup \{(TS, tt, \Psi \cup \{\psi^*, \varphi_2\}),$ $(TS, tt, \Psi \cup \{\psi^*, x_{\psi'} \geq 0, \varphi_1\})\}$
7	$\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2$	$P \cup \{(TS, tt, \Psi \cup \{\psi^*, \varphi_2\}), (TS, tt, \Psi \cup \{\psi^*, \varphi_1, \varphi_2\})\}$
8 a	$\psi = \varphi_1 \mathsf{V}_{< y_{\psi'}} \varphi_2$	$P \cup \{(TS, \Psi \cup \{\psi^*, y_{\psi'} \leq 0\}), (TS, tt, \Psi \cup \{\psi^*, \varphi_1, \varphi_2\}),$ $(TS, tt, \Psi \cup \{\psi^*, \varphi_2\})\}$
8 b	$\psi = \varphi_1 \mathsf{V}_{\leq y_{\psi'}} \varphi_2$	$P \cup \{(TS, \Psi \cup \{\psi^*, y_{\psi'} < 0\}), (TS, tt, \Psi \cup \{\psi^*, \varphi_1, \varphi_2\}),$ $(TS, tt, \Psi \cup \{\psi^*, \varphi_2\})\}$

Table 8.4: Local consistency procedure. If an unmarked formula is added to a set already containing that formula, in the new set the formula will be unmarked

```
L_0 := {(TS[0], Φ) | (TS, #oc, Φ) ∈ NF(φ, #oc)}
L := ∅, Edges := ∅
New := {Φ | (ν, Φ) ∈ L_0}
while New ≠ ∅ do
  Let Φ ∈ New
  New := New\{Φ}
  L := L∪{Φ}
  for every (TS, tt, Φ') ∈ NF(Next(Φ)) do
    Edges := Edges∪{(Φ, TS, tt, Φ')}
    if Φ' ∉ L then New := New∪{Φ'}
  od
od
```

Figure 8.21: Algorithm for constructing the locations and edges of the on-the-fly tableau automaton

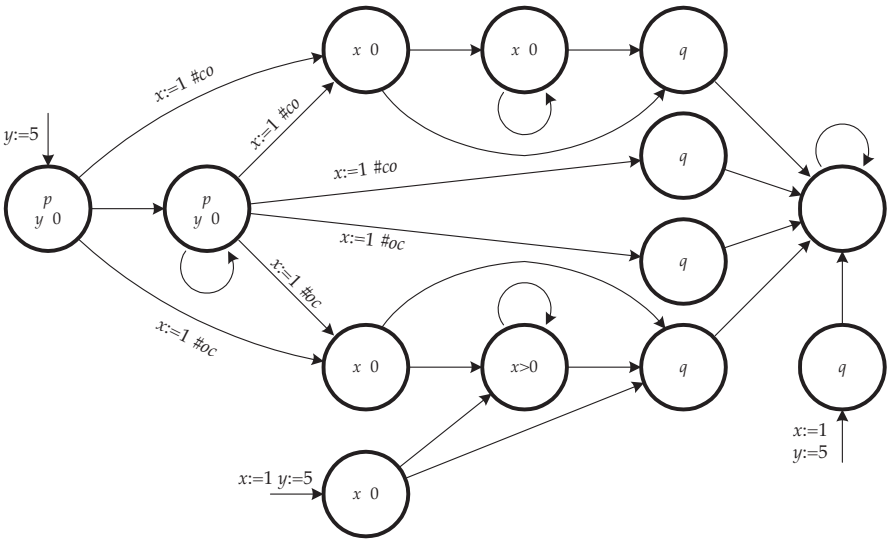


Figure 8.22: Example of the on-the-fly tableau of the formula $p\mathsf{U}_{\leq 5}(\Diamond_{\leq 1} q)$

Completeness To show completeness, a similar argument as in the restricted case is used to show that one can incrementally, from one fine interval to the next, create a run on the tableau automaton for any timed state sequence satisfying its formula. We now discriminate between left-open and left-closed intervals.
The next lemma tells us how to find a suitable initial extended location for the run. The initial locations are generated by starting the procedure from φ and transition type *#oc*. The lemma states that one of the resulting initial extended locations can form the basis for constructing the run.

8.2.28 LEMMA *Let $\Phi \subseteq cl(\varphi)$ be a set of formulas. Let $\overline{\rho} \models_\nu \Phi$ and let $\overline{I}$ be a φ-fine interval sequence. Then there is some $(TS, \#oc, \Phi') \in \mathrm{NF}(\Phi)$ such that $\overline{\rho}^t \models_{TS[\nu]-t} \Phi'$ for all $t \in \overline{I}(0)$ and $\overline{\rho}^{|\overline{I}(0)|} \models_{TS[\nu]-|\overline{I}(0)|} \mathrm{Next}(\Phi', tt)$ where tt matches the transition from $\overline{I}(0)$ to $\overline{I}(1)$.*

The proof is in appendix B.2.5.
The following two lemmas state how one can repeatedly find suitable next locations for the construction of the run. The first lemma deals with the case that the particular interval is left-closed and the second lemma deals with left-open intervals.

8.2.29 LEMMA *Let Φ be a location in the on-the-fly tableau automaton A_φ, let $\overline{\rho}$ and ν be such that $\overline{\rho} \models_\nu \mathrm{Next}(\Phi, \#oc)$ and let $\overline{I}$ be a φ-fine interval sequence. Then there is an edge $(\Phi, TS', \#oc, \Phi')$ of A_φ such that $\overline{\rho}^t \models_{TS'[\nu]-t} \Phi'$ for all $t \in \overline{I}(0)$ and $\overline{\rho}^{|\overline{I}(0)|} \models_{TS'[\nu]-|\overline{I}(0)|} \mathrm{Next}(\Phi', tt')$ where tt' matches the transition from $\overline{I}(0)$ to $\overline{I}(1)$.*

The proof is in appendix B.2.5.

8.2.30 LEMMA *Let Φ be a location in the on-the-fly tableau automaton A_φ, let $\overline{\rho}$ and ν be such that $\overline{\rho} \models_\nu \mathrm{Next}(\Phi, \#co)$ and let $\overline{I} = [0,0]I_1I_2\ldots$ be a φ-fine interval sequence. Then there is an edge $(\Phi, TS', \#co, \Phi')$ of A_φ such that $\overline{\rho}^t \models_{TS'[\nu]-t} \Phi'$ for all $t \in \overline{I}(1)$ and $\overline{\rho}^{|\overline{I}(1)|} \models_{TS'[\nu]-|\overline{I}(1)|} \mathrm{Next}(\Phi', tt')$ where tt' matches the transition from $\overline{I}(1)$ to $\overline{I}(2)$.*

The proof is in appendix B.2.5.
Finally, the next lemma claims the completeness of the on-the-fly tableau.

8.2.31 LEMMA *Let A_φ be the 'on-the-fly' tableau automaton of φ, then $\mathcal{L}(\varphi) \subseteq \mathcal{L}(A_\varphi)$.*

PROOF Let $\overline{\rho}$ be a timed state sequence such that $\overline{\rho} \models \varphi$. It can be shown using lemmas 8.2.28, 8.2.29 and 8.2.30 that there is a run for $\overline{\rho}$ on A_φ. □

8.3 Timed Automata for Prefixes

In this section we look at the analysis of finite prefixes of timed state sequences for the verification of safety properties[5] in particular during simulations. A finite timed state sequence $\overline{\tau} = (\overline{\xi}, \overline{I})$ consists of a (finite) prefix $\overline{\xi}$ of a state sequence and a prefix $\overline{I}$ (of equal length) of an interval sequence. We would like to adapt the analysis of good and bad prefixes to the setting of timed state sequences and logic. Again, local and temporal consistency requirements capture the notion of informativeness of prefixes. An information sequence is now a sequence of intervals and corresponding sets of formulas, timer valuations and timer settings. As such it describes completely why particular formulas are satisfied for certain timed prefixes.

8.3.1 DEFINITION *A timed informative sequence $\overline{IS}$ for a finite timed state sequence $\overline{\tau}$, is a tuple $(\overline{\Phi}, \overline{TS}, \overline{\nu}, \overline{I})$ of finite sequences, say of length $n+1$, such that*

- $\overline{\Phi}(n) = \varnothing$;
- *for all $0 \leq i < n$ and $\psi \in \overline{\Phi}(i)$,*
  - *if ψ is a propositional formula then $\overline{\tau}^t \models \psi$ for all $t \in \overline{I}(i)$;*
  - *if ψ is a timer condition then $\overline{\tau}^t \models_{\overline{\nu}(n)-(t-l(\overline{I}(n)))} \psi$ for all $t \in \overline{I}(i)$;*
  - *$\overline{\Phi}(i)$ is locally consistent;*
  - *$\overline{\nu}(i+1) = \overline{TS}(i+1)(\overline{\nu}(i) - |\overline{I}(i)|)$;*
  - *$\overline{\Phi}(i+1)$ is a temporally consistent successor of $\overline{\Phi}(i)$ under the timer setting $\overline{TS}(i)$ and transition type tt matching the transition from $\overline{I}(i)$ to $\overline{I}(i+1)$.*

8.3.2 DEFINITION *A prefix $\overline{\tau}$ of a timed state sequence is an informative good prefix for a formula ψ in timer environment ν if there is an informative sequence $\overline{IS}$ such that $\psi \in \overline{\Phi}(0)$ and $\overline{\nu}(0) = \nu$.* □

[5] In MITL$_{\leq}$, all expressible properties are safety properties since every Until formula is bounded.

The on-the-fly tableau automaton for real-time temporal logic as presented earlier, when interpreted on finite (prefixes of) timed state sequences (denoted as $[A_\varphi]$), can be used to detect the informative good as well as the informative bad prefixes, exactly as in the untimed case.
We now show that a timed state sequence is a bad prefix if and only if there is no run at all and that it is a good prefix if and only if there is a run to the location $\varnothing$, beginning with the latter. The following two lemmas correspond to lemma 7.4.8 in the untimed case. They state (for *#oc* and *#co* type transitions respectively) that if there is an informative sequence and an extended location of a tableau automaton such that the extended location represents a weaker constraint than the start of the informative sequence indicates, then there is an initial interval and an edge to a new location such that the extended location, after making the transition, represents a weaker constraint than the corresponding point in the informative sequence. Since the informative sequences end in an empty set of formulas, representing no constraint at all, the automaton must be able to end up in a corresponding location as well. The relation $\preceq$ in an expression $(\Phi_1, \nu_1) \preceq (\Phi_2, \nu_2)$ defines what it means for a combination (Φ_1, ν_1) to represent a weaker constraint than the combination (Φ_2, ν_2). This relation is defined in appendix B.2.6 where also the proofs of the lemmas can be found. First the lemma for the case of an *#oc* transition type.

8.3.3 LEMMA *Let Φ be a set of formulas, let ν be a timer valuation and let $\overline{IS} = (\overline{\Phi}, \overline{TS}, \overline{\nu}, \overline{I})$ be an informative sequence such that $(\Phi, \nu) \preceq (\overline{\Phi}(0), \overline{\nu}(0))$. Then there is some $(TS, \Phi', \#oc) \in \mathrm{NF}(\Phi, \#oc)$ such that $(\Phi', TS[\nu]) \preceq (\overline{\Phi}(0), \overline{\nu}(0))$ and $(\Phi'', TS''[TS[\nu] - |\overline{I}(0)|]) \preceq (\overline{\Phi}(1), \overline{\nu}(1))$ where $(TS'', \Phi'') = \mathrm{Next}(\Phi', tt)$ if tt is the type of transition from $\overline{I}(0)$ to $\overline{I}(1)$.*

The corresponding lemma for the case of a *#co* transition type.

8.3.4 LEMMA *Let Φ be a set of formulas, let ν be a timer valuation and let $\overline{IS} = (\overline{\Phi}, \overline{TS}, \overline{\nu}, \overline{I})$ be an informative sequence such that $l(\overline{I}(1)) = 0$ and $(\Phi, \nu) \preceq (\overline{\Phi}(1), \overline{\nu}(1))$. Then there is some $(TS, \Phi', \#co) \in \mathrm{NF}(\Phi, \#co)$ such that $(\Phi', TS[\nu]) \preceq (\overline{\Phi}(1), \overline{\nu}(1))$ and $(\Phi'', TS''[TS[\nu] - |\overline{I}(1)|]) \preceq (\overline{\Phi}(2), \overline{\nu}(2))$ where $(TS'', \Phi'') = \mathrm{Next}(\Phi', tt)$ if tt is the type of transition from $\overline{I}(1)$ to $\overline{I}(2)$.*

8.3.5 THEOREM *$\overline{\tau}$ is an informative good prefix of the formula φ iff there is a run for $\overline{\tau}$ on $[A_\varphi]$ ending in the location $\varnothing$.*

The good prefix theorem follows from lemmas 8.3.3 and 8.3.4.

PROOF ($\Leftarrow$) The run itself forms, with the timer settings on the edges that were followed, the informative sequence demonstrating that it is an informative good prefix. ($\Rightarrow$) Let $\overline{IS} = (\overline{\Phi}, \overline{TS}, \overline{\nu}, \overline{I})$ be the corresponding informative sequence. By the lemmas 8.3.3 and 8.3.4 and the construction of the tableau automaton, there is a run $\bar{r} = \left(\overline{\Phi}', \overline{I}, \overline{\nu}'\right)$ for $\overline{\tau}$ on $[A_\varphi]$ such that $\left(\overline{\Phi}'(n), \overline{\nu}'(n)\right) \preceq (\overline{\Phi}(n), \overline{\nu}(n))$ for all $n \geq 0$. Thus for some $k \geq 0$, $\overline{\Phi}'(k) = \overline{\Phi}(k) = \varnothing$. □

We now show that the informative bad prefixes are the ones that have no run on the automaton. The first lemma states that the normal form procedure preserves 'informative badness' (compare lemma 7.4.4 in the untimed case).

8.3.6 LEMMA *If $\overline{\tau}$ is an informative bad prefix for $\mathrm{NF}(\Phi, \#oc)$ in timer environment ν, then $\overline{\tau}$ is an informative bad prefix for Φ in timer environment ν.*

This is proved by induction on the steps of the normal form procedure (compare the proof of lemma 7.4.4). The proof is in appendix B.2.6.
The following lemma (corresponding to lemma 7.4.5) says that if there is no run for a prefix starting from a particular extended location, then this prefix is informatively bad for the formula that is represented by this extended location.

8.3.7 LEMMA *Let A_φ be a tableau automaton, let Φ be a location of A_φ and let $\overline{\tau}$ be a φ-fine prefix of a state sequence for which there is no (accepting) run on A_φ starting from extended location (Φ, ν). Then $\overline{\tau}$ is an informative bad prefix for Φ in timer environment ν or $\overline{\tau}^{|\overline{I}(0)|}$ is an informative bad prefix of Φ' in timer environment $TS[\nu - |\overline{I}(0)|]$ where $\overline{\tau} = (\overline{\xi}, \overline{I})$, $(TS, \Phi') = \mathrm{Next}(\Phi, tt)$ and tt matches the transition from $\overline{I}(0)$ to $\overline{I}(1)$.*

This lemma is also proved in appendix B.2.6. The next lemma is used for the converse; if there is an informative sequence contradicting some formula in some location, then there is no run for that prefix starting from that location (compare lemma 7.4.6).

8.3.8 LEMMA *Let $\psi \in \Phi$, let ν be a timer valuation and let $\overline{\tau}$ be an informative bad prefix for ψ in ν. Then there is no run for $\overline{\tau}$ on $[A_\varphi]$ starting from the extended location (Φ, ν).*

This is proved in appendix B.2.6 by induction on ψ and the length of the prefix.
Finally, we have the theorem about informative bad prefixes.

8.3.9 THEOREM *There is a run for $\overline{\tau}$ on $[A_\varphi]$ iff $\overline{\tau}$ is not an informative bad prefix of the formula φ.*

The theorem follows from lemmas 8.3.6 and 8.3.8.

PROOF ($\Rightarrow$) Assume towards a contradiction that $\overline{\tau}$ is an informative bad prefix for φ. Any initial run starts from a location Φ such that $\varphi \in \Phi$. But by lemma 8.3.8, such a run cannot exist. ($\Leftarrow$) Again, by contradiction, assume that $\overline{\tau}$ is not accepted by $[A_\varphi]$. Then by lemmas 8.3.6 and 8.3.7, $\overline{\tau}$ is an informative bad prefix for φ. □

Now that we have automata that discriminate the informative good and bad prefixes, we need to see if we can effectively determine if a prefix is accepted or rejected by the timed automaton.

8.4 Deterministic Timed Automata

The on-the-fly tableau construction presented in section 8.2.5 enables the effective model-checking with linear-time temporal logic formulas in a dense time domain. This is achieved by the classic approach of constructing the synchronous product of the system automaton and the tableau automaton of the negated formula and performing an emptiness check on the resulting automaton, as explained in chapter

6. Emptiness can be decided by the famous algorithm of [7]. The clever use of an equivalence relation (region equivalence) on the infinite set of extended locations allows these locations to be grouped into a finite number of equivalence classes. Such an equivalence relation exists since both the formula and the system automaton only refer to integer (or rational) time bounds.
If we would like to employ the same tableau construction for verifying individual (prefixes of) timed state sequences that result for instance from a system simulation, then the region equivalence does not help. As explained in chapter 6, for this application we need the observer automaton to be deterministic (or at least a deterministic procedure to analyse the acceptance of a state sequence), and a determinisation of timed automata is impossible in general.
It is for this reason that we now investigate the use of a fragment of the logic, that we can employ to produce deterministic timed observer automata. The fragment is the timed equivalent of the deterministic fragment discussed in the previous chapter.

8.4.1 DEFINITION *The* syntax of deterministic MITL$_{\leq}$ *is defined by the following grammar*

$$\varphi ::= \pi \mid \pi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \mathsf{U}_{\leq d} \pi \mid \pi \mathsf{V}_{\leq d} \varphi$$

where $\pi \in$ PL *and* $d \in \mathbb{N}$. □

For this logic and the appropriate normal form procedure (the rules of which are depicted in table 8.5), every disjunctive term arising in the normal form contains mutually exclusive propositional formulas and timer constraints.
In table 8.5, only rules 2, 5 a / b, 6 and 7a / b have been adapted in a way similar to the untimed case. Whenever sets are split, the choice between the sets is deterministic. The choice between outgoing edges of a location is entirely deterministic.

Example As an example, the deterministic tableaux automaton for the formula $\Box_{\leq 100} \Diamond_{\leq 5} p$ is shown in figure 8.23.

The Linear/Quadratic Fragment Another way to circumvent the problem of non-determinisability, is to adopt a linear fragment. A subset construction does not work in the general case, since one has to remember sets of extended locations. If, however, we restrict ourselves to a fragment of the logic that yields tableau automata the locations of which are non-trivial with respect to at most one Until or Release formula, then there is at most one timer 'active' in any particular location. Moreover, the timer behaves monotonically; if a timed state sequence is accepted from a particular location with a particular timer value, then it is also accepted from that same location for any larger timer value in case of Until formulas and for any smaller timer value in the case of Release formulas. In this situation it is possible to select, from a set of extended locations, a single representative extended location by the observation that $\varphi_1 \mathsf{U}_{\leq x} \varphi_2 \vee \varphi_1 \mathsf{U}_{\leq y} \varphi_2 \equiv \varphi_1 \mathsf{U}_{\leq x} \varphi_2$ if $\nu(x) \geq \nu(y)$. Now a sort of subset-like analysis is possible for a determinisation. The syntax for this fragment is the following.

8.4.2 DEFINITION *The* syntax of linear MITL$_{\leq}$ *is defined by the following grammar*

$$\varphi ::= \pi \mid \varphi \vee \varphi \mid \pi \wedge \varphi \mid \pi \mathsf{U}_{\leq d} \varphi \mid \varphi \mathsf{V}_{\leq d} \pi$$

where $\pi \in$ PL *and* $d \in \mathbb{N}$. □

	Case	$P \cup \{(TS, tt, \Psi \cup \{\psi\})\}$ reduces to:
1	$\psi = \pi, x > 0,$ $x \geq 0, y < 0,$ $y \leq 0$	$P \cup \{(TS, tt, \Psi \cup \{\psi^*\})\}$
2	$\psi = \pi \vee \psi'$	$P \cup \{(TS, tt, \Psi \cup \{\psi^*, \pi\}), (TS, tt, \Psi \cup \{\psi^*, \neg\pi, \psi'\})\}$
3	$\psi = \psi_1 \wedge \psi_2$	$P \cup \{(TS, tt, \Psi \cup \{\psi^*, \psi_1, \psi_2\})\}$
4	$\psi = \varphi \mathsf{U}_{\leq d} \pi$	$P \cup \{\left(\{x_\psi := d\}[TS], tt, \Psi \cup \{\psi^*, \varphi \mathsf{U}_{\leq x_\psi} \pi\}\right)\}$ if $\varphi \mathsf{U}_{\leq x_\psi} \pi \notin \Psi$, $\varphi \mathsf{U}_{\leq x_\psi + \epsilon} \pi \notin \Psi$ and $tt = \#oc$
		$P \cup \{\left(\{x_\psi := d\}[TS], tt, \Psi \cup \{\psi^*, \varphi \mathsf{U}_{\leq x_\psi + \epsilon} \pi\}\right)\}$ if $\varphi \mathsf{U}_{\leq x_\psi} \pi \notin \Psi$, $\varphi \mathsf{U}_{\leq x_\psi + \epsilon} \pi \notin \Psi$ and $tt = \#co$
		$P \cup \{\left(TS, tt, \Psi \cup \{\psi^*, \varphi \mathsf{U}_{\leq x_\psi} \pi\}\right)\}$ if $\varphi \mathsf{U}_{\leq x_\psi} \pi \in \Psi$
		$P \cup \{\left(TS, tt, \Psi \cup \{\psi^*, \varphi \mathsf{U}_{\leq x_\psi + \epsilon} \pi\}\right)\}$ if $\varphi \mathsf{U}_{\leq x_\psi + \epsilon} \pi \in \Psi$
5 a	$\psi = \varphi \mathsf{U}_{\leq x_{\psi'}} \pi$	$P \cup \{(TS, tt, \Psi \cup \{\psi^*, \pi\}),$ $(TS, tt, \Psi \cup \{\psi^*, x_{\psi'} > 0, \neg\pi, \varphi\})\}$
5 b	$\psi = \varphi \mathsf{U}_{\leq x_{\psi'} + \epsilon} \pi$	$P \cup \{(TS, tt, \Psi \cup \{\psi^*, \pi\}),$ $(TS, tt, \Psi \cup \{\psi^*, x_{\psi'} \geq 0, \neg\pi, \varphi\})\}$
6	$\psi = \pi \mathsf{V}_{\leq d} \varphi$	$P \cup \{(TS, tt, \Psi \cup \{\psi^*, \neg\pi, \varphi\}), (TS, tt, \Psi \cup \{\psi^*, \pi, \varphi\})\}$
7 a	$\psi = \pi \mathsf{V}_{< y_{\psi'}} \varphi$	$P \cup \{(TS, \Psi \cup \{\psi^*, y_{\psi'} \leq 0\}),$ $(TS, tt, \Psi \cup \{\psi^*, y_{\psi'} > 0, \pi, \varphi\}),$ $(TS, tt, \Psi \cup \{\psi^*, y_{\psi'} > 0, \neg\pi, \varphi\})\}$
7 b	$\psi = \pi \mathsf{V}_{\leq y_{\psi'}} \varphi$	$P \cup \{(TS, \Psi \cup \{\psi^*, y_{\psi'} < 0\}),$ $(TS, tt, \Psi \cup \{\psi^*, y_{\psi'} \geq 0, \pi, \varphi\}),$ $(TS, tt, \Psi \cup \{\psi^*, y_{\psi'} \geq 0, \neg\pi, \varphi\})\}$

Table 8.5: Local consistency procedure for the deterministic fragment. If an unmarked formula is added to a set already containing that formula, in the new set the formula will be unmarked

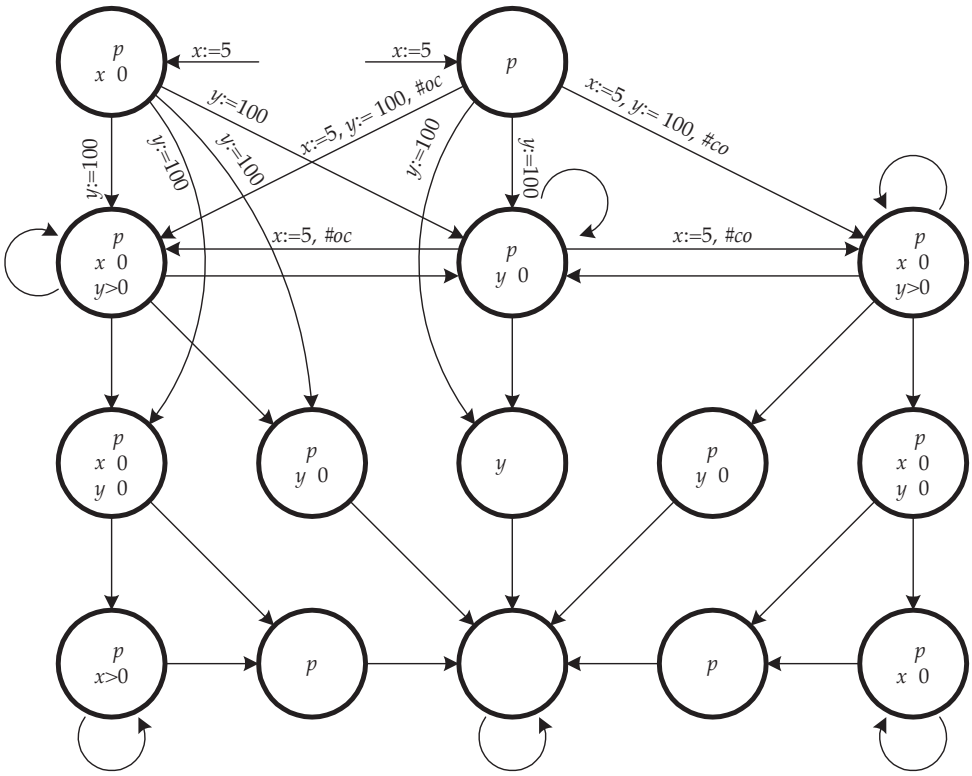


Figure 8.23: Deterministic tableau automaton of the formula $\Box_{\leq 100}\Diamond_{\leq 5}$

Being the dual of the deterministic fragment, the negation of every linear formula is in fact a deterministic formula. Since the on-the-fly construction for deterministic formulas can detect both good and bad prefixes, the same construction can be applied to the (deterministic) negation of the formula. An informative good prefix for a linear $\text{MITL}_{\leq}$ formula φ is an informative bad prefix for $\neg\varphi$ and an informative bad prefix for φ is an informative good prefix for $\neg\varphi$.
As in the untimed case, for this fragment every location resulting from the normal form procedure is non-trivial for at most one Until or Release formula. Therefore, we can use the method sketched earlier to analyse the acceptance of a timed state sequence (for instance under simulations). We do not elaborate on this and assume that they are analysed using the deterministic tableau construction.

8.5 Related Work

Real-temporal logics

Timed temporal logics exist in many variations. Some as extensions of LTL, some adopt the branching time model of behaviour and some are derived from interval temporal logic. They may be interpreted on a discrete time domain or on a dense time domain. Several ways have been put forward to quantify time in formulas, such as adding bounds to existing operators (MTL [120], MITL [9]) of the untimed logic, or the introduction of clocks and constraints on those clocks (TPTL [13], L_ν [126], state-clock logic [158]).
The complexity of timed temporal logics varies with the adopted operators and time domain. Many combinations in fact lead to undecidable logics. In a dense time domain for instance, the ability to express punctuality constraints such as 'every event *p* is followed after *exactly* 1 unit of time by an event *q*' leads to undecidability [13]. MITL is a decidable variant that restricts timing constraints on temporal operators to be of the shape of an interval bounded by integer numbers. Its complexity is EXPSPACE, whereas a further restriction as used in this chapter is PSPACE [9].
On the branching time side of the spectrum there are logics such as TCTL [6, 102], an extension of CTL with a temporally constrained Until operator, timed modal logic L_ν [126] with explicit timers and a timed version of the modal μ-calculus [106]. Real-time extensions of other logics also exist, such as interval temporal logic [157].
Overviews and discussions of real-time models, real-time temporal logics and their complexities (a topic left aside in this thesis) can be found in [12, 11, 9, 101].

Tableaux for real-time temporal logic

Tableau constructions are important components of automated analysis of real-time systems and requirements. Whereas tableau constructions for untimed modal logic have been developed to a high degree of efficiency, tableaux for linear timed modal logics do not yet exist or are intended to establish the theoretical connection between logic and automata rather than to be used for practical automated analyses.
A tableau construction for TPTL interpreted in a discrete time domain is found in [13]. Here the timing constraints are unfolded into locations of the automaton, a strategy that does not apply in the dense time case.

The construction of so-called testing automata from formulas in a safety modal logic for the UPPAAL tool is described in [3] and [2]. This logic is a restricted variant of the logic in [126], with restrictions similar to the deterministic fragment described in this chapter. Although the logic is a branching time logic, the restrictions allow the model-checking to be performed as a reachability analysis of the synchronous composition of the system with the test automaton.
The only tableau construction (that the author knows of) for dense linear time temporal logic is the tableau for MITL in [5, 9]. As it uses operators constrained with intervals with an integer lower and upper bound, the complexity of the construction is high (EXPSPACE). Moreover, the presented tableau is not intended for practical model-checking, but rather to establish the theoretical connection between the logic and the theory of timed automata. We have seen in this chapter that the construction does not deal correctly with formulas of the form $\varphi_1 \mathsf{U}_{(0,d)} \varphi_2$.
The on-the-fly tableau construction for the restricted type of interval sequences discussed in this chapter was presented in [83]. A decision procedure for real-time interval logic is given in [157]. It employs a similar restriction on intervals (both on interval constraints and intervals of models) to enforce a uniform treatment of the right continuous interval $[t, \infty)$ (and possibly to avoid problems with a different transition type as discussed in this chapter).
Branching time temporal logic model-checking is performed on the region graph (e.g. TCTL, enumeratively in [6], symbolically in [102] and symbolically on-the-fly in [36]). Timing constraints are interpreted directly and symbolically on this graph. This is based on the fact that two extended locations that are in the same region satisfy the same TCTL formulas.

Tools for the verification of real-time systems

A number of tools exist for automata theoretic analysis of (dense) real-time and hybrid systems. The UPPAAL tool [127, 128] performs reachability analysis on timed and hybrid systems. It can perform model-checking for a restricted real-time temporal logic. It has been successfully applied to verify a number of telecommunication and consumer electronics protocols. The Kronos tool [63] allows for TCTL model-checking using a symbolic verification algorithm [102].
Extensions have also been made to the Spin verification tool and model-checker. Extensions to Spin such as [173, 35], analyse discrete or dense timed systems. These extensions do not support timed temporal logic model-checking. HyTech [10] is another tool for the verification of (parametric) timed and hybrid systems.

8.6 Conclusions and Future Work

In this chapter we have presented a tableau construction for a real-time temporal logic. The construction is presented in two steps, first a restricted version is discussed introducing the use of timers to check quantitative timing constraints. Then the unrestricted version was discussed, adding the necessary bookkeeping to deal with the shape of interval transitions.
The tableau construction forms the basis of formal analysis tools that deal with real-time systems described as timed automata and real-time properties expressed in the

logic, such as satisfiability analysis and model-checking. It is also the basis for the construction of observers to monitor quantitative timing properties during simulations.
We introduced fragments of the logic that lead (similar to the untimed fragments of the previous chapter) to deterministic tableaux and tableaux where along any path at any given time, at most one Until or Release operator is active.
Future work includes the investigation of weakly monotonic timed state sequences. The interleaving semantics of the calculus in this thesis does in fact give rise to timed state sequences that are not strictly monotonic. Processes of the calculus can perform a sequence of instantaneous actions at a certain point in time. A similar interpretation is adopted for the timed automata in the UPPAAL tool [2]. This implies that atomic proposition may change value a number of times at that point in time. This requires a somewhat adapted interpretation of the logic and of the notion of a run on the automaton. The types of transitions would include the possibility of a transition from a right-closed to a left-closed interval of time. It remains to be investigated how such a model fits in the described framework.
Although we have a prototype implementation, a real implementation still has to be made that can be integrated with the existing analysis tools for simulation and verification. In such an implementation, one also needs to consider optimisations of the construction along the lines of [89] and [60] in order to obtain automata that are small enough for effective analysis, especially since the analysis of timed automata is much more involved in practice than the analysis of ordinary Büchi automata.
Furthermore, one could think about the use of similar techniques on extensions versions of $\text{MITL}_{\leq}$ or other logics such as interval logics, which also have an attractive graphical representation that could facilitate their use.

Chapter 9

Conclusion

Increasing complexity in real-time distributed systems calls for techniques to automate and support their design. In particular concurrent and communicating systems are hard to design correctly. This is especially important for embedded and safety-critical systems. Formal techniques are useful for the construction of tools that support the design process and verification of a design. This thesis has introduced or extended formal techniques that assist in the assessment of the correct operation of a system's design, in particular in its earliest stages of the design trajectory, when executable specifications are made and analysed. Errors detected early in the design process can be eliminated without excessive costs.

System-level modelling languages are applied to create executable specifications. These specifications model the system at a high level of abstraction and can be used to explore the design space. If a design involves certain communication protocols or interaction patterns, it is necessary to test the correct operation of the interactions. Unexpected sequences of events may lead to deadlock situations or otherwise unwanted forms of behaviour. Although traditional formal verification methods are designed to attack these types of problems, they possess some characteristics that clash with the dynamic nature of the initial design process:

- They tend to be quite involved and not suitable for quick assessment of a design.

- They assume that both the design and the specified requirements are fixed. This is not true in a design-space exploration phase. The conceptual solution has not yet been fixed and the requirements list is still incomplete and changing. Both the design and the detailed list of requirements evolve concurrently during this phase. Since typical current formal methods require a considerable effort in terms of time, expertise and resources, they cannot easily deal with quickly changing designs and specifications.

- The fact that the application of formal verification methods is laborious and aims at finding the 'final' few errors, it should not be applied to an immature design that is likely to contain many errors that can be discovered by studying the specification or by executing it in a simulation.

In practice in the early stages of the design, a popular tool to study designs is simulation. Too often such simulations are done in languages that do not have suitable primitives to model aspects like concurrency or real-time, or do not support appropriate structuring concepts. Many languages do not possess a well defined (formally defined) semantics. Simulation is very effective for a quick assessment of a design, since it is flexible, easy to use, interactive and gives the designer the opportunity to develop some feeling for the system's behaviour.
As a design evolves, the need for more thorough analyses emerges. The design and its requirements are more stable and in this phase the investments required for the application of (more) exhaustive formal verification techniques can be justified. The currently available tools largely lack the support for large-scale application, such as support for management of designs and specifications and support for push-button regressive verification after small changes have been made to a design. In principle, however, these methods can be applied. To smoothen the design process, it is necessary to allow a formal design specification and formal requirements to evolve together, from initial specifications to the refined specifications they become later and to allow formal verification of the design to evolve simultaneously from non-exhaustive simulation techniques to exhaustive analyses of the more mature design.

9.1 Contributions of this thesis

This thesis contributes a number of formal techniques that are effective to study the behaviour and in particular the correct operation of distributed real-time systems.

- In this thesis, the calculus of chapter 3 is an abstract semantical model on which executable specification languages can be built. An example of such a specification / modelling language is POOSL. Such languages can be characterised as concurrent, real-time, expressing structure and architecture as well as behaviour. A more operational description of terminating behaviour is described towards implementing it.

  In chapter 4, an execution technique for such executable specification languages for concurrent communicating real-time systems is introduced, that is built on the basis of the formal semantics. This technique has been applied to the formal specification language POOSL, to obtain the SHESim tool. This tool supports interactive modelling and simulation of POOSL models and the graphical specification and visualisation of the system's architecture.

- The semantics of the calculus is extended to expose internal behaviour in chapter 5. In simulations one typically employs closed systems that model both the actual system under design and the behaviour of its environment. In this case, the semantics in terms of externally observable behaviour alone is inadequate. Moreover, we introduce a semantics that focusses in particular on the static structure of a system and that allows one to establish the connection between the behaviour of the entire system and the contributing behaviour of individual components.

- A formalism to specify properties of such statically structured systems is also introduced in chapter 5, by attributing temporal logic properties to individual

components. This approach is in particular suitable to support the concept of encapsulation and component-based approaches.

- Chapter 6 discusses the advantages and disadvantages of different approaches to formal verification and in particular of random state-space exploration (simulation) as opposed to traditional exhaustive verification techniques. It has been shown that different techniques serve different needs in different stages of the design. Non-exhaustive techniques apply to bigger systems and to changing specifications and requirements. Exhaustive techniques, on the other hand, are superior when it comes to finding the subtle errors that depend on very particular combinations of events. Besides that, the application of such methods forces the user to think more closely about the design and the reasons why it must be correct (an activity which by itself may lead to the discovery of problems in the design).

- In chapter 7 the construction is discussed of finite-state automata that can monitor correctness requirements specified in linear temporal logic. A general framework is introduced to describe the construction of such automata, in which different approaches can be captured uniformly. It is shown that particular fragments of temporal logic can be identified for which deterministic automata can be produced. It is also shown how tableau automata can be used to incrementally monitor finite executions produced by simulations for temporal logic properties.

- Chapter 8 extends the tableau method to a real-time temporal logic. In this logic, it is possible to express quantitative temporal requirements as well as qualitative ones. It is shown that for a particular class of formulas, efficient automata can be constructed (on-the-fly) that can be used as monitors for real-time model-checking in practice. It is shown that this extension is applicable to exhaustive verification methods for real-time systems as well as to simulations. A prototype implementation of the constructions has been implemented and some experimental results are presented.

9.2 Future Work

Obviously, the work presented in this thesis does not solve all of the problems we are currently facing with verification. Indeed, the fact that formal verification methods have been around for a while, but are not yet fully embraced by industrial designers suggests that there is a lot more work to be done on the theoretical side, but equally important, on the practical side as well. In this section we mention but a few aspects of the work that remains to be done.

- To apply automatic verification methods, exhaustive or non-exhaustive, both the system and its requirements have to be formalised. Many requirements are generated in the initial design. Specifications in natural language, interaction diagrams, use-case diagrams, activity diagrams, object-class diagrams, timing diagrams, and so forth, informally state requirements of the system. It remains to be investigated how such information is effectively transformed into formal

specifications and what kind of formalisms capture them best. Timing diagrams, for instance, can be expressed in linear temporal logic.

- The extension of the automata theoretic approach to timed properties remains to be implemented and the effectiveness of the considered fragments of the logic in practical verification needs to be assessed. A prototype implementation of the construction has been made, but needs to be integrated with verification and simulation tools for timed systems.

- Practical aspects of tool support require more attention. Managing catalogs of models and requirements should be automated. Support is required for tracking dependencies between models and requirements and re-verification after changes have been made. Interoperability between tools is also important. One would like for instance, to be able to replay counterexamples found by a verification tool in one's popular simulation environment.

- More practical case studies are required to investigate the strong and weak points of the different methodologies, for example to test the error-finding capabilities of the different approaches and to find guidelines for assessing when to invest time in detailed verification efforts and for what subsystems.

Appendix A

POOSL Description of the APS Protocol

This appendix shows the (simple) POOSL model of the APS protocol[1]. Since the subject of data objects is not touched upon in this thesis, the specification of the data classes `Request`, `RequestType` and `SignalCondition` have been omitted. They can be thought of as objects à la Smalltalk, Java or C++ and their behaviour should be easy to grasp from the context. This appendix merely serves as an indication of what the POOSL model and language look like. It is beyond the scope of this thesis to go into the exact syntax and semantics of POOSL. The interested reader is directed to [156, 86, 85]. The POOSL description is divided in the description of the process classes and the description of the cluster classes, presented in the following sections.

A.1 Process Classes

This section contains the process class definitions. Process classes capture the dynamic behaviour of process objects, the POOSL counterparts of the dynamic processes described in section 3.1.1 of this thesis. The process classes that together make up an APS node are: `LocalProcess`, `ExternalProcess`, `APSProcess` and `GlobalProcess`, and `NetworkManager` and `Environment` model the surroundings interacting with the protocol.

```
process class LocalProcess(NumberOfSignals: Integer)

instance variables localRequest: Request; lockedout: Array; signalConditions: Array

communication channels lop, sgnCond, locReq

message interface
    locReq!changeLocalRequest(1);
    sgnCond?changeCondition(2);
    lop?lockout(1);
    lop?clearlockout(1)

initial method call init()()
```

[1] A more detailed model is described in [132].

```
instance methods

updateLocalRequest()()
    | newLocalRequest: RequestType |
    computeLocalRequest()(newLocalRequest);
    if (newLocalRequest=localRequest) not then
        localRequest := newLocalRequest;
        locReq!changeLocalRequest(localRequest);
    fi.

normalOperation()()
    | n: Integer; condition: SignalCondition |
    sel
        sgnCond?changeCondition(n,condition);
        signalConditions put(n, condition);
    or
        lop?lockout(n);
        lockedout put(n, true);
    or
        lop?clearlockout(n);
        lockedout put(n, false);
    les;
    updateLocalRequest()();
    normalOperation()().

init()()
    | i: Integer |
    signalConditions := new(Array) size(NumberOfSignals);
    i:=1;
    while i<=NumberOfSignals do
        signalConditions put(i,new(SignalCondition) setToNormal);
        i:=i+1;
        od;
    lockedout := new(Array) size(NumberOfSignals) putAll(false);
    computeLocalRequest()(localRequest);
    locReq!changeLocalRequest(localRequest);
    normalOperation()().

computeLocalRequest()(resultRequest: Request)
    | i: Integer |
    resultRequest := new(Request) setRequestTypeToNR setSignalNumber(0);
    i:=1;
    while i<=NumberOfSignals do
        if (signalConditions get(i) requestType higherThan(resultRequest requestType)) &
                (lockedout get(i) not) then
            resultRequest setRequestType( signalConditions get(i) requestType);
            resultRequest setSignalNumber(i);
        fi;
        i:=i+1;
    od.
```

```
process class ExternalProcess()

instance variables activeCommand: String; externalRequest: Request;
                   activeSignal: Integer

communication channels lop, extReq, extComm

message interface
    lop!lockout(1);
    extReq!changeExternalRequest(1);
    extComm?command(2);
    lop!clearlockout(1)

initial method call init()()
```

```
instance methods

computeExternalRequest()(newExternalRequest: Request)
    if activeCommand = "forcedswitch" then
        newExternalRequest := new(Request) setRequestTypeToFSw
                                setSignalNumber(activeSignal)
    else
    if activeCommand="clear" then
        newExternalRequest := new(Request) setRequestTypeToNR setSignalNumber(0)
    else
        delay nil error("unknown command")
    fi
    fi.

processExternalCommand(comm: String; signal: Integer)()
    if comm="lockout" then
        lop!lockout(signal);
    else
    if comm="clearlockout" then
        lop!clearlockout(signal)
    else
        activeCommand := comm;
        activeSignal := signal;
    fi
    fi;
    updateExternalRequest()().

normalOperation()()
    | comm: String; signal: Integer; newExternalRequest: Request |
    extComm?command(comm, signal);
    processExternalCommand(comm, signal)();
    normalOperation()().

updateExternalRequest()()
    | newExternalRequest: Request |
    computeExternalRequest()(newExternalRequest);
    if (newExternalRequest=externalRequest) not then
        externalRequest := newExternalRequest;
        extReq!changeExternalRequest(externalRequest);
    fi.

init()()
    activeCommand := "clear";
    activeSignal := 0;
    normalOperation()().
```

```
process class APSProcess(Side: String)

instance variables globalRequest: RequestType; remoteRequest: RequestType

communication channels sndAPS, glbReq, recAPS, rmtReq

message interface
    sndAPS!request(1);
    recAPS?request(1);
    rmtReq!changeRemoteRequest(1);
    glbReq?changeGlobalRequest(1)

initial method call init()()

instance methods

sendRemoteRequest()()
    sndAPS!request(globalRequest);
    delay 1;
    readRemoteRequest()().
```

```
readGlobalRequest()()
    | newGlobalRequest: RequestType |
    glbReq?changeGlobalRequest(newGlobalRequest);
    if (newGlobalRequest=globalRequest) not then
        globalRequest := newGlobalRequest;
    fi;
    readGlobalRequest()().

readRemoteRequest()()
    | newRemoteRequest: RequestType |
    recAPS?request(newRemoteRequest);
    if (newRemoteRequest=remoteRequest) not then
        remoteRequest := newRemoteRequest;
        rmtReq!changeRemoteRequest(remoteRequest);
    fi;
    sendRemoteRequest()().

init()()
    par
        readGlobalRequest()()
    and
        if Side="left" then
            sendRemoteRequest()()
        else
            readRemoteRequest()()
        fi
    rap.
```

```
process class GlobalProcess()

instance variables localRequest: Request; globalRequest: Request;
                   remoteRequest: Request; externalRequest: Request

communication channels extReq, glbReq, locReq, rmtReq

message interface
    locReq?changeLocalRequest(1);
    extReq?changeExternalRequest(1);
    rmtReq?changeRemoteRequest(1);
    glbReq!changeGlobalRequest(1)

initial method call init()()

instance methods

computeGlobalRequest()(resultRequest: RequestType)
    | highestRequest: Request |
    highestRequest := localRequest max(remoteRequest) max(externalRequest);
    if highestRequest==remoteRequest then
        resultRequest := new(Request) setRequestTypeToRR
                                      setSignalNumber(remoteRequest signalNumber)
    else
        resultRequest := highestRequest
    fi.

normalOperation()()
    | newGlobalRequest: RequestType |
    /* Wait for a new request */
    sel
        locReq?changeLocalRequest(localRequest);
    or
        extReq?changeExternalRequest(externalRequest);
    or
        rmtReq?changeRemoteRequest(remoteRequest);
    les;
    computeGlobalRequest()(newGlobalRequest);
    if (newGlobalRequest=globalRequest) not then
        globalRequest := newGlobalRequest;
        glbReq!changeGlobalRequest(globalRequest);
```

```
    fi;
    normalOperation()().

init()()
    localRequest := new(Request) setRequestTypeToNR setSignalNumber(0);
    remoteRequest := new(Request) setRequestTypeToNR setSignalNumber(0);
    externalRequest := new(Request) setRequestTypeToNR setSignalNumber(0);
    normalOperation()().
```

```
process class NetworkManager()

instance variables

communication channels extComm1, extComm2

message interface
    extComm1!command(2);
    extComm2!command(2)

initial method call init()()

instance methods

scenario1()()
    delay 7.8;
    extComm1!command("lockout",1);
    extComm2!command("lockout",1).

scenario2()()
    delay 50;
    extComm2!command("forcedswitch",2);
    delay 50;
    extComm1!command("forcedswitch",1);
    delay 50;
    extComm1!command("clear",0);
    extComm2!command("clear",0).

init()()
    scenario1()().
```

```
process class Environment()

instance variables random: RandomGenerator

communication channels sgnCond1, sgnCond2

message interface
    sgnCond1!changeCondition(2);
    sgnCond2!changeCondition(2)

initial method call init()()

instance methods

changeSignals()()
    | n: Integer; cond: Integer; sc: SignalCondition |
    n := (random random() *2) floor() +1;
    cond := (random random() *20) floor() +1;
    sc := new(SignalCondition);
    if cond<18 then
        sc setToNormal
    else if cond=18 then
            sc setToDegraded
        else sc setToFail
        fi
    fi;
    sgnCond1!changeCondition(n,sc);
    delay random random * 10 ;
    n := (random random() *2) floor() +1;
```

```
    cond := (random random() *20) floor() +1;
    sc := new(SignalCondition);
    if cond<18 then
        sc setToNormal
    else if cond=18 then
            sc setToDegraded
        else sc setToFail
        fi
    fi;
    sgnCond2!changeCondition(n,sc);
    delay random random * 10 ;
    changeSignals()().

scenario1()()
    | sc1, sc2: SignalCondition |
    sc1 := new(SignalCondition);
      sc1 setToDegraded;
    sc2 := new(SignalCondition);
      sc2 setToFail;
    delay 5.5;
    sgnCond1!changeCondition(1,sc1);
    sgnCond2!changeCondition(1,sc2).

init()()
    scenario1()().
```

A.2 Cluster Classes

POOSL's cluster classes capture structure and hierarchy of the model and are the counterparts of the static processes (see section 3.1.2). This model contains two cluster classes, first the class APSNode, built from process objects of the classes LocalProcess, ExternalProcess, APSProcess and GlobalProcess, and the class APSProtocol, built from two APSNode's, one NetworkManager and one Environment, as visualised in figures 5.4 and 5.5.

```
cluster class APSNode(NumberOfSignals: Integer; Side: String)

communication channels recAPS, sndAPS, sgnCond, extComm

message interface
    sndAPS ! request(1);
    recAPS ? request(1);
    sgnCond ? changeCondition(2);
    extComm ? command(2)

behaviour specification
(GlobalProcess: GlobalProcess || LocalProcess: LocalProcess(NumberOfSignals) ||
 ExternalProcess: ExternalProcess || APSMessages: APSProcess(Side)
)\{rmtReq, locReq, lop, glbReq, extReq}
```

```
cluster class APSProtocol()

communication channels

message interface

behaviour specification
(NetworkManagement: NetworkManager || Environment: Environment ||
 APSNode1: APSNode(2, "left")
   [extComm1/extComm, aps1/sndAPS, aps2/recAPS, sgnCond1/sgnCond] ||
 APSNode2: APSNode(2, "right")
   [extComm2/extComm, aps2/sndAPS, aps1/recAPS, sgnCond2/sgnCond]
)\{extComm1, extComm2, aps1, aps2, sgnCond1, sgnCond2}
```

Appendix B

Proofs

This appendix contains the proofs of lemmas in chapters 7 and 8

B.1 Proofs of chapter 7

This section contains the proofs omitted from chapter 7.

B.1.1 Proof of lemma 7.2.7

To complete the soundness proofs, we need an alternative definition of local consistency. In the normal form procedure, a set of locally consistent sets is produced. During this construction the sets will be partially locally consistent, more precisely, they will be locally consistent w.r.t. those formulas that have been processed.

B.1.1 DEFINITION *A set Φ of formulas is called* locally consistent *w.r.t. the set L of formulas if for every $\psi \in L$*

1. *if $\psi = \text{false}$, then $\psi \notin \Phi$;*
2. *if ψ is of the form $\psi_1 \vee \psi_2$ and $\psi \in \Phi$, then $\psi_1 \in \Phi$ or $\psi_2 \in \Phi$;*
3. *if ψ is of the form $\psi_1 \wedge \psi_2$ and $\psi \in \Phi$, then $\psi_1 \in \Phi$ and $\psi_2 \in \Phi$;*
4. *if ψ is of the form $\psi_1 \mathsf{U} \psi_2$ and $\psi \in \Phi$, then $\psi_1 \in \Phi$ or $\psi_2 \in \Phi$;*
5. *if ψ is of the form $\psi_1 \mathsf{V} \psi_2$ and $\psi \in \Phi$, then $\psi_2 \in \Phi$.* □

To prove lemma 7.2.7, we first prove the following lemma.

B.1.2 LEMMA *Let P_n be a set of sets of formulas generated during the procedure described in definition 7.2.5. Then every set $\Psi \in P_n$ is locally consistent w.r.t. its marked formulas.*

PROOF One can prove that the rules for generating P_{n+1} from P_n preserve this property. Since the procedure is started from P_0 containing no marked formulas and end with only marked formulas, the lemma follows. Note that none of the rules ever removes a formula from a set. Once a set is consistent w.r.t. certain formulas it will remain consistent w.r.t. those formulas or the set is removed completely. The proof follows by induction on n

- no $\Psi \in P_0$ has marked formulas;
- let every $\Psi \in P_n$ be consistent w.r.t. all marked formulas. Then it can be shown that every $\Psi' \in P_{n+1}$ is also consistent w.r.t. all marked formulas. It is shown for every case separately
  - case 1, trivial since the set containing false is removed;
  - case 2, 3, 4, the formula is marked and does not impose any new requirements;
  - case 5, both new sets $\Psi' \cup \{\psi^*, \psi_1\}$ and $\Psi' \cup \{\psi^*, \psi_2\}$ are locally consistent with respect to the newly marked formula ψ^*;
  - case 6, the new set $\Psi' \cup \{\psi^*, \psi_1, \psi_2\}$ is locally consistent with respect to the newly marked formula ψ^*;
  - case 7, it is easy to see that the new sets satisfy also local consistency constraint 4 which applies to the newly marked formula;
  - case 8, similarly it can be shown that constraint 5 is satisfied in the new sets. □

Now lemma 7.2.7 follows.

7.2.7 LEMMA *Let Φ be a set of formulas and let $\Phi' \in \mathrm{NF}(\Phi)$. Then Φ' is locally consistent. Moreover, $\Phi \rightarrow \Phi'$ for any $\Phi' \in \mathrm{NF}(\mathrm{Next}(\Phi))$*

PROOF At the end of the procedure, the formulas in Φ' are all marked. From this it follows easily using lemma B.1.2 that Φ' is locally consistent. Furthermore, for any set of formulas Φ, $\Phi \rightarrow \mathrm{Next}(\Phi)$ and those formulas are never removed during the procedure, for every $\Phi' \in \mathrm{NF}(\Phi)$, $\mathrm{Next}(\Phi) \subseteq \Phi'$ and thus $\Phi \rightarrow \Phi'$. □

B.1.2 Proof of lemma 7.2.9

For the proof of lemma 7.2.9, regarding the completeness of the on-the-fly construction, we need some more definitions. We need to consider temporal consistency constraints with respect to marked formulas (as opposed to consistency constraints with respect to all formulas in the set, given by the function Next). Such a function gives the constraints only w.r.t. those formulas that have already been processed by the procedure.

B.1.3 DEFINITION *Let Ψ be a set of formulas, some of which are marked. Then the set* $\mathrm{MNext}(\Psi)$ *of* marked temporal consistency constraints *is the smallest set such that if the Until or Release formula $\psi \in \Psi$ is marked and Ψ is non-trivial for ψ then* $\mathrm{MNext}(\Psi)$ *contains ψ.* □

Thus, at the end of the procedure, if all formulas of Ψ are marked, then $\mathrm{MNext}(\Psi) = \mathrm{Next}(\Psi)$. The following property is used to demonstrate completeness of the on-the-fly tableau construction. It is formulated as an invariant of the local consistency procedure.

B.1.4 DEFINITION *Let the set P contain sets $\Psi \subseteq cl(\varphi)$ for some* LTL *formula φ. Let $\overline{\sigma}$ be a state sequence. The predicate* $\mathrm{Inv}(P, \overline{\sigma})$ *holds iff there is some $\Psi \in P$ such that*

(i) for all $\psi \in \Psi$, $\overline{\sigma} \models \psi$;

(ii) $\overline{\sigma}^1 \models \psi$ for all $\psi \in \mathrm{MNext}(\Psi)$;

(iii) for every marked Until formula $\psi_1 \mathsf{U} \psi_2$ in Ψ such that $\overline{\sigma} \models \psi_2$, we have $\psi_2 \in \Psi$. □

The next lemma shows that $\mathrm{Inv}(P_n, \overline{\sigma})$ is invariant during the construction of $\mathrm{NF}(\Phi)$.

B.1.5 LEMMA *Let P_n contain sets $\Psi \subseteq cl(\varphi)$ for some* LTL *formula φ and let $\overline{\sigma}$ be a state sequence. Assume* $\mathrm{Inv}(P_n, \overline{\sigma})$ *and let P_{n+1} be obtained from P_n by applying one of the above rules, then* $\mathrm{Inv}(P_{n+1}, \overline{\sigma})$.

PROOF If the $\mathrm{Inv}(P_n, \overline{\sigma})$ holds for some $\Psi \in P_n$ different from the one on which the case applies then the lemma holds trivially since $\Psi \in P_{n+1}$. If it holds for the $\Psi \in P_n$ on which the case applies, then $\mathrm{Inv}(P_{n+1}, \overline{\sigma})$ can be proved for every case separately

- case 1, cannot occur since $\overline{\sigma} \not\models$ false;
- case 2, 3, 4 are trivial since $P_{n+1} = P_n$ (except for the markings, but no new consistency constraints are introduced);
- case 5, from $\overline{\sigma} \models \psi$ it follows that $\overline{\sigma} \models \psi_1$ or $\overline{\sigma} \models \psi_2$. In the first case, $\Psi' \cup \{\psi^*, \psi_1\}$ satisfies the requirements and in the latter case $\Psi' \cup \{\psi^*, \psi_2\}$ does (no new temporal consistency constraints are introduced);
- case 6, from $\overline{\sigma} \models \psi$ it follows that $\overline{\sigma} \models \psi_1$ and $\overline{\sigma} \models \psi_2$, and thus $\Psi' \cup \{\psi^*, \psi_1, \psi_2\}$ satisfies the requirements;
- case 7, from $\overline{\sigma} \models \psi_1 \mathsf{U} \psi_2$ it follows that $\overline{\sigma} \models \psi_1$ or $\overline{\sigma} \models \psi_2$. If $\overline{\sigma} \models \psi_2$ then $\Psi' \cup \{\psi^*, \psi_2\}$ satisfies the requirements, if not then $\Psi' \cup \{\psi^*, \psi_1\}$ does. If $\overline{\sigma} \not\models \psi_2$ then $\overline{\sigma}^1 \models \psi$.
- case 8, from $\overline{\sigma} \models \psi_1 \mathsf{V} \psi_2$ it follows that $\overline{\sigma} \models \psi_2$. If $\overline{\sigma} \models \psi_1$ then $\Psi' \cup \{\psi^*, \psi_1, \psi_2\}$ satisfies the requirements, if not then $\Psi' \cup \{\psi^*, \psi_2\}$ does. If $\overline{\sigma} \not\models \psi_1$ then $\overline{\sigma}^1 \models \psi$. □

From the invariant, the desired lemma follows.

7.2.9 LEMMA *Let Φ be a set of formulas. Let $\overline{\sigma} \models \psi$ for all $\psi \in \Phi$. Then there is some $\Phi' \in \mathrm{NF}(\Phi)$ such that (i) $\overline{\sigma} \models \psi$ for all $\psi \in \Phi'$, (ii) $\overline{\sigma}^1 \models \mathrm{Next}(\Phi')$ and (iii) for every Until formula $\psi_1 \mathsf{U} \psi_2$ in Φ such that $\overline{\sigma} \models \psi_2$, we have $\psi_2 \in \Phi$.*

PROOF Follows immediately from lemma B.1.5. Initially, nothing is marked and (ii) and (iii) of the invariant follow trivially. At the end of the procedure the invariant still holds and coincides with the conclusion of the lemma. □

B.1.3 Proof of lemma 7.4.4

Lemma 7.4.4 states that informativeness of prefixes is preserved by the normal form construction. To prove it we need to define when a prefix is considered to be informative for the artifacts used during the normal form procedure.

B.1.6 DEFINITION *A finite state sequence $\overline{\xi}$ is an informative bad prefix for a (partially) marked set Ψ of formulas if there is some $\psi \in \Psi$ such that $\overline{\xi}$ is an informative bad prefix for ψ or there is some $\psi \in \mathrm{MNext}(\Psi)$ (*MNext *as defined in section B.1.2) such that $\overline{\xi}^1$ is an informative bad prefix for ψ.*

A set of such sets, corresponds to the disjunction of the formulas associated with these sets. To disprove their disjunction, a prefix has to be informative for every term in the disjunction.

B.1.7 DEFINITION *A finite state sequence $\overline{\xi}$ is an informative bad prefix for a set P of marked sets of formulas if $\overline{\xi}$ is an informative bad prefix for every $\Psi \in P$.*

Two informative sequences can be combined into a single new one, simply by taking the union of the corresponding sets. If $\overline{IS}_1$ and $\overline{IS}_2$ are both informative sequences, then $(\overline{IS}_1 \cup \overline{IS}_2)(k) = \overline{IS}_1(k) \cup \overline{IS}_2(k)$ for all $k \geq 0$ (taking $\overline{IS}(k) = \varnothing$ if $k > |\overline{IS}|$). It is easy to see that if $\overline{IS}_1$ and $\overline{IS}_2$ are informative sequences for $\overline{\xi}$, then $\overline{IS}_1 \cup \overline{IS}_2$ is an informative sequence for $\overline{\xi}$ as well.
The next lemma shows that reductions in the normal form procedure preserve informativeness of bad prefixes.

B.1.8 LEMMA *Let prefix $\overline{\xi}$ be an informative bad prefix for P' and let $P \Rightarrow P'$ in the normal form procedure. Then $\overline{\xi}$ is an informative bad prefix for P.*

PROOF One can prove this for the reduction cases individually, which is a tedious case analysis. We only show cases 5 and 7.

- Case 5, $P = P'' \cup \{\Psi \cup \{\psi_1 \vee \psi_2\}\}$ and $P' = P'' \cup \{\Psi \cup \{\psi_1 \vee \psi_2^*, \psi_1\}, \Psi \cup \{\psi_1 \vee \psi_2^*, \psi_2\}\}$. If $\overline{\xi}$ is an informative bad prefix of P', it is a bad prefix of both $\Psi \cup \{\psi_1 \vee \psi_2^*, \psi_1\}$ and $\Psi \cup \{\psi_1 \vee \psi_2^*, \psi_2\}$. If $\overline{IS}$ is an informative sequence demonstrating this (both), then $\overline{IS} \cup \{\neg(\psi_1 \vee \psi_2)\}$ is an informative sequence for $\Psi \cup \{\psi_1 \vee \psi_2\}$. From this it follows straightforwardly that $\overline{\xi}$ is an informative bad prefix for P (note that marking $\psi_1 \vee \psi_2$ does not add any consistency constraints).
- Case 7 is a little more intricate. It corresponds to the rewriting of $\psi_1 \mathsf{U} \psi_2 \equiv \psi_2 \vee (\psi_1 \wedge \bigcirc(\psi_1 \mathsf{U} \psi_2))$. The case analysis below essentially amounts to saying that either the bad prefix disproves other formulas than $\psi_1 \mathsf{U} \psi$ or it disproves $\psi_1 \mathsf{U} \psi_2$ by $\neg\psi_2$ and either $\neg\psi_1$ or $\neg \bigcirc (\psi_1 \mathsf{U} \psi_2)$.
  In this case we have $P = P'' \cup \{\Psi \cup \{\psi_1 \mathsf{U} \psi_2\}\}$ and $P' = P'' \cup \{\Psi \cup \{\psi_1 \mathsf{U} \psi_2^*, \psi_1\}, \{\Psi \cup \{\psi_1 \mathsf{U} \psi_2^*, \psi_2\}\}$. Let $\overline{\xi}$ be an informative bad prefix for P' and let $\overline{IS}$ be an informative sequence demonstrating this.
  - If there is some $\psi \in \Psi$ such that $\neg\psi \in \overline{IS}(0)$ or if there is some $\psi \in \mathrm{MNext}(\Psi)$ such that $\neg\psi \in \overline{IS}(1)$ then $\overline{IS}$ is an informative sequence for P.
  - Otherwise,
    * If $\neg(\psi_1 \mathsf{U} \psi_2) \in \overline{IS}(0)$ then $\overline{IS}$ is an informative sequence for P.
    * Otherwise, $\neg\psi_2 \in \overline{IS}(0)$ and either $\neg\psi_1 \in \overline{IS}(0)$ or $\neg(\psi_1 \mathsf{U} \psi_2) \in \overline{IS}(1)$. In either case, $\overline{IS} \cup \{\neg(\psi_1 \mathsf{U} \psi_2)\}$ is an informative sequence demonstrating that $\overline{\xi}$ is an informative bad prefix of P. □

From this it follows immediately that the entire normal form procedure preserves informativeness of bad prefixes.

7.4.4 LEMMA *If $\overline{\xi}$ is an informative bad prefix for* $\mathrm{NF}(\Phi)$*, then $\overline{\xi}$ is an informative bad prefix for Φ.*

B.1.4 Proof of lemma 7.4.6

This lemma says that an informative bad prefix cannot have a run on the on-the-fly tableau automaton.

7.4.6 LEMMA *Let $\psi \in \Phi$ and let $\overline{IS}$ be an informative sequence demonstrating $\neg\psi$ for $\overline{\xi}$. Then there is no run for $\overline{\xi}$ on $[A_\varphi]$ starting from Φ.*

PROOF By induction on the length of $\overline{\xi}$ and the structure of ψ.

- $\psi = p$, since $\overline{IS}$ is an informative sequence for $\neg p$, $p \notin \overline{\xi}(0)$. But $p \in \Phi$ and thus there is no run on $[A_\varphi]$ for $\overline{\xi}$ starting from Φ.
- $\psi = \psi_1 \vee \psi_2$, then both $\neg\psi_1$ and $\neg\psi_2$ are in $\overline{IS}(0)$ while $\psi_1 \in \Phi$ or $\psi_2 \in \Phi$. Thus by induction there is no run on $[A_\varphi]$ for $\overline{\xi}$ starting from Φ.
- $\psi = \psi_1 \wedge \psi_2$ then $\neg\psi_1$ or $\neg\psi_2$ is in $\overline{IS}(0)$ while both $\psi_1 \in \Phi$ and $\psi_2 \in \Phi$. Thus by induction there is no run on $[A_\varphi]$ for $\overline{\xi}$ starting from Φ.
- $\psi = \psi_1 \mathsf{U} \psi_2$, then either $\psi_2 \in \Phi$ or $\psi_1 \in \Phi$ and $\psi \in \overline{\ell}(1)$ for any appropriate run $\overline{\ell}$. Since $\neg(\psi_1 \mathsf{U} \psi_2) \in \overline{IS}(0)$, $\neg\psi_2 \in \overline{IS}(0)$ and $\neg\psi_1 \in \overline{IS}(0)$ or $\neg\psi \in \overline{IS}(1)$. That such a run $\overline{\ell}$ cannot exist follows by induction. Notice that the latter case can only occur if $|\overline{\xi}| > 1$ since $\overline{IS}(|\overline{\xi}|) = \varnothing$, i.e. $\neg\psi$ cannot be postponed forever.
- $\psi = \psi_1 \mathsf{V} \psi_2$ then $\psi_2 \in \Phi$ and $\psi_1 \in \Phi$ or $\psi \in \overline{\ell}(1)$ for any appropriate run $\overline{\ell}$. Since $\neg(\psi_1 \mathsf{V} \psi_2) \in \overline{IS}(0)$, $\neg\psi_2 \in \overline{IS}(0)$ or $\neg\psi_1 \in \overline{IS}(0)$ and $\neg\psi \in \overline{IS}(1)$. That such a run $\overline{\ell}$ does not exist follows by induction again. □

B.1.5 Proof of lemma 7.4.8

This lemma suggests how informative sequences can be used to construct a run to the empty location. The lemma is proved using an invariant on the normal form procedure, introduced in the next definition.

B.1.9 DEFINITION *In the following lemmas, the predicate $Inv(P, \overline{IS})$ holds iff there is some $\Psi \in P$ such that $\Psi \subseteq \overline{IS}(0)$ and* $\mathrm{MNext}(\Psi) \subseteq \overline{IS}(1)$ *(*MNext *as defined in section B.1.2).*

$Inv(P, \overline{IS})$ states that $\overline{IS}$ is informative for at least one of (the formulas associated with) the sets in P and thus for (the formula associated with) the set itself. We show that $Inv(P, \overline{IS})$ is invariant under reductions in the normal form procedure.

B.1.10 LEMMA *Let $P \Rightarrow P'$, let $\overline{IS}$ be an informative sequence and assume $Inv(P, \overline{IS})$. Then $Inv(P', \overline{IS})$.*

PROOF By case analysis of the procedure. We only show cases 6 and 8.

- Case 6, $P = P'' \cup \{\Psi \cup \{\psi_1 \wedge \psi_2\}\}$ and $P' = P'' \cup \{\Psi \cup \{\psi_1 \wedge \psi_2^*, \psi_1, \psi_2\}\}$. If there is some $\Psi' \in P''$ such that $\Psi' \subseteq \overline{IS}(0)$ and $\mathrm{MNext}(\Psi') \subseteq \overline{IS}(1)$ then the result is trivial. Otherwise, the set satisfying the property is $\Psi \cup \{\psi_1 \wedge \psi_2\}$. Then $\psi_1 \wedge \psi_2 \in \overline{IS}(0)$ and by local consistency, $\psi_1 \in \overline{IS}(0)$ and $\psi_2 \in \overline{IS}(0)$. Then $\Psi \cup \{\psi_1 \wedge \psi_2^*, \psi_1, \psi_2\} \subseteq \overline{IS}(0)$ and $\mathrm{MNext}(\Psi \cup \{\psi_1 \wedge \psi_2^*, \psi_1, \psi_2\}) \subseteq \mathrm{MNext}(\Psi \cup \{\psi_1 \wedge \psi_2\}) \subseteq \overline{IS}(1)$.

- Case 8, $P = P'' \cup \{\Psi \cup \{\psi_1 \mathsf{V} \psi_2\}\}$ and $P' = P'' \cup \{\Psi \cup \{\psi_1 \mathsf{V} \psi_2^*, \psi_1, \psi_2\}, \Psi \cup \{\psi_1 \mathsf{V} \psi_2^*, \psi_2\}\}$. If there is some $\Psi' \in P''$ such that $\Psi' \subseteq \overline{IS}(0)$ and $\mathrm{MNext}(\Psi') \subseteq \overline{IS}(1)$ then the result is trivial. Otherwise, the set satisfying the property is $\Psi \cup \{\psi_1 \mathsf{V} \psi_2\}$. Then $\psi_1 \mathsf{V} \psi_2 \in \overline{IS}(0)$ and by local consistency $\psi_2 \in \overline{IS}(0)$.
  - If $\psi_1 \in \overline{IS}(0)$ then $\Psi \cup \{\psi_1 \mathsf{V} \psi_2^*, \psi_1, \psi_2\} \subseteq \overline{IS}(0)$ and $\mathrm{MNext}(\Psi \cup \{\psi_1 \mathsf{V} \psi_2^*, \psi_1, \psi_2\}) \subseteq \mathrm{MNext}(\Psi \cup \{\psi_1 \mathsf{V} \psi_2\}) \subseteq \overline{IS}(1)$.
  - If $\psi_1 \notin \overline{IS}(0)$ then $\Psi \cup \{\psi_1 \mathsf{V} \psi_2^*, \psi_2\} \subseteq \overline{IS}(0)$ and $\mathrm{MNext}(\Psi \cup \{\psi_1 \mathsf{V} \psi_2^*, \psi_2\}) \subseteq \mathrm{MNext}(\Psi \cup \{\psi_1 \mathsf{V} \psi_2\}) \cup \{\psi_1 \mathsf{V} \psi_2\} \subseteq \overline{IS}(1)$ since $\psi_1 \mathsf{V} \psi_2 \in \overline{IS}(1)$ by temporal consistency. □

From the previous lemma it follows directly that the following holds for the entire normal form procedure.

7.4.8 LEMMA *Let Φ be a set of formulas and let $\overline{IS}$ be an informative sequence with $\Phi \subseteq \overline{IS}(0)$. Then there is some $\Phi' \in \mathrm{NF}(\Phi)$ such that $\Phi' \subseteq \overline{IS}(0)$ and $\mathrm{Next}(\Phi') \subseteq \overline{IS}(1)$.*

B.2 Proofs of chapter 8

This section contains the proofs left out of chapter 8.

B.2.1 Proof of lemma 8.1.26

Here, again, we need a modified definition of local consistency to deal with the intermediate stages during the normal form construction.

B.2.1 DEFINITION *A set Φ is called* locally consistent *w.r.t. the set L of formulas and labels, if for every $\psi \in L$*

1. *if ψ is* false*, then $\psi \notin \Phi$;*
2. *if ψ is of the form $\psi_1 \vee \psi_2$ and $\psi \in \Phi$, then $\psi_1 \in \Phi$ or $\psi_2 \in \Phi$;*
3. *if ψ is of the form $\psi_1 \wedge \psi_2$ and $\psi \in \Phi$, then $\psi_1 \in \Phi$ and $\psi_2 \in \Phi$;*
4. *if ψ is of the form $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ and $\psi \in \Phi$, then $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \in \Phi$;*
5. *if ψ is of the form $\varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$ and $\psi \in \Phi$, then $\varphi_2 \in \Phi$ or $x_{\psi'} > 0 \in \Phi$ and $\varphi_1 \in \Phi$;*
6. *if ψ is of the form $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$ and $\psi \in \Phi$, then $\varphi_2 \in \Phi$;*
7. *if ψ is of the form $\varphi_1 \mathsf{V}_{< y_{\psi'}} \varphi_2$ and $\psi \in \Phi$, then $y_{\psi'} \leq 0 \in \Phi$ or $\varphi_2 \in \Phi$.* □

B.2.2 LEMMA *Let P_n be a set of locations generated during the normal form procedure. Then for every pair (TS, Ψ), $\Psi \in P_n$ is locally consistent w.r.t. its marked formulas.*

PROOF It is easy to verify that the rules for generating P_{n+1} from P_n preserve this property. The rules closely follow the consistency requirements. Since the procedure is started from P_0 containing no marked formulas, the lemma follows. □

B.2.3 COROLLARY *Let* $\mathrm{NF}((TS, \Phi))$ *be a set of pairs generated during the normal form procedure and let* $(TS', \Phi') \in \mathrm{NF}((TS, \Phi))$. *Then* Φ' *is locally consistent.*

PROOF Let $\psi \in cl(\varphi)$. Then either $\psi \in \Psi$ in which case Ψ is consistent w.r.t. ψ (by lemma B.2.2) or $\psi \notin \Psi$ in which case Ψ is also consistent w.r.t. ψ since all consistency constraints are of the form "if $\psi \in \Psi$ then ..." (except the false rule, but false $\notin \Psi$). □

B.2.4 LEMMA *Let* $\Psi \xrightarrow{TS'} \Psi'$ *for all* $(TS', \Psi') \in P_n$ *and let* P_{n+1} *be obtained from* P_n *by a reduction of the procedure. Then* $\Psi \xrightarrow{TS'} \Psi'$ *for all* $(TS', \Psi') \in P_{n+1}$.

PROOF Proved by the rules individually. Let (TS', Ψ') be the pair on which the reduction is applied.

- Rules 1 through 4 and 6 through 8 do not affect the temporal consistency constraints.
- Rule 5. If $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \notin \Psi'$ then Ψ cannot be non-trivial for $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ thus the addition of $x_\psi := d$ does not violate temporal consistency condition 1. Moreover, the addition guarantees that condition 2 is satisfied. If on the other hand $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \in \Psi'$ then nothing changes. □

Lemma 8.1.26 follows directly from corollary B.2.3 and lemma B.2.4.

8.1.26 LEMMA *Let* $\Phi \subseteq cl(\varphi)$ *be a set of formulas and let* $\Phi' \in \mathrm{NF}(\Phi)$. *Then* Φ' *is locally consistent. Moreover,* $\Phi \xrightarrow{TS'} \Phi'$ *for any* $(TS', \Phi') \in \mathrm{NF}(\mathrm{Next}(\Phi))$.

B.2.2 Proof of lemma 8.1.28

This lemma lies at the heart of the construction of a run for a timed state sequence that satisfies the formula of the tableau automaton. To prove it, we need the following extra definitions. During the normal form procedure, if only part of the formulas have been marked, we want to look at the consistency constraints associated with those marked formulas only.

B.2.5 DEFINITION *Let* Ψ *be a set of formulas, some of which may be marked. Then* $\mathrm{MNext}(\Psi)$ *is the pair* (TS, Ψ') *where TS and* Ψ' *are the smallest sets such that*

- *if* $\psi = \varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ *is marked and* Ψ *is non-trivial for* ψ, *then* Ψ' *contains* ψ;
- *if* $\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2$, ψ *is marked and* Ψ *is non-trivial for* ψ, *then* Ψ' *contains* $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2$ *and TS contains* $y_\psi := d$;
- *if* $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2$ *is marked and* Ψ *is non-trivial for* $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2$, *then* Ψ' *contains* $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2$.

□

Once all formulas are marked, $\mathrm{Next}(\Psi) = \mathrm{MNext}(\Psi)$. Similar to the untimed case we can establish an invariant during the procedure that tells us that if a timed state sequence satisfies the initial formulas, then there will be a term in the normal form that it satisfies, and thus there will be an appropriate edge to be taken in the automaton.

B.2.6 DEFINITION *Let P contain pairs (TS, Ψ), $\Psi \subseteq cl(\varphi)$ for some MITL$_{\leq}$ formula φ. Let $\overline{\rho}$ be a timed state sequence, and let ν be a timer valuation such that for every timer x in the domain of ν, corresponding to the Until formula $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ or the Release formula $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$, $\nu(x) \leq d$. The predicate* $\mathrm{Inv}(P, \overline{\rho}, \nu)$ *holds iff for any φ-fine interval sequence $\overline{I}$, there is some $(TS, \Psi) \in P$ such that*

(i) for all $t \in \overline{I}(0)$, $\overline{\rho}^t \models_{TS[\nu]-t} \Psi$;

(ii) $\overline{\rho}^{|\overline{I}(0)|} \models_{TS[\nu]-|\overline{I}(0)|} \mathrm{MNext}(\Psi)$. □

We now show that this predicate is an invariant during the normal form procedure.

B.2.7 LEMMA *Let P_n contain pairs (TS, Ψ) where $\Psi \subseteq cl(\varphi)$ for some MITL$_{\leq}$ formula φ. Let $\overline{\rho}$ be a timed state sequence, let ν be a timer valuation, assume that* $\mathrm{Inv}(P_n, \overline{\rho}, \nu)$ *and let P_{n+1} be obtained from P_n by applying one of the above rules. Then* $\mathrm{Inv}(P_{n+1}, \overline{\rho}, \nu)$.

PROOF If there is such a set other than the one on which the case applies, then the lemma follows trivially. If not, it is shown case by case that the invariant is preserved. Let $(TS, \Psi) \in P_n$ and let $\psi \in \Psi$ on which the case is applied.

- Cases 1,2,3 and 4 are trivial or similar to the untimed case and consistency constraints are not affected.
- Case 5.
  - If $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \notin \Psi$, then $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ is added and $x_\psi := d$ is added to TS. Because x_ψ does not occur in any formula $\psi' \in \Psi$, it follows from $\overline{\rho} \models_{TS[\nu]} \psi'$ that $\overline{\rho}^t \models_{(\{x_\psi := d\}[TS])[\nu]-t} \psi'$ for all $t \in \overline{I}(0)$. Furthermore, $\overline{\rho}^t \models_{(\{x_\psi := d\}[TS])[\nu]-t} \psi$ for all $t \in \overline{I}(0)$. (Either $\overline{\rho}^t \models_{(\{x_\psi := d\}[TS])[\nu]-t} \varphi_2$ for all $t \in \overline{I}(0)$ or $|\overline{I}(0)| \leq d$.) No temporal consistency constraints are added.
  - If $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \in \Psi$, then it is trivial since nothing is added.
- Case 6. From $\overline{\rho} \models_{TS[\nu]} \psi$ it follows that $\overline{\rho} \models_{TS[\nu]} \varphi_2$ or $\nu(x_{\psi'}) > 0$ and $\overline{\rho} \models_{TS[\nu]} \varphi_1$. In the former case, the first pair satisfies the requirement and no new consistency constraints are introduced. If $\overline{\rho} \not\models_{TS[\nu]} \varphi_2$, then the second pair satisfies the requirement. Since the interval sequence is φ_2-fine, the first interval cannot be longer than $\nu(x_{\psi'})$ and thus satisfies $x_{\psi'} > 0$ for all $t \in \overline{I}(0)$. The consistency constraint $\varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$ is added, but it is satisfied by $\overline{\rho}$: $\overline{\rho}^{|\overline{I}(0)|} \models_{TS[\nu]-|\overline{I}(0)|} TS'.\varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$ (TS' does not modify $x_{\psi'}$).
- Case 7. Straightforward. If $\overline{\rho} \models_{TS[\nu]} \varphi_1$ then the second pair satisfies the invariant. Otherwise the first does and the new consistency constraint is satisfied by $\overline{\rho}$.
- Case 8. Similar. □

Lemma 8.1.28 now follows directly from the previous lemma.

8.1.28 LEMMA *Let φ be an MITL$_<$ formula and let $\Phi \subseteq cl(\varphi)$ be a set of extended formulas. Let $\overline{\rho}$ be a timed state sequence such that $\overline{\rho} \models_\nu \Phi$. Then for any φ-fine interval sequence $\overline{I}$, there is some $(TS, \Phi') \in \mathrm{NF}(\Phi)$ such that $\overline{\rho}^t \models_{TS[\nu]-t} \Phi'$ for all $t \in \overline{I}(0)$ and $\overline{\rho}^{|\overline{I}(0)|} \models_{TS[\nu]-|\overline{I}(0)|} \mathrm{Next}(\Phi')$.*

B.2.3 Proofs for the Unrestricted Case

For the proofs of the on-the-fly tableau for unrestricted timed state sequences, we use the following alternative definition of local consistency.

B.2.8 DEFINITION *A set Φ is called* locally consistent *w.r.t. a set L of formulas and labels if for every $\psi \in L$*

1. *if ψ is* false*, then $\psi \notin \Phi$;*
2. *if ψ is of the form $\psi_1 \vee \psi_2$ and $\psi \in \Phi$, then $\psi_1 \in \Phi$ or $\psi_2 \in \Phi$;*
3. *if ψ is of the form $\psi_1 \wedge \psi_2$ and $\psi \in \Phi$, then $\psi_1 \in \Phi$ and $\psi_2 \in \Phi$;*
4. *if ψ is of the form $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ and $\psi \in \Phi$, then either $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \in \Phi$ or $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2 \in \Phi$ (not both);*
5. (a) *if ψ is of the form $\varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$ and $\psi \in \Phi$, then $\varphi_2 \in \Phi$ or both $x_{\psi'} > 0 \in \Phi$ and $\varphi_1 \in \Phi$;*
   (b) *if ψ is of the form $\varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2$ and $\psi \in \Phi$, then $\varphi_2 \in \Phi$ or both $x_{\psi'} \geq 0 \in \Phi$ and $\varphi_1 \in \Phi$;*
6. *if ψ is of the form $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$ and $\psi \in \Phi$, then $\varphi_2 \in \Phi$;*
7. (a) *if ψ is of the form $\varphi_1 \mathsf{V}_{< y_{\psi'}} \varphi_2$ and $\psi \in \Phi$, then $y_{\psi'} \leq 0 \in \Phi$ or $\varphi_2 \in \Phi$;*
   (b) *if ψ is of the form $\varphi_1 \mathsf{V}_{\leq y_{\psi'}} \varphi_2$ and $\psi \in \Phi$, then $y_{\psi'} < 0 \in \Phi$ or $\varphi_2 \in \Phi$.* □

B.2.4 Proof of lemma 8.2.26

This lemma states the soundness of the tableau construction. It is shown, by an appropriate invariant on the normal form procedure, that the generated locations are consistent.

B.2.9 LEMMA *Let P_n be a set of locations generated during the normal form procedure. For every $(TS, tt, \Psi) \in P_n$, location Ψ is locally consistent w.r.t. its marked formulas.*

PROOF It is easy to verify that the rules for generating P_{n+1} from P_n preserve this property. The rules closely follow the consistency requirements. Since the procedure is started from P_0 containing no marked formulas, the lemma follows. □

B.2.10 COROLLARY *Let $\mathrm{NF}((TS, \Phi))$ be a set of triples generated during the normal form procedure and let $(TS', tt, \Phi') \in \mathrm{NF}((TS, \Phi))$. Then Φ' is locally consistent.*

The next lemma deals with the second part of lemma 8.2.26, with temporal consistency.

B.2.11 LEMMA *Let* $\Psi \xrightarrow{TS',tt} \Psi'$ *for all* $(TS', tt, \Psi') \in P_n$ *and let* P_{n+1} *be obtained from* P_n *by a reduction of the procedure. Then* $\Psi \xrightarrow{TS',tt} \Psi'$ *for all* $(TS', tt, \Psi') \in P_{n+1}$.

PROOF Proved by the rules individually. Let (TS', tt, Ψ') be the pair on which the reduction is applied.

- Rules 1 through 4 and 6 through 8 do not affect the temporal consistency constraints.
- Rule 5. Similar to the restricted case. Timer settings and $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ and $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2$ formulas are dealt with correctly. □

8.2.26 LEMMA *Let* $\Phi \subseteq cl(\varphi)$ *be a locally consistent set of formulas and let* $(TS', tt, \Phi') \in \mathrm{NF}(\Phi)$. *Then* Φ' *is locally consistent. Moreover,* $\Phi \xrightarrow{TS',tt} \Phi'$ *for any* $(TS', tt, \Phi') \in \mathrm{NF}(\mathrm{Next}(\Phi))$.

PROOF Follows directly from corollary B.2.10 and lemma B.2.11. □

B.2.5 Proof of the lemmas 8.2.28, 8.2.29 and 8.2.30

These lemmas tell us how to construct a run for a timed state sequence that satisfies the formula of the tableau automaton. Lemma 8.2.28 tells us how to select an appropriate initial extended location and lemmas 8.2.29 and 8.2.30 tell us how to select subsequent locations in the case of left-closed and left-open intervals respectively. First, we define temporal consistency constraints w.r.t. marked formulas.

B.2.12 DEFINITION *Let* Ψ *be a set of formulas, some of which may be marked. Then* $\mathrm{MNext}(\Psi, tt)$ *is the pair* (TS, Ψ') *where TS and* Ψ' *are the smallest sets such that*

- *if* $\psi = \varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$ *is marked and* Ψ *is non-trivial for* ψ, *then* Ψ' *contains* ψ;
- *if* $\psi = \varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2$ *is marked and* Ψ *is non-trivial for* ψ, *then* Ψ' *contains* ψ;
- *if* $\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2$ *is marked,* Ψ *is non-trivial for* ψ *and* $tt = \#oc$, *then* Ψ' *contains* $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2$ *and TS contains* $y_\psi := d$;
- *if* $\psi = \varphi_1 \mathsf{V}_{\leq d} \varphi_2$ *is marked,* Ψ *is non-trivial for* ψ *and* $tt = \#co$, *then* Ψ' *contains* $\varphi_1 \mathsf{V}_{\leq y_\psi} \varphi_2$ *and TS contains* $y_\psi := d$;
- *if* $\psi = \varphi_1 \mathsf{V}_{< y_{\psi'}} \varphi_2$ *is marked and* Ψ *is non-trivial for* ψ, *then* Ψ' *contains* ψ;
- *if* $\psi = \varphi_1 \mathsf{V}_{\leq y_{\psi'}} \varphi_2$ *is marked and* Ψ *is non-trivial for* ψ, *then* Ψ' *contains* ψ. □

If all formulas are marked, then $\mathrm{Next}(\Psi, tt) = \mathrm{MNext}(\Psi, tt)$. The rules in the normal form procedure are different for the different types of transitions. This is caused by the fact that the interval may have or may not have a first instant where the formulas of the new interval need to hold. Both types of reduction rules maintain slightly different invariants.

B.2.13 DEFINITION *Let P contain triples* (TS, tt, Ψ)*, let* $\Psi \subseteq cl(\varphi)$ *for some MITL*$_{\leq}$ *formula* φ*. Let* $\overline{\rho}$ *be a timed state sequence, let* ν *be a timer valuation such that for every timer x in the domain of* ν*, corresponding to the Until formula* $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ *or the Release formula* $\varphi_1 \mathsf{V}_{\leq d} \varphi_2$*,* $\nu(x) \leq d$*.*
The predicate $\mathrm{Inv}^{\#oc}(P, \overline{\rho}, \nu)$ *holds iff for any* φ*-fine interval sequence* $\overline{I}$*, there is some* $(TS, \#oc, \Psi) \in P$ *such that*

(i) for all $t \in \overline{I}(0)$*,* $\overline{\rho}^t \models_{TS[\nu]-t} \Psi$*;*

(ii) $\overline{\rho}^{|\overline{I}(0)|} \models_{TS[\nu]-|\overline{I}(0)|} \mathrm{MNext}(\Psi, tt')$ *where* tt' *matches the transition from* $\overline{I}(0)$ *to* $\overline{I}(1)$*.*

For the intervals starting in a left-open fashion we have the following invariant, which is identical except for the fact that the formulas in Ψ *need not hold at* $t = 0$*, but instead for the left-open interval following the first singular interval* $[0, 0]$*.*
The predicate $\mathrm{Inv}^{\#co}(P, \overline{\rho}, \nu)$ *holds iff for any* φ*-fine interval sequence* $\overline{I} = [0, 0] I_1 I_2 \ldots$*, there is some* $(TS, \#co, \Psi) \in P$ *such that*

(i) for all $t \in \overline{I}(1)$*,* $\overline{\rho}^t \models_{TS[\nu]-t} \Psi$*;*

(ii) $\overline{\rho}^{|\overline{I}(1)|} \models_{TS[\nu]-|\overline{I}(1)|} \mathrm{MNext}(\Psi, tt')$ *where* tt' *matches the transition from* $\overline{I}(1)$ *to* $\overline{I}(2)$*.* □

B.2.14 LEMMA *Let* P_n *contain triples* (TS, tt, Ψ) *where* $\Psi \subseteq cl(\varphi)$ *for some MITL*$_{\leq}$ *formula* φ*. Let* $\overline{\rho}$ *be a timed state sequence, let* ν *be a timer valuation, assume* $\mathrm{Inv}^{tt}(P_n, \overline{\rho}, \nu)$ *for some* $tt \in \{\#oc, \#co\}$ *and let* P_{n+1} *be obtained from* P_n *by applying one of the rules of table 8.4. Then* $\mathrm{Inv}^{tt}(P_{n+1}, \overline{\rho}, \nu)$*.*

PROOF If there is such a set other than the one on which the case applies, then the lemma follows trivially. If not, it is shown case by case that the invariant is preserved. Let $(TS, tt, \Psi) \in P_n$ and let $\psi \in \Psi$ on which the case is applied.

- Cases 1,2,3 and 4 are trivial or similar to the untimed case and consistency constraints are not affected.
- Case 5.1. In this case $tt = \#oc$, $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \notin \Psi$ and $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2 \notin \Psi$. Thus the addition of $x_\psi := d$ to TS has no influence on the existing formulas in Ψ. The new formula $(\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2)$ is satisfied in the timer environment after setting x_ψ to d, since $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ is satisfied. It is satisfied in the entire interval, since either φ_2 holds or the interval cannot be longer than d. With regard to part (ii) of the invariant, we observe that it still holds, since no new consistency constraints will be added after marking $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$.
- Case 5.2. In this case $tt = \#co$, $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \notin \Psi$ and $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2 \notin \Psi$. Thus the addition of $x_\psi := d$ to TS has no influence on the existing formulas in Ψ. The new formula $(\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2)$ is satisfied during the second interval in the timer environment after setting x_ψ to d, since $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$ is satisfied in the interval. With regard to part (ii) of the invariant, we observe that it still holds, since no new consistency constraints will be added after marking $\varphi_1 \mathsf{U}_{\leq d} \varphi_2$.
- Case 5.3. In this case tt can be either $\#co$ or $\#oc$. In both cases, nothing is added to Ψ or TS. Therefore, the invariant trivially still holds.
- Case 5.4. Identical to case 5.3; nothing is added.

- Case 6 a.
  - If $tt = \#oc, \overline{\rho} \models_{TS[\nu]} \varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$ and thus either
    - $\overline{\rho} \models_{TS[\nu]} \varphi_2$. Then the new triple $(TS, \#oc, \Psi \cup \{\psi^*, \varphi_2\})$ satisfies the requirements. Requirement (i) is satisfied since the newly added φ_2 holds. For requirement (ii) no new consistency constraints are added;
    - $\overline{\rho} \not\models_{TS[\nu]} \varphi_2$, $TS[\nu](x_\psi) > 0$ and $\overline{\rho} \models_{TS[\nu]} \varphi_1$. Then the new triple $(TS, \#oc, \Psi \cup \{\psi^*, x_{\psi'} > 0, \varphi_1\})$ satisfies the requirements. Requirement (i) is satisfied since the newly added $x_{\psi'} > 0$ and φ_1 hold (the interval cannot be longer than $TS[\nu](x_{\psi'})$). For (ii) the consistency constraint $\varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$ is added and $\overline{\rho}^{|\overline{I}(0)|} \models_{TS[\nu]-|\overline{I}(0)|} TS'.\varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$ (TS' does not modify $x_{\psi'}$).
  - If $tt = \#co, \overline{\rho}$ satisfies $\varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$ in the environment $TS[\nu]$ on the second interval and thus
    - if it satisfies φ_2 then the new triple $(TS, \#co, \Psi \cup \{\psi^*, \varphi_2\})$ satisfies the requirements. Requirement (i) is satisfied since the newly added φ_2 holds. For requirement (ii) no new consistency constraints are added;
    - if it does not satisfy φ_2 then it must satisfy $x_{\psi'} > 0$ and φ_1 on the interval. Then the new triple $(TS, \#co, \Psi \cup \{\psi^*, x_{\psi'} > 0, \varphi_1\})$ satisfies the requirements. (i) is satisfied since the newly added $x_{\psi'} > 0$ and φ_1 hold (for every $t \in \overline{I}(0)$, $t < TS[\nu](x_{\psi'})$). For (ii) the consistency constraint $\varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$ is added, $\overline{\rho}^{|\overline{I}(1)|} \models_{TS[\nu]-|\overline{I}(1)|} TS'.\varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$ (TS' does not modify $x_{\psi'}$).
- Case 6 b.
  - If $tt = \#oc, \overline{\rho} \models_{TS[\nu]} \varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2$ and thus either:
    - $\overline{\rho} \models_{TS[\nu]} \varphi_2$. Then the new triple $(TS, \#oc, \Psi \cup \{\psi^*, \varphi_2\})$ satisfies the requirements. Requirement (i) is satisfied since the newly added φ_2 holds. For (ii) no new consistency constraints are added;
    - $\overline{\rho} \not\models_{TS[\nu]} \varphi_2$. Then $TS[\nu](x_\psi) \geq 0$ and $\overline{\rho} \models_{TS[\nu]} \varphi_1$ Then the new triple $(TS, \#oc, \Psi \cup \{\psi^*, x_{\psi'} \geq 0, \varphi_1\})$ satisfies the requirements. (i) is satisfied since the newly added $x_{\psi'} \geq 0$ and φ_1 hold (again the interval is no longer than $TS[\nu](x_{\psi'})$). For (ii) the consistency constraint $\varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2$ is added and $\overline{\rho}^{|\overline{I}(0)|} \models_{TS[\nu]-|\overline{I}(0)|} TS'.\varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2$ (TS' does not modify $x_{\psi'}$).
  - If $tt = \#co, \overline{\rho}$ satisfies $\varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2$ in the environment $TS[\nu]$ on the interval and thus
    - if it satisfies φ_2 in the interval, then the new triple $(TS, \#co, \Psi \cup \{\psi^*, \varphi_2\})$ satisfies the requirements. Requirement (i) is satisfied since the newly added φ_2 holds. For requirement (ii) no new consistency constraints are added;
    - if it does not satisfy φ_2, then $\nu(x_{\psi'}) > 0$ and thus it satisfies $x_{\psi'} \geq 0$ and φ_1 on the interval (the interval is no longer than $TS[\nu](x_{\psi'})$). Then the new triple $(TS, \#co, \Psi \cup \{\psi^*, x_{\psi'} \geq 0, \varphi_1\})$ satisfies the requirements. (i) is satisfied since the newly added $x_{\psi'} \geq 0$ and φ_1 hold. For requirement (ii) the consistency constraint $\varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2$ is added;
      $\overline{\rho}^{|\overline{I}(1)|} \models_{TS[\nu]-|\overline{I}(1)|} TS'.\varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2$ (TS' does not modify $x_{\psi'}$).
- Case 7. φ_2 is obviously satisfied.
  - If φ_1 is satisfied, then the requirements clearly hold for the new triple $(TS, tt, \Psi \cup \{\psi^*, \varphi_1, \varphi_2\})$. No consistency constraints are added.

  - If φ_1 is not satisfied, then it follows from ψ-fineness of $\overline{I}$ that ψ is still satisfied at the end of the interval and depending on the type of transition to the next interval, the corresponding added consistency constraint holds.

- Case 8 a. $\varphi_1 \mathsf{V}_{<y_{\psi'}} \varphi_2$ holds and thus $TS[\nu](y_{\psi'}) \leq 0$ or φ_2 holds.
  - If $y_{\psi'} \leq 0$ holds, then the triple $\left(TS, tt, \Psi \cup \{\psi^*, y_{\psi'} \leq 0\}\right)$ satisfies the requirements. No consistency constraints are introduced.
  - If φ_1 holds, then the triple $(TS, tt, \Psi \cup \{\psi^*, \varphi_1, \varphi_2\})$ satisfies the requirements. No consistency constraints are introduced.
  - If neither $y_{\psi'} \leq 0$ nor φ_1 holds, then the triple $(TS, tt, \Psi \cup \{\psi^*, \varphi_2\})$ satisfies the requirements. The consistency constraint $\varphi_1 \mathsf{V}_{<y_{\psi'}} \varphi_2$ is added. Since $\overline{I}$ is φ_1-fine and $TS[\nu](y_{\psi'}) \leq d, \overline{\rho}^{|\overline{I}(0)|} \models_{TS[\nu]-|\overline{I}(0)|} TS'.\varphi_1 \mathsf{V}_{<y_{\psi'}} \varphi_2$.

- Case 8 b.
  - If $tt = \#oc$, $\varphi_1 \mathsf{V}_{\leq y_{\psi'}} \varphi_2$ holds and thus $TS[\nu](y_{\psi'}) < 0$ or φ_2 holds.Further
    - if $y_{\psi'} < 0$ holds, then the triple $\left(TS, \#oc, \Psi \cup \{\psi^*, y_{\psi'} < 0\}\right)$ satisfies the requirements. No consistency constraints are introduced.
    - if φ_1 holds, then the triple $(TS, \#oc, \Psi \cup \{\psi^*, \varphi_1, \varphi_2\})$ satisfies the requirements. No consistency constraints are introduced
    - if neither $y_{\psi'} < 0$ nor φ_1 holds, then the triple $(TS, \#oc, \Psi \cup \{\psi^*, \varphi_2\})$ satisfies the requirements. The consistency constraint $\varphi_1 \mathsf{V}_{\leq y_{\psi'}} \varphi_2$ is added. Since $\overline{I}$ is φ_1-fine and $TS[\nu](y_{\psi'}) \leq d, \overline{\rho}^{|\overline{I}(0)|} \models_{TS[\nu]-|\overline{I}(0)|} TS'.\varphi_1 \mathsf{V}_{\leq y_{\psi'}} \varphi_2$.
  - If $tt = \#co$, $\varphi_1 \mathsf{V}_{\leq y_{\psi'}} \varphi_2$ holds and thus $TS[\nu](y_{\psi'}) \leq 0$ or φ_2 holds. Further
    - if $TS[\nu](y_{\psi'}) \leq 0$, then $y_{\psi'} < 0$ holds during the interval and the triple $\left(TS, \#oc, \Psi \cup \{\psi^*, y_{\psi'} < 0\}\right)$ satisfies the requirements, no consistency constraints are introduced
    - if φ_1 holds, then the triple $(TS, \#oc, \Psi \cup \{\psi^*, \varphi_1, \varphi_2\})$ satisfies the requirements. No consistency constraints are introduced;
    - if neither $y_{\psi'} < 0$ nor φ_1 holds, then the triple $(TS, \#oc, \Psi \cup \{\psi^*, \varphi_2\})$ satisfies the requirements. The consistency constraint $\varphi_1 \mathsf{V}_{\leq y_{\psi'}} \varphi_2$ is added. Since $\overline{I}$ is φ_1-fine and $TS[\nu](y_{\psi'}) \leq d, \overline{\rho}^{|\overline{I}(0)|} \models_{TS[\nu]-|\overline{I}(0)|} TS'.\varphi_1 \mathsf{V}_{\leq y_{\psi'}} \varphi_2$. □

8.2.28 LEMMA *Let $\Phi \subseteq cl(\varphi)$ be a set of formulas. Let $\overline{\rho} \models_\nu \Phi$ and let $\overline{I}$ be a φ-fine interval sequence. Then there is some $(TS, \#oc, \Phi') \in \mathrm{NF}(\Phi)$ such that $\overline{\rho}^t \models_{TS[\nu]-t} \Phi'$ for all $t \in \overline{I}(0)$ and $\overline{\rho}^{|\overline{I}(0)|} \models_{TS[\nu]-|\overline{I}(0)|} \mathrm{Next}(\Phi', tt)$ where tt matches the transition from $\overline{I}(0)$ to $\overline{I}(1)$.*

PROOF Follows from lemma B.2.14. The procedure is started from $P_0 = \{(\varnothing, \#oc, \Phi), (\varnothing, \#co, \Phi)\}$. $\mathrm{Inv}^{\#oc}(P_0, \overline{\rho}, \nu)$ holds. Thus $\mathrm{Inv}^{\#oc}(P_n, \overline{\rho}, \nu)$ holds when the procedure ends and from this the result follows. □

8.2.29 LEMMA *Let Φ be a location in the on-the-fly tableau automaton A_φ, let $\overline{\rho}$ and ν be such that $\overline{\rho} \models_\nu \mathrm{Next}(\Phi, \#oc)$ and let $\overline{I}$ be a φ-fine interval sequence. Then there is an edge $\left(\Phi, TS', \#oc, \Phi'\right)$ of A_φ such that $\overline{\rho}^t \models_{TS'[\nu]-t} \Phi'$ for all $t \in \overline{I}(0)$ and $\overline{\rho}^{|\overline{I}(0)|} \models_{TS'[\nu]-|\overline{I}(0)|} \mathrm{Next}(\Phi', tt')$ where tt′ matches the transition from $\overline{I}(0)$ to $\overline{I}(1)$.*

PROOF It follows from lemma B.2.14 and the construction of the on-the-fly tableau. The invariant $\mathrm{Inv}^{\#oc}$ holds for some edge. □

8.2.30 LEMMA *Let Φ be a location in the on-the-fly tableau automaton A_φ, let $\overline{\rho}$ and ν be such that $\overline{\rho} \models_\nu \mathrm{Next}(\Phi, \#co)$ and let $\overline{I} = [0,0]I_1I_2\ldots$ be a φ-fine interval sequence. Then there is an edge $(\Phi, TS', \#co, \Phi')$ of A_φ such that $\overline{\rho}^t \models_{TS'[\nu]-t} \Phi'$ for all $t \in \overline{I}(1)$ and $\overline{\rho}^{|\overline{I}(1)|} \models_{TS'[\nu]-|\overline{I}(1)|} \mathrm{Next}(\Phi', tt')$ where tt' matches the transition from $\overline{I}(1)$ to $\overline{I}(2)$.*

PROOF It follows from lemma B.2.14 and the construction of the on-the-fly tableau. The invariant $\mathrm{Inv}^{\#co}$ holds for some edge. □

B.2.6 Proof of the Lemmas on Informative Prefixes

In this section we prove the lemmas 8.3.3, 8.3.4, 8.3.6, 8.3.8 and 8.3.7.

B.2.15 DEFINITION *Let Ψ_1 and Ψ_2 be two sets of extended formulas and let ν_1 and ν_2 be two timer valuations. Then $(\Psi_1, \nu_1) \preceq (\Psi_2, \nu_2)$ iff for every $\psi \in \Psi_1$*

- *if ψ is an MITL$_\leq$ formula, then $\psi \in \Psi_2$;*
- *if $\psi = x_{\psi'} > 0$, then either $x_{\psi'} > 0 \in \Psi_2$ and $\nu_1(x_{\psi'}) \geq \nu_2(x_{\psi'})$ or $x_{\psi'} \geq 0 \in \Psi_2$ and $\nu_1(x_{\psi'}) > \nu_2(x_{\psi'})$;*
- *if $\psi = \varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$, then either $\varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2 \in \Psi_2$ and $\nu_1(x_{\psi'}) \geq \nu_2(x_{\psi'})$ or $\varphi_1 \mathsf{U}_{\leq x_{\psi'}+\epsilon} \varphi_2 \in \Psi_2$ and $\nu_1(x_{\psi'}) > \nu_2(x_{\psi'})$;*
- *if $\psi = y_{\psi'} < 0$, then either $y_{\psi'} < 0 \in \Psi_2$ and $\nu_1(y_{\psi'}) \leq \nu_2(y_{\psi'})$ or $y_{\psi'} \leq 0 \in \Psi_2$ and $\nu_1(y_{\psi'}) < \nu_2(y_{\psi'})$;*
- *if $\psi = x_{\psi'} \geq 0$, then $\nu_1(x_{\psi'}) \geq \nu_2(x_{\psi'})$ and $x_{\psi'} \geq 0 \in \Psi_2$ or $x_\psi > 0 \in \Psi_2$;*
- *if $\psi = \varphi_1 \mathsf{U}_{\leq x_{\psi'}+\epsilon} \varphi_2$, then $\nu_1(x_{\psi'}) \geq \nu_2(x_{\psi'})$ and either $\varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2 \in \Psi_2$ or $\varphi_1 \mathsf{U}_{\leq x_{\psi'}+\epsilon} \varphi_2 \in \Psi_2$;*
- *if $\psi = y_{\psi'} \leq 0$, then $\nu_1(y_{\psi'}) \leq \nu_2(y_{\psi'})$ and either $y_{\psi'} \leq 0 \in \Psi_2$ or $y_{\psi'} < 0 \in \Psi_2$.*
- *if $\psi = \varphi_1 \mathsf{V}_{\leq y_{\psi'}} \varphi_2$, then $\varphi_1 \mathsf{V}_{\leq d} \varphi_2 \in \Psi_2$ or both $\varphi_1 \mathsf{V}_{\leq y_{\psi'}} \varphi_2 \in \Psi_2$ and $\nu_1(y_{\psi'}) \leq \nu_2(y_{\psi'})$;*
- *if $\psi = \varphi_1 \mathsf{V}_{< y_{\psi'}} \varphi_2$, then either*
  - *$\varphi_1 \mathsf{V}_{\leq d} \varphi_2 \in \Psi_2$ (and $\nu_1(y_{\psi'}) \leq d$) or*
  - *$\nu_1(y_{\psi'}) \leq \nu_2(y_{\psi'})$ and $\varphi_1 \mathsf{V}_{< y_\psi} \varphi_2 \in \Psi_2$ or $\varphi_1 \mathsf{V}_{\leq y_\psi} \varphi_2 \in \Psi_2$.*

(Relation $\preceq$ is reflexive and transitive.)

8.3.3 LEMMA *Let Φ be a set of formulas, let ν be a timer valuation and let $\overline{IS} = (\overline{\Phi}, \overline{TS}, \overline{\nu}, \overline{I})$ be an informative sequence such that $(\Phi, \nu) \preceq (\overline{\Phi}(0), \overline{\nu}(0))$. Then there is some $(TS, \Phi', \#oc) \in \mathrm{NF}(\Phi, \#oc)$ such that $(\Phi', TS[\nu]) \preceq (\overline{\Phi}(0), \overline{\nu}(0))$ and $(\Phi'', TS''[TS[\nu] - |\overline{I}(0)|]) \preceq (\overline{\Phi}(1), \overline{\nu}(1))$ where $(TS'', \Phi'') = \mathrm{Next}(\Phi', tt)$ if tt is the type of transition from $\overline{I}(0)$ to $\overline{I}(1)$.*

The lemma is proved similar to lemma 7.4.8 in the untimed case.

PROOF In the following proof, the predicate $Inv(P, \nu, \overline{IS})$ holds iff $\overline{IS} = (\overline{\Phi}, \overline{TS}, \overline{\nu}, \overline{I})$, there is some $(TS, \#oc, \Psi) \in P$ such that $(\Psi, TS[\nu]) \preceq (\overline{IS}(0), \overline{\nu}(0))$ and $(\Phi'', TS''[TS[\nu] - |\overline{I}(0)|]) \preceq (\overline{IS}(1), \overline{\nu}(1))$ where $(TS'', \Phi'') = \mathrm{MNext}(\Psi, tt)$ if tt matches the transition from $\overline{I}(0)$ to $\overline{I}(1)$. Then if $Inv(P, \nu, \overline{IS})$ is an invariant of the DTNF procedure, the lemma follows. The proof proceeds for every reduction rule separately. If the rule was applied on another tuple, then the invariant remains true. Otherwise we get:

- Case 6b (for instance). Then $\psi = \varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2 \in \Psi$ and ψ is unmarked ($\Psi' = \Psi \backslash \{\psi\}$). Then $TS[\nu](x_{\psi'}) \geq \overline{\nu}(0)(x_{\psi'})$ and either of the following is true:
  - $\varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2 \in \overline{\Phi}(0)$. By local consistency, either $\varphi_2 \in \overline{\Phi}(0)$ or both $\varphi_1 \in \overline{\Phi}(0)$ and $x_{\psi'} \geq 0 \in \overline{\Phi}(0)$. Then
    - if $\varphi_2 \in \overline{\Phi}(0)$, $(\Psi' \cup \{\psi^*, \varphi_2\}, TS[\nu]) \preceq (\overline{\Phi}(0), \overline{\nu}(0))$ and nothing is added to TS. $(\Phi', TS'[TS[\nu] - |\overline{I}(0)|]) \preceq (\Phi'', TS''[TS[\nu] - |\overline{I}(0)|]) \preceq (\overline{\Phi}(1), \overline{\nu}(1))$, where $(TS', \Phi') = \mathrm{MNext}(\Psi' \cup \{\psi^*, \varphi_2\}, \#oc)$ and $(TS'', \Phi'') = \mathrm{MNext}(\Psi, \#oc)$;
    - if $\varphi_2 \notin \overline{\Phi}(0)$, we have by local consistency, $\varphi_1 \in \overline{\Phi}(0)$ and $x_{\psi'} \geq 0 \in \overline{\Phi}(0)$. Thus $(\Psi' \cup \{\psi^*, \varphi_1, x_{\psi'} \geq 0\}, TS[\nu]) \preceq (\overline{\Phi}(0), \overline{\nu}(0))$ and $(\Phi', TS'[TS[\nu] - |\overline{I}(0)|]) \preceq (\overline{\Phi}(1), \overline{\nu}(1))$ where $(TS', \Phi') = \mathrm{MNext}(\Psi' \cup \{\psi^*, \varphi_1, x_{\psi'} \geq 0\}, \#oc) = \mathrm{MNext}(\Psi, \#oc) \cup \{\varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2\}$, by temporal consistency since $\varphi_2 \notin \overline{\Phi}(0)$ and $TS[\nu](x_{\psi'}) \geq \overline{\nu}(0)(x_{\psi'})$.
  - $\varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2 \in \overline{\Phi}(0)$. By local consistency, either $\varphi_2 \in \overline{\Phi}(0)$ or both $\varphi_1 \in \overline{\Phi}(0)$ and $x_{\psi'} > 0 \in \overline{\Phi}(0)$. Then
    - if $\varphi_2 \in \overline{\Phi}(0)$, then the case is identical to the case $\varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2 \in \overline{\Phi}(0)$;
    - if $\varphi_2 \notin \overline{\Phi}(0)$, we have by local consistency, $\varphi_1 \in \overline{\Phi}(0)$ and $x_{\psi'} > 0 \in \overline{\Phi}(0)$. Thus $(\Psi' \cup \{\psi^*, \varphi_1, x_{\psi'} \geq 0\}, TS[\nu]) \preceq (\overline{\Phi}(0), \overline{\nu}(0))$ and $(\Phi', TS'[TS[\nu] - |\overline{I}(0)|]) \preceq (\overline{\Phi}(1), \overline{\nu}(1))$ where $(TS', \Phi') = \mathrm{MNext}(\Psi' \cup \{\psi^*, \varphi_1, x_{\psi'} \geq 0\}, \#oc) = \mathrm{MNext}(\Psi, \#oc) \cup \{\varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2\}$, by temporal consistency since $\varphi_2 \notin \overline{\Phi}(0)$ and $TS[\nu](x_{\psi'}) \geq \overline{\nu}(0)(x_{\psi'})$.

□

8.3.4 LEMMA *Let Φ be a set of formulas, let ν be a timer valuation and let $\overline{IS} = (\overline{\Phi}, \overline{TS}, \overline{\nu}, \overline{I})$ be an informative sequence such that $l(\overline{I}(1)) = 0$ and $(\Phi, \nu) \preceq (\overline{\Phi}(1), \overline{\nu}(1))$. Then there is some $(TS, \Phi', \#co) \in \mathrm{NF}(\Phi, \#co)$ such that $(\Phi', TS[\nu]) \preceq (\overline{\Phi}(1), \overline{\nu}(1))$ and $(\Phi'', TS''[TS[\nu] - |\overline{I}(1)|]) \preceq (\overline{\Phi}(2), \overline{\nu}(2))$ where $(TS'', \Phi'') = \mathrm{Next}(\Phi', tt)$ if tt is the type of transition from $\overline{I}(1)$ to $\overline{I}(2)$.*

This lemma is similar to the previous one, but this time for the *#co* case.

PROOF In the following lemmas, the predicate $Inv(P, \nu, \overline{IS})$ holds iff (let $\overline{IS} = (\overline{\Phi}, \overline{TS}, \overline{\nu}, \overline{I})$) there is some $(TS, \#co, \Psi) \in P$ such that $(\Psi, TS[\nu]) \preceq (\overline{IS}(1), \overline{I}(1))$ and $(\mathrm{MNext}(\Psi, tt), TS[\nu] - |\overline{I}(1)|) \preceq (\overline{IS}(2), \overline{\nu}(2))$ where tt matches the transition from $\overline{I}(1)$ to $\overline{I}(2)$. Then if $Inv(P, \nu, \overline{IS})$ is an invariant of the DTNF procedure, the lemma follows. The proof proceeds for every reduction rule separately. If the rule was applied on another tuple, then the invariant remains true. Otherwise we get:

- Case 5, for instance. Then $\psi = \varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in \Psi$ is unmarked. Let $\Psi' = \Psi \backslash \{\psi\}$. Then $\varphi_1 \mathsf{U}_{\leq d} \varphi_2 \in \overline{\Phi}(1)$ and thus $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \in \overline{\Phi}(1)$ or $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2 \in \overline{\Phi}(1)$. There are three cases to the rule
  - $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \notin \Psi$, $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2 \notin \Psi$.
    Then $TS' = \{x_\psi := d\}[TS]$ and $\Psi'' = \Psi' \cup \{\varphi_1 \mathsf{U}_{\leq d} \varphi_2^*, \varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2\}$
    $(\Psi'', TS'[\nu]) = (\Psi' \cup \{\varphi_1 \mathsf{U}_{\leq d} \varphi_2^*, \varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2\}, \{x_\psi := d\}[TS][\nu]) \preceq (\overline{IS}(1), \overline{\nu}(1))$
    since $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \in \overline{\Phi}(1)$ or $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2 \in \overline{\Phi}(1)$ and $\overline{\nu}(1)(x_\psi) \leq d$.
  - $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \in \Psi$.
    Then $TS' = TS$ and $\Psi'' = \Psi' \cup \{\psi^*, \varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2\}$. $(\Psi'', TS'[\nu]) =$
    $(\Psi' \cup \{\varphi_1 \mathsf{U}_{\leq d} \varphi_2^*, \varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2\}, TS[\nu]) \preceq (\overline{\Phi}(1), \overline{\nu}(1))$ because
    - if $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2 \in \overline{\Phi}(1)$, then $TS[\nu](x_\psi) \geq \overline{\nu}(1)(x_\psi)$ and
    - if $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2 \in \overline{\Phi}(1)$, then $TS[\nu](x_\psi) > \overline{\nu}(1)(x_\psi)$.
  - $\varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2 \in \Psi$.
    Then $TS' = TS$ and $\Psi'' = \Psi' \cup \{\psi^*, \varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2\}$. $(\Psi'', TS'[\nu]) =$
    $(\Psi' \cup \{\varphi_1 \mathsf{U}_{\leq d} \varphi_2^*, \varphi_1 \mathsf{U}_{\leq x_\psi + \epsilon} \varphi_2\}, TS[\nu]) \preceq (\overline{\Phi}(1), \overline{\nu}(1))$ because
    $TS[\nu](x_\psi) \geq \overline{\nu}(1)(x_\psi)$.

  In all three cases, $(\Phi', TS'[TS[\nu] - |\overline{I}(1)|]) \preceq (\Phi'', TS''[TS[\nu] - |\overline{I}(1)|]) \preceq (\overline{IS}(2), \overline{\nu}(2))$ where $(TS', \Phi') = \mathrm{MNext}(\Psi'', tt)$ and $(TS'', \Phi'') = \mathrm{MNext}(\Psi, tt)$ (by the induction hypothesis) □

B.2.16 DEFINITION *A pair (Φ, ν) contradicts another pair (Φ', ν') if one of the following holds:*

- *there is some MITL$_\leq$ formula $\varphi \in \Phi$ such that $\neg\varphi \in \Phi'$;*
- *there is some $\psi \in \Phi$ for which one of the following applies*
  - *if $\psi = \varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$, then there is some $(\neg\varphi_1)\mathsf{V}_{\leq y}(\neg\varphi_2) \in \Phi'$ and $\nu(x_{\psi'}) \leq \nu'(y)$;*
  - *if $\psi = \varphi_1 \mathsf{U}_{\leq x_{\psi'} + \epsilon} \varphi_2$, then there is some $(\neg\varphi_1)\mathsf{V}_{\leq y}(\neg\varphi_2) \in \Phi'$ and $\nu(x_{\psi'}) < \nu'(y)$;*
  - *if $\psi = \varphi_1 \mathsf{V}_{< y_{\psi'}} \varphi_2$, then there is some $(\neg\varphi_1)\mathsf{U}_{\leq x}(\neg\varphi_2) \in \Phi'$ and $\nu'(x) < \nu(y_{\psi'})$;*
  - *if $\psi = \varphi_1 \mathsf{V}_{\leq y_{\psi'}} \varphi_2$, then there is some $(\neg\varphi_1)\mathsf{U}_{\leq x}(\neg\varphi_2) \in \Phi'$ and $\nu'(x) \leq \nu(y_{\psi'})$.*

B.2.17 DEFINITION *A prefix $\overline{\tau}$ is an informative bad prefix for the pair (Φ, ν) if there is an informative sequence $\overline{IS} = (\overline{\Phi}, \overline{TS}, \overline{\nu}, \overline{I})$ such that either of the following holds:*

- *(Φ, ν) contradicts $(\overline{\Phi}(0), \overline{\nu}(0))$;*

- *or* $(TS', \Phi') = \mathrm{Next}(\Phi, tt)$ *where tt is the type of transition from* $\overline{I}(0)$ *to* $\overline{I}(1)$ *and* $(\Phi', TS'[\nu - |\overline{I}(0)|])$ *contradicts* $(\overline{\Phi}(1), \overline{\nu}(1))$.

Proof of lemma 8.3.6

8.3.6 LEMMA *If* $\overline{\tau}$ *is an informative bad prefix for* $\mathrm{NF}(\Phi, \#oc)$ *in timer environment* ν*, then* $\overline{\tau}$ *is an informative bad prefix for* Φ *in timer environment* ν.

This is proved by induction on the steps of the normal form procedure (compare the proof of lemma 7.4.4).

B.2.18 DEFINITION *A prefix* $\overline{\tau}$ *is an informative bad prefix for P in timer environment* ν *if for every* $(TS, \#oc, \Psi) \in P$, $\overline{\tau}$ *is an informative bad prefix for* $(TS, \#oc, \Psi)$ *in timer environment* ν.

B.2.19 DEFINITION *A prefix* $\overline{\tau}$ *is an informative bad prefix for* $(TS, \#oc, \Psi')$ *in timer environment* ν *if there is some informative sequence* $\overline{IS} = (\overline{\Phi}, \overline{TS}, \overline{\nu}, \overline{I})$ *such that either of the following is true:*

- $(\Psi', TS[\nu])$ *contradicts* $(\overline{\Phi}(0), \overline{\nu}(0))$*;*
- $(\Phi'', TS'[TS[\nu - |\overline{I}(0)|]])$ *contradicts* $(\overline{\Phi}(1), \overline{\nu}(1))$ *where* $(TS', \Phi'') = \mathrm{MNext}(\Psi', tt)$ *and tt matches the transition from* $\overline{I}(0)$ *to* $\overline{I}(1)$.

The induction step for the proof of lemma 8.3.6 is provided by the following lemma.

B.2.20 LEMMA *Let* $P_n \Rightarrow P_{n+1}$ *be a reduction step in the procedure. Let* $\overline{\tau}$ *be an informative bad prefix for* P_{n+1} *in* ν*. Then* $\overline{\tau}$ *is an informative bad prefix for* P_n *in* ν.

PROOF This lemma is proved by case analysis. Assume the reduction is applied to $(TS, \#oc, \Psi \cup \{\psi\})$ for unmarked formula ψ.

- Case 6a (for instance). $\psi = \varphi_1 \mathsf{U}_{\leq x_{\psi'}} \varphi_2$. Then $\overline{\tau}$ is an informative bad prefix for both $(TS, \#oc, \Psi \cup \{\psi^*, \varphi_2\})$ and $(TS, \#oc, \Psi \cup \{\psi^*, \varphi_1, x_{\psi'} > 0\})$. If there is an informative sequence that contradicts Ψ, then there is also one that contradicts P_n. Otherwise, there is one that disproves φ_2 and φ_1 or $x_{\psi'} > 0$ or $\varphi_1 \mathsf{U}_{\leq x_\psi} \varphi_2$ at the next interval. From this informative sequence one can construct an informative sequence that disproves P_n. □

Proof of lemma 8.3.7

8.3.7 LEMMA *Let* A_φ *be a tableau automaton, let* Φ *be a location of* A_φ *and let* $\overline{\tau}$ *be a* φ*-fine prefix of a state sequence for which there is no (accepting) run on* A_φ *starting from extended location* (Φ, ν)*. Then* $\overline{\tau}$ *is an informative bad prefix for* Φ *in timer environment* ν *or* $\overline{\tau}^{|\overline{I}(0)|}$ *is an informative bad prefix of* Φ' *in timer environment* $TS[\nu - |\overline{I}(0)|]$ *where* $\overline{\tau} = (\overline{\xi}, \overline{I})$, $(TS, \Phi') = \mathrm{Next}(\Phi, tt)$ *and tt matches the transition from* $\overline{I}(0)$ *to* $\overline{I}(1)$.

PROOF The lemma is proved by induction on the length of the timed state sequence $|\overline{\xi}|$, comparable to lemma 7.4.5.

- If $|\overline{\xi}| = 1$, then there is some atomic proposition or some timer condition in Φ that conflicts with $\overline{\xi}(0)$ of ν respectively. In both cases it is straightforward to construct an informative sequence.
- If $|\overline{\xi}| > 1$, then either the first location Φ contradicts, or all successor locations (with transitions of type corresponding to the transition from $\overline{l}(0)$ to $\overline{l}(1)$) do not accept the tail of $\overline{\tau}$. By induction and lemma 8.3.6, $\overline{\tau}^{|\overline{l}(0)|}$ is an informative bad prefix of Φ' in timer environment $TS[\nu - |\overline{l}(0)|]$ where $(TS, \Phi') = \text{Next}(\Phi, tt)$ and tt matches the transition from $\overline{l}(0)$ to $\overline{l}(1)$. □

Proof of lemma 8.3.8

8.3.8 LEMMA *Let $\psi \in \Phi$, let ν be a timer valuation and let $\overline{\tau}$ be an informative bad prefix for ψ in ν. Then there is no run for $\overline{\tau}$ on $[A_\varphi]$ starting from the extended location (Φ, ν).*

This is proved by induction on ψ and on the length of the prefix (compare the proof of lemma 7.4.6).

Bibliography

[1] L. Aceto. A static view of localities. *Formal Aspects of Computing*, 6(2):201–222, 1994.

[2] L. Aceto, P. Bouyer, A. Burgueño, and K.G. Larsen. The power of reachability testing for timed automata. In V. Arvind and R. Ramanujam, editors, *Proceedings of the 18th Conference on the Foundations of Software Technology and Theoretical Computer Science (FST&TCS'98), Chennai, India, Dec. 1998, LNCS Vol.1530*, pages 245–256, Berlin, 1998. Springer Verlag.

[3] L. Aceto, A. Burgueño, and K.G. Larsen. Model checking via reachability testing for timed automata. In B. Steffen, editor, *Proceedings of Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Lisbon, Portugal, March 28 - April 4, 1998, LNCS Vol. 1384*, pages 263–280, Berlin, 1998. Springer Verlag.

[4] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[5] R. Alur. *Techniques for Automatic Verification of Real-Time Systems*. PhD thesis, Stanford University, 1991.

[6] R. Alur, C. Courcoubetis, and D.L. Dill. Model checking for real-time systems. In *Proc. Of the Fifth Annual Symposium on Logic in Computer Science*, pages 414–425. IEEE Computer society Press, 1990.

[7] R. Alur and D.L. Dill. Automata for modeling real-time systems. In M.S. Paterson, editor, *Proc. Of ICALP90: Automata, Languages and Programming LNCS Vol. 443*, pages 322–335. Springer Verlag, 1990.

[8] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[9] R. Alur, T. Feder, and T. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, January 1996.

[10] R. Alur, T. Henzinger, and P.H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.

[11] R. Alur and T.A. Henzinger. Logics and models of real-time: A survey. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Real-Time Theory in Practice. REX Workshop Proceedings Mook, the Netherlands 3-7 June 1991, LNCS Vol. 600*, pages 74–106, Berlin, 1992. Springer Verlag.

[12] R. Alur and T.A. Henzinger. Real-time logics: Complexity and expressiveness. *Information and Computation*, 104(1):35–77, 1993.

[13] R. Alur and T.A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, 1994.

[14] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In A.J. Hu and M.Y. Vardi, editors, *Proceedings of the Tenth International Conference on Computer-Aided Verification (CAV 1998), LNCS 1427*, pages 521–525, Berlin, 1998. Springer Verlag.

[15] R. Alur, G.J. Holzmann, and D. Peled. An analyzer for message sequence charts. *Software - Concepts and Tools*, 17(2):70–77, 1996.

[16] P.H.M. America and J.J.M.M. Rutten. *A Parallel Object-Oriented Language: Design and Semantic Foundations*. PhD thesis, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands, 1989.

[17] D. P. Appenzeller and A. Kuehlmann. Formal verification of the PowerPC^{TM} microprocessor. In *Proceedings of the International Conference on Computer Design (ICCD'95, Oct)*, pages 79–84, 1995.

[18] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[19] J.C.M. Baeten and J.A. Bergstra. Real time process algebra with infinitesimals. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes 1994, Workshop in Computing Series*, pages 148–187. Springer Verlag, 1995.

[20] J.C.M. Baeten and J.A. Bergstra. Discrete time process algebra. *Formal Aspects of Computing*, 8(2):188–208, 1996.

[21] J.C.M. Baeten and C.A. Middelburg. Process algebra with timing: Real time and discrete time. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 10, pages 627–684. Elsevier Science, Amsterdam, 2001.

[22] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1990.

[23] H. Barringer, R. Kuiper, and A. Pnueli. A really abstract concurrent model and its temporal logic. In *Proc. 13th Annual ACM Symposium on Principles of Programming Languages*, pages 173–183, New York, January 1986. ACM Press.

[24] G. Behrmann, A. Fehnker, T. Hune, K.G. Larsen, P. Pettersson, and J. Romijn. Efficient guiding towards cost-optimality in UPPAAL. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS Vol. 2031*, pages 174–188, Berlin, 2001. Springer Verlag.

[25] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Proceedings of Hybrid Systems III: Verification and Control, DIMACS/SYCON Workshop, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA. LNCS, Vol. 1066*, pages 232–243, Berlin, 1995. Springer Verlag.

[26] J.A. Bergstra, J. Heering, and P. Klint. *Algebraic Specification*. ACM Press, New York, 1989.

[27] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1):109–137, 1984.

[28] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. Elsevier Science, Amsterdam, 2001.

[29] L.J. van Bokhoven, J.P.M. Voeten, and M.C.W. Geilen. Software synthesis for system-level design using process execution trees. In *Proceedings 25th Euromicro Conference, Milan, Italy, 1999*, pages 463–467, Los Alamitos, California, 1999. IEEE Computer Society Press.

[30] R. Bol and J.F. Groote. The meaning of negative premises in transition system specifications. *Journal of the ACM*, 43(5):863–914, September 1996.

[31] T. Bolognesi et al. Converging towards a timed LOTOS standard. *Computer Standards and Interfaces*, 16:87–118, 1994.

[32] T. Bolognesi and F. Lucidi. Timed process algebras with urgent interactions and a unique powerful binary operator. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Real-Time Theory in Practice. REX Workshop Proceedings Mook, the Netherlands 3-7 June 1991*, pages 124–48, Berlin, 1992. Springer Verlag.

[33] V. Bos and J.J.T. Kleijn. Formalisation of a production system modelling language: The operational semantics of χ core. *Fundamenta Informaticae*, 41(4):367–392, 2000.

[34] D. Bošnački. *Enhancing State Space Reduction Techniques for Model Checking*. PhD thesis, Eindhoven University of Technology, 2001.

[35] D. Bošnački and D. Dams. Integrating real-time in Spin: A prototype implementation. In *FORTE/PSTV'98*, pages 423–439. Kluwer, 1998.

[36] A. Bouajjani, S. Tripakis, and S. Yovine. On-the-fly model checking for real-time systems. In *Proc. 1997 IEEE Real-Time Systems Symposium RTSS'97, San Francisco, CA*. IEEE Computer Society Press, december 1997.

[37] G. Boudol and I. Castellani. Permutation of transitions: An event structure semantics for CCS and SCCS. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *LNCS, Vol. 354: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 411–427. Springer Verlag, Berlin, 1989.

[38] G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn. Observing localities. *Theoretical Computer Science*, 114:31–61, 1993.

[39] R.S. Boyer, M. Kaufmann, and J.S. Moore. The Boyer-Moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*, 29(2):27–62, 1995.

[40] J. Bradfield and C. Stirling. Modal logics and μ-calculi: An introduction. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 4, pages 293–330. Elsevier Science, Amsterdam, 2001.

[41] J.P. Briand, M.C. Fehri, L. Logrippo, and A. Obaid. Executing LOTOS specifications. In B. Sarikaya and G. Von Bochmann, editors, *Proceedings of PSTV'86, 6th International Workshop on Protocol Specification, Testing and Verification, 1986, June 10-13, Montreal, Quebec, Canada*, pages 73–84, North-Holland, 1987. Elsevier Science Publishers B.V.

[42] B. Brock, W.A. Hunt, and M. Kaufmann. The FM9001 microprocessor proof. Technical Report 86, Computational Logic, Inc., Austin, TX, 1994.

[43] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. International Conference Logic, Method and Philos. Sci. 1960*, pages 1–12, Stanford, 1962. Stanford University Press.

[44] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–179, 1992.

[45] W. Canfield, E.A. Emerson, and A. Saha. Checking formal specifications under simulation. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors*, pages 455–460, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[46] I. Castellani. *Bisimulations for Concurrency*. PhD thesis, University of Edinburgh, 1988.

[47] I. Castellani. Process algebras with localities. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 15, pages 945–1045. Elsevier Science, Amsterdam, 2001.

[48] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.

[49] L. Chen. An interleaving model for real-time systems. In A. Nerode and M. Taitslin, editors, *Logical Foundations of Computer Science Tver'92 Proc. 2nd Int. Symp. Tver, Russia, 20-24 July. 1992*, pages 81–92, Berlin, 1992. Springer Verlag.

[50] Y.-A. Chen, E. Clarke, P.-H. Ho, Y. Hoskote, T. Kam, M. Khaira, J. O'Leary, and X. Zhao. Verification of all circuits in a Floating-Point Unit using word-level model checking. In M. Srivas and A. Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96, Palo Alto, CA, Nov.) LNCS Vol. 1166*, pages 19–33, New York, NY, 1996. Springer Verlag.

[51] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8:117–141, 1974.

[52] E.M. Clarke, O. Grumberg, H. Hiraishi, S.Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2):217–232, 1995.

[53] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science, Asilomar Conference Center, Pacific Grove, California, 5-8 June 1989.*, pages 353–362. IEEE Computer Society Press, 1989.

[54] R. Cleaveland and D. Yankelevich. An operational framework for value-passing processes. In *Proceeding of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 326–338, New York, 1994. ACM.

[55] D. Coleman, P. Arnold, and S. Bodoff. *Object-Oriented Development : The Fusion Method.* Prentice Hall, Englewood Cliffs, 1994.

[56] S. Conrad. Compositional object specification and verification. In I. Rozman and M. Pivka, editors, *Proc. Int. Conf. On Software Quality (ICSQ'95), Maribor, Slovenia*, pages 55–64. University of Maribor Press, 1995.

[57] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[58] D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, July 1996.

[59] D.R. Dams. Flat fragments of CTL and CTL*: Separating the expressive and distinguishing powers. *Logic Journal of the IGPL*, 7(1):55–78, 1999.

[60] M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear temporal logic. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification: 11th International Conference Proceedings, CAV'99, Trento, Italy, July 6-10, 1999 (LNCS 1633)*, pages 249–260. Springer Verlag, 1999.

[61] P.R. D'Argenio, J.-P. Katoen, and E. Brinksma. Specification and analysis of soft real-time systems: Quantity and quality. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, Phoenix, Arizona, USA, pages 104–114. IEEE Computer Society Press, 1999.

[62] J. Davies, D.M. Jackson, G.M. Reed, J.N Reed, A.W. Roscoe, and S.A. Schneider. Timed CSP: theory and practice. In J.W. de Bakker, C. Huizing C, W.P. de Roever, and G. Rozenberg, editors, *Real-Time Theory in Practice. REX Workshop Proceedings Mook, the Netherlands 3-7 June 1991*, pages 640–75, Berlin, 1992. Springer Verlag.

[63] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool Kronos. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Proceedings of Hybrid Systems III: Verification and Control, DIMACS/SYCON Workshop, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA. LNCS, Vol. 1066*, pages 208–219, Berlin, 1995. Springer Verlag.

[64] F. de Boer and J. Hooman. The real-time behaviour of asynchronously communicating processes. In J. Vytopil, editor, *Proceedings of the 2nd Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, Nijmegen, The Netherlands, January 8-10*, pages 451–472, Berlin, 1991. Springer Verlag.

[65] M.L. de Leijer. A POOSL-compiler for Smalltalk and C++. Master's thesis, Eindhoven University of Technology, 1997.

[66] T.T.H. Dennemans. Specification and design of a distributed real-time reactive control system, using the method Software / Hardware Engineering. Master's thesis, Faculty of Electrical Engineering, Eindhoven University of Technology, 1998.

[67] D. D'Souza. On-the-fly verification for linear time temporal logic. Master's thesis, SPIC Mathematical Institute, Madras, June 1997.

[68] H.-D. Ehrich, C. Caleiro, A. Sernadas, and G. Denker. Logics for specifying concurrent information systems. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 167–198. Kluwer, 1998.

[69] P. van Eijk. A comparison of behavioural language simulators. In B. Sarikaya and G. Von Bochmann, editors, *Proceedings of PSTV'86 International Workshop on Protocol Specification, Testing and Verification, June 10-13, Montreal, Quebec, Canada*, pages 85–96, North-Holland, 1986. Elsevier Science Publishers B.V.

[70] P. van Eijk. The design of a simulator tool. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Language LOTOS*, pages 351–390. Elsevier Science Publishers B.V., North-Holland, 1989.

[71] P.H.J. van Eijk, C. A. Vissers, and M. Diaz. *The Formal Description Technique LOTOS : Results of the ESPRIT/SEDOS Project*. North-Holland, Amsterdam, 1989.

[72] E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[73] K. Etessami and G. Holzman. Optimising Büchi automata. In C. Palamidessi, editor, *Proceedings of the 11th Int. Conf. On Concurrency Theory (CONCUR'2000), Pennsylvania State University, Pennsylvania, USA, August 22-25, 2000, LNCS 1877*, pages 153–167, Berlin, 2000. Springer.

[74] P.-A. Etique. *Service Specification Verification and Validation for the Intelligent Network.* PhD thesis, École Polytechnnique Fédérale de Lausanne, 1995.

[75] ETSI. *ETS 300-417-1-1 - Transmission and Multiplexing (TM); Generic Functional Requirements for Synchronous Digital Hierarchy (SDH) Equipment; Part 1-1: Generic Processes and Performance.* ETSI, 1996.

[76] J.W.G. Fleurkens, C.A.J. van Eijk, and J.A.G. Jess. Run-time consistency checking in discrete simulation models. In *Proceedings of the European Design and Test Conference, 1995. ED&TC; 1995*, pages 223–227, Brussels, 1995. IEEE Computer Society Press.

[77] W.J. Fokkink, J.F.Th. Kamperman, and H.R. Walters. Within ARM's reach: Compilation of left-linear rewrite systems via minimal rewrite systems. *ACM Transactions on Programming Languages and Systems*, 20(3):679–706, May 1998.

[78] R. Fraer, G. Kamhi, B. Ziv, M.Y. Vardi, and L. Fix. Prioritized traversal: Efficient reachability analysis for verification and falsification. In E.A. Emerson and A.P. Sistla, editors, *Proceedings of the 12th International Conference, CAV 2000 Chicago, IL, USA, July 15-19, 2000*, Berlin, 2000. Springer Verlag.

[79] N. Francez. *Fairness.* Springer Verlag, Berlin, 1986.

[80] M.C.W. Geilen. Real-time concepts for Software / Hardware Engineering. Master's thesis, Faculty of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands, 1996.

[81] M.C.W. Geilen. Formal models for encapsulation, structure and hierarchy in distributed systems. In J.P. Veen, editor, *Proceedings of the 10th Annual Workshop on Circuits, Systems and Signal Processing, Mierlo, The Netherlands, November 24-26, 1999*, pages 155–166, Utrecht, The Netherlands, 1999. STW / IEEE Technology Foundation.

[82] M.C.W. Geilen. On the construction of monitors for temporal logic properties. In K. Havelund and G. Rosu, editors, *Proceedings of RV'01 - First Workshop on Runtime Verification. Satellite Workshop of CAV'01. July 23, 2001 Paris, France. Electronic Notes in Theoretical Computer Science 55(2)*, Amsterdam, 2001. Elsevier Science.

[83] M.C.W. Geilen and D.R. Dams. An on-the-fly tableau construction for a real-time temporal logic. In M. Joseph, editor, *Proceedings of the Sixth International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT2000, 20-22 September 2000 Pune, India, LNCS 1926*, pages 276–290, Berlin, 2000. Springer Verlag.

[84] M.C.W. Geilen, D.R. Dams, and J.P.M. Voeten. Applying verification methods to non-exhaustive verification of Software / Hardware systems. In J. Veen, editor, *Proceedings of CSSP-98, 9th Annual ProRISC/IEEE Workshop on Circuits, Systems and Signal Processing Mierlo, Netherlands, November 25-27, 1998*, pages 177–183, Utrecht, The Netherlands, 1998. STW, Technology Foundation.

[85] M.C.W. Geilen and P.H.A. van der Putten. Brief manual for the SHESim, the modelling and simulation tool for the SHE methodology, 2000. Available via www: `http://www.ics.ele.tue.nl/~mgeilen/shesim`.

[86] M.C.W. Geilen and J.P.M. Voeten. Real-time concepts for a formal specification language for software / hardware systems. In *Proceedings of ProRISC 1997*, Utrecht, 1997. STW, Technology Foundation.

[87] M.C.W. Geilen and J.P.M. Voeten. Object-oriented modelling and specification using SHE. In D. Bošnački, S. Mauw, and T. Willemse, editors, *Proceedings of the First International Symposium on Visual Formal Methods VFM'99*, pages 16–24. Computing Science Reports 99 / 08 Department of Mathematics and Computer Science, Eindhoven University of Technology, 1999.

[88] M.C.W. Geilen, J.P.M. Voeten, P.H.A. van der Putten, L.J. van Bokhoven, and M.P.J. Stevens. Object-oriented modelling and specification using SHE. *Journal of Computer Languages*, 27(1-3):19–38, April-October 2001.

[89] R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. IFIP/WG6.1 Symp. Protocol Specification Testing and Verification (PSTV95), Warsaw Poland*, pages 3–18. Chapman & Hall, June 1995.

[90] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032 of *LNCS*. Springer Verlag, Berlin, 1996.

[91] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, 1997.

[92] P. Godefroid, G.J. Holzmann, and D. Pirottin. State-space caching revisited. *Formal Methods and System Design*, 7(3):1–15, November 1995.

[93] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, Reading, Massachusetts, 1989.

[94] M. Gordon. HOL: a proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis*. Kluwer, 1987.

[95] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Amsterdam, 1996.

[96] J.F. Groote. The syntax and semantics of timed μCRL. Technical Report SEN-R9709, Centrum voor Wiskunde en Informatica, 1997.

[97] J.F. Groote and M.A. Reniers. Algebraic pocess verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 17, pages 1151–1208. Elsevier Science, Amsterdam, 2001.

[98] H. Hansson and B. Jonsson. A calculus for communicating systems with time and probabilities. In *Proceedings of 11th Real-Time Systems Symposium, 1990*, pages 278–287. IEEE, 1990.

[99] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[100] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. Wiley, Chichester, 1990.

[101] T. Henzinger. It's about time: Real-time logics reviewed. In D. Sangiorgi and R. de Simone, editors, *Proceedings of CONCUR '98: Concurrency Theory, 9th International Conference, Nice, France, September 8-11, 1998, LNCS 1466*, pages 439–454, Berlin, 1998. Springer Verlag.

[102] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[103] C. Ho-Stuart, H.S.M. Zedan, and M. Fang. Congruent weak bisimulation with dense real-time. *Information Processing Letters*, 46(2):55–61, 1993.

[104] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[105] D.W. Hoffmann, J. Ruf, T. Kropf, and W. Rosenstiel. Simulation meets verification - checking temporal properties in SystemC. In F. Vajda, editor, *Proceedings of the 26th EUROMICRO Conference - Volume I, Maastricht, the Netherlands, Sept 5-7, 2000*, pages 435–438, Los Alamitos, California, 2000. IEEE Computer Society.

[106] U. Holmer, K.G. Larsen, and W. Yi. Deciding properties of regular real-timed processes. In K.G. Larsen and A. Skou, editors, *Proc. CAV'91 Computer Aided Verification 3rd Int. Workshop, Aalborg Denmark 1-4 July 1991*, pages 443–453. Springer Verlag, 1992.

[107] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[108] T. Hune, K.G. Larsen, and P. Pettersson. Guided synthesis of control programs using UPPAAL. *Nordic Journal of Computing*, 8(1):43–64, 2001.

[109] A. Ingólfsdóttir and H. Lin. A symbolic approach to value-passing processes. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 7, pages 427–478. Elsevier Science, Amsterdam, 2001.

[110] ITU-T. *Recommendation G.841 - Types and Characteristics of SDH Network Protection Architectures*. ITU, 1998.

[111] ITU-TS. ITU-TS Recommendation Z.120: Message Sequence Chart 1996 (MSC96). Technical report, ITU-TS, Geneva, 1996.

[112] G.L.J.M. Janssen. *Logics for Digital Circuit Verification*. PhD thesis, Eindhoven University of Technology, February 1999.

[113] W. Janssen, R. Mateescu, S. Mauw, P. Fennema, and P. van der Stappen. Model checking for managers. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking: Proceedings of the 5th and 6th International SPIN Workshops, 1999, LNCS 1680*, pages 92–107, Berlin, 1999. Springer Verlag.

[114] H.E. Jensen. *Abstraction-Based Verification of Distributed Systems*. PhD thesis, Department of Computer Science, Aalborg University, 1999.

[115] J.L.Fiadeiro and T.Maibaum. Verifying for reuse: Foundations of object-oriented system verification. In C.Hankin, I.Makie, and R.Nagarajan, editors, *Theory and Formal Methods 1994*. World Scientific Publishing Company, 1994.

[116] M. Kantrowitz and L.M. Noack. I'm done simulating; now what? Verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor. In *Proceedings of the 33rd Annual Conference on Design Automation Conference June 3 - 7, 1996, Las Vegas, NV USA*, pages 325–330, 1996.

[117] C. Kern and M.R. Greenstreet. Formal verification in hardware design: A survey. *ACM Transactions on Design Automation of Electronic Systems*, 4(2):123–193, april 1999.

[118] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In *Proceedings of the 5th Conference on Computer Aided Verification, Lecture Notes in Computer Science 697*, pages 97–109, Berlin, 1993. Springer Verlag.

[119] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2:176–201, 1993.

[120] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-time Systems*, 2:255–299, 1990.

[121] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[122] R. Kuiper and W. Penczek. *Lecture Notes on Modal and Temporal Logic in Computer Science*. Faculty of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, 1998.

[123] O. Kupferman and M.Y. Vardi. Model checking of safety properties. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification: 11th International Conference Proceedings, CAV'99, Trento, Italy, July 6-10, 1999 (LNCS 1633)*, pages 172–183. Springer Verlag, 1999.

[124] L. Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 657–668. Elsevier Science Publishers B.V. (Noth-Holland), 1983.

[125] L. Lamport. Logical foundation. In *Distributed Systems - Methods and Tools for Specification, LNCS 190*. Springer Verlag, 1985.

[126] F. Laroussinie, K.G. Larsen, and C. Weise. From timed automata to logic - and back. In J. Wiedermann and P. Hájek, editors, *Proceedings of Mathematical Foundations of Computer Science 1995, 20th International Symposium, MFCS'95, Prague, Czech Republic, August 28 - September 1, 1995, LNCS, Vol. 969*, pages 529–539, Berlin, 1995. Springer Verlag.

[127] K.G. Larsen, P. Pettersson, and W. Yi. Diagnostic model checking for real-time systems. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Proceedings of Hybrid Systems III: Verification and Control, DIMACS/SYCON Workshop, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA. LNCS, Vol. 1066*, pages 575–586, Berlin, 1995. Springer Verlag.

[128] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[129] K.G. Larsen and W. Yi. Time abstracted bisimulation: Implicit specifications and decidability. *Information and Communication*, 134:75–103, 1997.

[130] P. Liggesmeyer, M. Rothfelder, M. Rettelbach, and T. Ackerman. Qualitätssicherung Software-basierter technischer Systeme - Problembereiche und Lösungsansätze. *Informatik-Spektrum*, 21(5):249–258, 1998.

[131] L.M.B. Lopes. *On the Design and Implementation of a Virtual Machine for Process Calculi*. PhD thesis, University of Porto, 1999.

[132] G. Lopez. *Modélisation, Simulation et Vérification du Protocole APS*. Faculty of Electrical Engineering, Eindhoven University of Technology, Eindhoven, 1998.

[133] M. Maidl. The common fragment of CTL and LTL. In *Proceedings of 41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 643–652, Los Alamitos, California, 2000. IEEE Computer Society Press.

[134] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, New York, 1992.

[135] S. Mauw and M.A. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal*, 37(4):269–277, 1994.

[136] S. Mauw and G.J. Veltink, editors. *Algebraic Specification of Communication Protocols*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1993.

[137] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, 1993.

[138] M. Mihail and C.H. Papadimitriou. On the random walk method for protocol testing. In D. L. Dill, editor, *Proc. Of 6th International Conference on Computer Aided Verification LNCS 818*, pages 132–141. Springer Verlag, 1994.

[139] G.J. Milne. CIRCAL and the representation of communication, concurrency and time. *ACM Transactions on Programming Languages and Systems*, 7(2):270–298, april 1985.

[140] R. Milner. *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[141] S. Miyano and T. Hayashi. Alternating automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984.

[142] A. Mokkedem and D. Méry. A stuttering closed temporal logic for modular reasoning about concurrent programs. In D.M. Gabbay and H.J. Ohlbach, editors, *Proc. Of the First Int. Conf. On Temporal Logic (ICTL'94), LNAI 827*, pages 382–397. Springer Verlag, 1994.

[143] F. Moller and C. Tofts. A temporal calculus of communicating systems. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR'90 Theories of Concurrency: Unification and Extension Proc. Amsterdam, The Netherlands 27-30 Aug. 1990, Lecture Notes in Computer Science V.458*, pages 401–415, Berlin, 1990. Springer Verlag.

[144] M. Mukund and M. Nielsen. CCS, locations and asynchronous transition systems. In R.K. Shyamasundar, editor, *Proceedings of Foundations of Software Technology and Theoretical Computer Science, 12th Conference, New Delhi, India, December 18-20, 1992, LNCS, Vol. 652*, pages 328–341, Berlin, 1992. Springer Verlag.

[145] D. Murphy. Observing located concurrency. In A.M. Borzyszkowski and S. Sokolowski, editors, *Proceedings of Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 - September 3, 1993, LNCS, Vol. 711*, pages 566–576, Berlin, 1993. Springer Verlag.

[146] R. de Nicola and F.W. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Proceedings of Semantics of Systems of Concurrent Processes, LNCS Vol. 469*, pages 407–419, Berlin, 1990. Springer Verlag.

[147] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In K. Larsen and A. Skou, editors, *Proc. CAV'91 3rd International Workshop Computer Aided Verification, Ålborg, Denmark, July 1991 (LNCS 575)*, pages 376–398, Berlin, 1992. Springer Verlag.

[148] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE Transactions on Software Engineering*, 18(9):794–804, September 1992.

[149] M. Nielsen. CCS and its relationship to net theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, September 1986, LNCS, Vol. 255*, pages 393–415, Berlin, 1987. Springer Verlag.

[150] T. Nipkow and L.C. Paulson. Isabelle-91. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction, Saratoga Springs, NY, LNAI 607*, pages 673–676. Springer Verlag, 1992.

[151] Telecommunication Standardization Sector of ITU. *CCITT Recommendation Z.100: Specification and Description Language (SDL).* International Telecommunication Union, General Secretariat - Sales Section, Places des Nation, CH-1211 Geneva 20, 1988.

[152] S. Owre, J. Rushby, and N. Shankar. PVS: a prototype verification system. In D. Kapur, editor, *Proc. 11th International Conference on Automated Deduction*

(CADE) Lecture Notes in Artificial Intelligence Vol. 607, pages 748–752. Springer Verlag, June 1992.

[153] D.M.R. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proc. 5th GI Conference LNCS Vol. 104*, pages 167–183, Berlin, 1981. Springer Verlag.

[154] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Århus University, Computer Science Department, Århus, Denmark, 1981.

[155] A. Pnueli. The temporal logic of programs. In *Proc. Of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.

[156] P.H.A. van der Putten and J.P.M. Voeten. *Specification of Reactive Hardware / Software Systems*. PhD thesis, Eindhoven University of Technology, Department of Electrical Engineering, 1997.

[157] Y.S. Ramakrishna, P.M. Melliar-Smith, L.E. Moser, L.K. Dillon, and G. Kutty. Interval logics and their decision procedures part II: a real-time interval logic. *Theoretical Computer Science*, 170(1-2):1–46, december 1996.

[158] J.-F. Raskin and P.-Y. Schobbens. State clock logic: A decidable real-time logic. In *Proc. Hybrid and Real-Time Systems HART'97*, pages 33–47, Berlin, 1997. Springer.

[159] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[160] E.A.J. Reuter. Specification of a distributed real-time reactive control system, using Software / Hardware Engineering method. Master's thesis, Faculty of Electrical Engineering, Eindhoven University of Technology, 1997.

[161] I. Satoh and M. Tokoro. A formalism for real-time concurrent object-oriented computing. In *Proceedings of OOPSLA'92*, pages 315–326, 1992.

[162] R. Schlör and W. Damm. Specification and verification of system-level hardware designs using timing diagrams. In *Proc. European Conference on Design Automation Paris, France, February 22-25, 1993*, pages 518–524. IEEE Computer Society Press, 1993.

[163] S. Schwiderski, T. Hartmann, and G. Saake. Monitoring temporal preconditions in a behaviour oriented object model. *Data & Knowledge Engineering*, 14(2):143–186, 1994.

[164] A.P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6:495–511, 1994.

[165] S. Somé and R. Dssouli. From scenarios to timed automata: Building specifications from users requirements. In *Proceedings of Asia Pacific Software Engineering Conference, Pages 48-57, 1995*, pages 48–57, Los Alamitos, 1995. IEEE Computer Society Press.

[166] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000*, pages 248–263, Berlin, 2000. Springer.

[167] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 1992.

[168] H. Tauriainen. A randomized testbench for algorithms translating linear temporal logic formulae into Büchi automata. In *Proceedings of the Workshop Concurrency, Specification and Programming 1999 (CS&P'99)*, pages 251–262. Warsaw University, September 1999.

[169] J.N. van Tetrode. Implementing POOSL in C++. Master's thesis, Eindhoven University of Technology, Faculty of Electrical Engineering, 1997.

[170] B.D. Theelen. Towards modelling optical WDM transport networks. Master's thesis, Faculty of Electrical Engineering, Eindhoven University of Technology, 1999.

[171] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science Vol.*, pages 133–191. Elsevier Science Publishers (North Holland), 1990.

[172] H. Toetenel, R.L. Spelberg, S. Stuurman, and J. van Katwijk. Modeling and analysis of complex computer systems - the MTCCS approach. In *Proceedings of the Second IEEE International Conference on Engineering of Complex Computer Systems, 1996*, pages 423–430, Los Alamitos, 1996. IEEE Computer Society Press.

[173] S. Tripakis and C. Courcoubetis. Extending Promela and Spin for real-time. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings. LNCS 1055.*, pages 329–348, Berlin, 1996. Springer.

[174] I. Ulidowski and I. Phillips. Formats of ordered SOS rules with silent actions. In M. Bidoit and M. Dauchet, editors, *Proceedings of the 7th International Conference on Theory and Practice of Software Development TAPSOFT'97, Lille, France, LNCS 1214*. Springer Verlag, 1997.

[175] I. Ulidowski and S. Yuen. Extending process languages with time. In M. Johnson, editor, *Proceedings of the 6th International Conference on Algebraic Methodology and Software Technology AMAST'97 Sydney, Australia, LNCS 1349*. Springer Verlag, 1997.

[176] UML: Unified modelling language. rational software corporation, 1997. Available via www: `http://www.rational.com/uml`.

[177] M.Y. Vardi. Alternating automata: Checking truth and validity for temporal logics. In W. McCune, editor, *Proc. 14th Int. Conf. On Automated Deduction, Townsville Australia*, pages 191–206. Springer Verlag, 1997.

[178] M.Y. Vardi and P. Wolper. An Automata-Theoretic approach to automatic program verification. In *Proc. Of Logic in Computing Science*, 1986.

[179] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115:1–37, 1994.

[180] J.P.M. Voeten, M.C.W. Geilen, L.J. van Bokhoven, P.H.A. van der Putten, and M.P.J. Stevens. A probabilistic real-time calculus for performance evaluation. In G. Horton, D. Möller, and U. Rüde, editors, *Proceedings of the 11th European Simulation Symposium 1999, Erlangen, Germany*, pages 608–617, Delft, The Netherlands, 1999. SCS.

[181] C.H. West. Protocol validation in complex systems. *Computer Communication Review*, 19(4):303–312, 1989.

[182] F.N. van Wijk. A POOSL model of the MASCARA steady state control. Master's thesis, Faculty of Electrical Engineering, Eindhoven University of Technology, 1999.

[183] G. Winskel. *The Formal Semantics of Programming Languages. An Introduction*. Foundations of Computing. The MIT Press, Cambridge, Massachusetts, 1993.

[184] P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In *Proceedings of 24th IEEE Symposium on Foundation of Computer Science, Tuscan*, pages 185–194, 1983.

[185] C.H. Yang and D.L. Dill. Validation with guided search of the state space. In *Proceedings of the 35th Conference on Design Automation, Moscone Center, San Francisco, California, USA, June 15-19, 1998*, pages 599–604. ACM Press, 1998.

[186] W. Yi. Real-time behaviour of asynchronous agents. In *Proceedings of CONCUR90 (International Conference on Concurrency Theory), Amsterdam, LNCS 458*. Springer Verlag, 1990.

List of Publications

- M.C.W. Geilen. On the construction of monitors for temporal logic properties. *In K. Havelund and G. Rosu, editors,* Proceedings of RV'01 - First Workshop on Runtime Verification. Satellite Workshop of CAV'01. July 23, 2001 Paris, France. Electronic Notes in Theoretical Computer Science 55(2) , *Amsterdam, 2001. Elsevier Science.*

- M.C.W. Geilen, J.P.M. Voeten, P.H.A. van der Putten, L.J. van Bokhoven, and M.P.J. Stevens. Object-oriented modelling and specification using SHE. *Journal of Computer Languages 27(1-3),* pages 19–38, Amsterdam, 2001. Elsevier Science.

- M.C.W. Geilen. Non-exhaustive model-checking in component based systems. *Journal of Systems Architecture : the EUROMICRO Journal* (to be published).

- M.C.W. Geilen and D.R. Dams. An on-the-fly tableau construction for a real-time temporal logic. In M. Joseph, editor, *Proceedings of the Sixth International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT2000, 20-22 September 2000 Pune, India, LNCS 1926,* pages 276–290, Berlin, 2000. Springer Verlag.

- M.C.W. Geilen. Model-checking in simulations of distributed Systems. In D.P.F. Möller, editor, *Proceedings of 12th European Simulation Symposium ESS 2000, Hamburg, Germany Sept. 28-30, 2000,* pages 606–611, 2000, Society for Computer Simulation International.

- M.C.W. Geilen and J.P.M. Voeten. Object-oriented modelling and specification using SHE. In R.C. Backhouse and J.C.M. Baeten, editors, *Proceedings of the First International Symposium on Visual Formal Methods VFM'99,* pages 16–24. Computing Science Reports 99/08 Department of Mathematics and Computer Science, Eindhoven University of Technology, 1999.

- M.C.W. Geilen. Formal models for encapsulation, structure and hierarchy in distributed systems. In J.P. Veen, editor, *Proceedings of the 10th Annual Workshop on Circuits, Systems and Signal Processing, Mierlo, The Netherlands, November 24-26, 1999,* pages 155–166, Utrecht, The Netherlands, 1999. STW/IEEE Technology Foundation.

- M.C.W. Geilen, D.R. Dams, and J.P.M. Voeten. Applying verification methods to non-exhaustive verification of software/hardware systems. In J. Veen, editor, *Proceedings of CSSP-98, 9th Annual ProRISC/IEEE Workshop on Circuits,*

Systems and Signal Processing Mierlo, Netherlands, November 25-27, 1998, pages 177–183, Utrecht, The Netherlands, 1998. STW, Technology Foundation.

- M.C.W. Geilen and J.P.M. Voeten. Real-time concepts for a formal specification language for software / hardware systems. In *Proceedings of ProRISC 1997*, Utrecht, 1997. STW, Technology Foundation.

Curriculum Vitae

Marc Geilen was born on October 24th, 1972 in Sittard, the Netherlands. In 1991 he graduated from the Serviam Lyceum in Sittard for the gymnasium-β program. From 1991 he was a student in Information Technology at the Faculty of Electrical Engineering, Eindhoven University of Technology, where he received his degree with honours in 1996. From 1996 to 2000 he has worked as a Ph.D. student at the Information and Communication Systems Group at the same department.
In the period from September 2000 to January 2002, he held a position as an assistant professor in the Information and Communication Systems group in the field of formal verification of reactive systems. As of February 2002 he is active as a researcher in the European IST project Ozone, "New Technologies and Services for Emerging Nomadic Societies", where he focusses on programming paradigms for multiprocessor systems. His research interests include validation and (formal) verification, real-time systems, system-level modelling, simulation and programming paradigms for concurrent systems.

Index