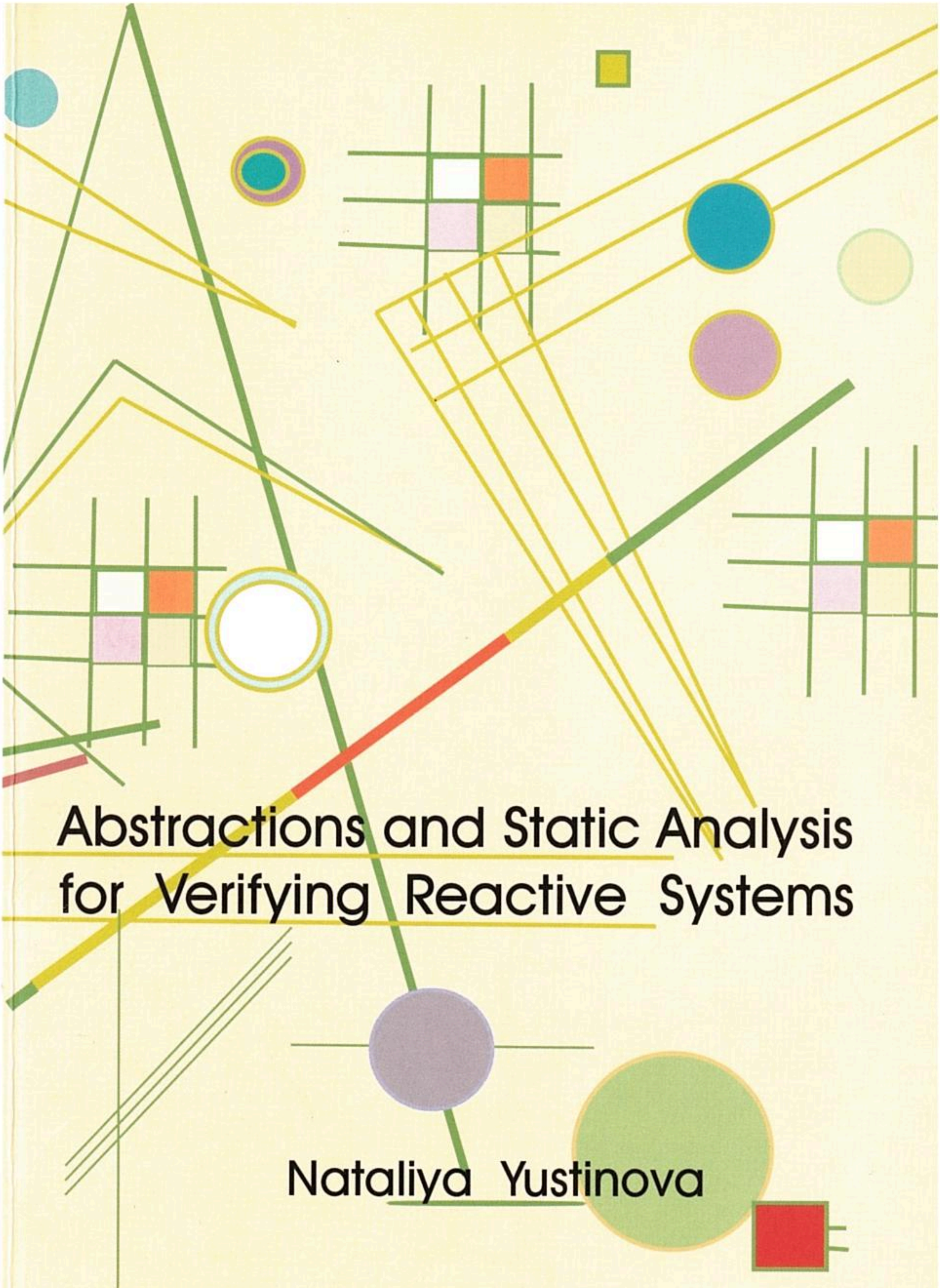

Abstractions and Static Analysis for Verifying Reactive Systems

Nataliya Yustinova

Abstractions and Static Analysis for Verifying Reactive Systems

Nataliya Yustinova

Centrum voor Wiskunde en Informatica

The work reported in this thesis has been carried out at the CWI (Centrum voor Wiskunde en Informatica) within the SVC (Systems Validation Centre) project funded by Telematics Institute and the KTVFM project funded by the Dutch Ministry of Defence.

Nataliya Yustinova
Abstractions and Static Analysis for Verifying Reactive Systems /
by Nataliya Yustinova. - Amsterdam: CWI, 2004.
Proefschrift. - ISBN 906196 525 X

Subject headings: formal methods / software verification / model checking / static analysis / abstraction / reactive systems

VRIJE UNIVERSITEIT

Abstractions and Static Analysis for Verifying Reactive Systems

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr.T. Sminia,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op donderdag 4 november 2004 om 10.45 uur
in het auditorium van de universiteit,
De Boelelaan 1105

door

Nataliya Yustinova

geboren te Yaroslavl, Rusland

promotor:	prof.dr. W.J. Fokkink
copromotor:	dr. N. Sidorova

Acknowledgments

The work presented in this thesis has been started in the last year I was employed as a teaching assistant at the Chair of Information and Communication Services, Rostock University, Germany. A major part of this work was done during the three years I was working at the SEN2 (Specification and Analysis of Embedded Systems) group of the Department of Software Engineering, CWI (Centrum voor Wiskunde en Informatica), within the SVC (Systems Validation Centre) project funded by Telematics Institute and the KTVFM project funded by the Dutch Ministry of Defence.

There are many people who helped me to finish this thesis and whom I would like to thank. I am grateful to my promotor Wan Fokkink for his confidence in my ability to succeed, for his advice, constructive criticism and support at important moments. He carefully read all parts of this work and provided feedback that helped me to look at the results of my work from a broader perspective and to reach a significant progress in presenting the results.

I deeply appreciate the help of my co-promotor Natalia Sidorova who guided me in my everyday research life. She encouraged and supported me by sharing with me scientific interests and by being patient explaining me complicated details. I am grateful for the time she spent reading and commenting on this thesis. I have learned a lot from her about planning research, writing scientific articles and giving presentations. Her optimism and good sense of humour kept me going in moments of uncertainty and helped me to finish this work.

My sincere thanks to my other co-authors Martin Steffen, Dragan Bošnački and Stefan Blom. I really enjoyed working with Martin. He never stopped asking critical questions until he was satisfied with the quality of the achieved results. I appreciate the time that Dragan spent helping me to get into subtle implementation details of *Spin* and *DTSpin*, and reading this work as a member of the reading committee. His comments helped me to improve the readability of this thesis. The competent support of Stefan facilitated my work with the μCRL toolset and verification framework. Thanks for our discussions and scientific quarrels.

I am grateful to Walter Vogler, Jaco van de Pol and Jan Willem Klop for their consent to be members of the reading committee. Their comments and remarks contributed to the quality of this thesis. I also thank Kees van Hee and Jan Bergstra who agreed to be members of the promotion committee.

Leon Wolters and Michail Petreczky spent uncounted hours in proof-reading this thesis.

I also owe my success to my former "bosses" Valery Sokolov and Clemens Cap who persuaded me to go on with my scientific career.

In Rostock, I have really enjoyed the journeys to Rügen and playing squash with my colleagues Stephan Preuss, Mykhailo Lyubich, Nico Maibaum and

Igor Sedov. Enormous thanks to my flatmates and friends Nadege Spella, Nina Kitzig, Helge Haufe, Katrin Zansinger, Karsten Kaika, Anja Gellert and Donald Reebs with whom I spent a lot of nice evenings.

In Amsterdam, I very much enjoyed the company of Simona Orzan and Daniel Benden who showed me the nightlife of this wonderful town. Thanks to Vincent van Oostrom for scientific hints. I am grateful to Paul Klint for true interest in the progress of my work. Thanks to Joost Visser, Alban Ponse and Engelbert Hubbers who helped me with learning Dutch.

The most heartfelt thanks I want to express to my parents who encouraged my intellectual curiosity and gave me the opportunity to make my own choices and mistakes. I always trusted in their help and support.

Table of Contents

1

Introduction

1.1 Towards Reliable Reactive Systems

In the last decades, the application domain of reactive systems has drastically increased. Nowadays, reactive systems are used in various areas, from avionics and automotive systems to telecommunication and manufacturing systems. *Reactive systems* are the systems whose role is to maintain an ongoing interaction with their environment, rather than produce a final value on termination. A typical reactive system exhibits the following distinctive characteristics [125, 84]:

- It *continuously interacts with its environment*, using inputs and outputs.
- The inputs and outputs are often *asynchronous*, meaning they may arrive and change values at any point of time.
- Its operation and reaction to inputs often reflects strict *time* requirements.
- It has many possible operational scenarios, depending on the current mode of operation and on the current values of its *data* as well as its past behavior.
- Often, it is not expected to terminate.
- In general, it consists of many interacting processes that operate in *parallel*.

Typical examples of reactive systems are on-line interactive systems, such as flight reservation systems; traffic control systems; systems controlling mechanical and electronic devices in a train or a plane; systems controlling ongoing processes in a nuclear reactor.

Behaviour of reactive systems is usually very complex. It cannot be described in an unambiguous, clear and concise way by giving a verbal description. Verbal descriptions tend to be lengthy, incomplete and usually not well-structured. They contain phrases that can be misinterpreted and implemented in many different ways. Therefore, the need for formal description techniques was realized a long time ago.

In the second half of the 1970s, ISO (International Organization for Standardization) started to work on *formal description techniques* (FDTs) that allow to *specify* reactive systems. After eight years of work, the outcome was standards for Estelle (Extended Finite State Machine Language), LOTOS (Language of Temporal Ordering Specification) and SDL (Specification and Description Language) [161]. The last one is the subject of particular attention in this thesis.

Nowadays SDL is a modern, high-level specification language suitable for the description of complex event-driven real-time communicating systems. SDL provides concepts for the specification of behaviour covering asynchronous communication and parallelism. It also allows to express qualitative and quantitative time requirements for reactive systems. SDL concepts used for describing system behaviour and communication were integrated into those of UML (e.g. SDL process diagrams correspond to UML state diagrams; SDL communication links correspond to UML associations [141]).

In the telecommunication field, SDL is the language of choice for the development of a broad range of software and hardware. Examples are 3G products, cellular phones, switches, WAP stacks, Bluetooth devices, GPRS systems, DECT phones, radio systems, network management platforms and network services systems. Other examples are telecommunication standards like UMTS, GSM, ISDN, V5.2, INAP, etc. SDL is also used in factory automation systems, aerospace and automotive applications, and other safety-critical systems, e.g. kidney-dialysis devices and train-control systems. SDL is used by standardization organizations, universities and companies all over the world (e.g. Alcatel, Ericsson, Fujitsu, Hewlett-Packard, Lucent Technologies, Motorola, Nokia, Nortel, Siemens, BT, Deutsche Telekom and NTT) [157]. In this thesis, we chose an approach to the specification of reactive systems inspired by SDL.

There is a wide range of reactive systems where errors can have catastrophic consequences leading to loss of lives, serious environmental damage, failure of an important mission, or major economic loss. Here are just two examples of such errors.

Huge losses of monetary and intellectual investment were caused by a rocket boost failure in Ariane 5 in June, 1996 ([99]). Ariane 5, an expendable launch system, was designed by the European Space Agency (ESA) and manufactured, operated and marketed by Arianespace as part of the Ariane program. Ariane 5 software reused old code from Ariane 4 that was not respecified and retested in the new environment. Ariane 5 being more powerful than Ariane 4 caused an unanticipated floating-point exception that would never have occurred on Ariane 4. The exception was not caught. Direct cost of this failure was estimated at 0.5 billion EUR, indirect cost at 2 billion EUR.

Another example is a failure that caused a power shutdown of cruiser USS Yorktown in November 1998 ([99]). A crew member of the guided-missile cruiser USS Yorktown mistakenly entered zero for a data value, which resulted in a division by zero. The division by zero caused an arithmetic exception, which propagated through the system, crashed all LAN consoles and remote terminal units, and led to a power shutdown for about three hours.

The failure in Ariane 5 software and the power shutdown of cruiser USS Yorktown are just a few of many reactive system's failures with serious consequences. Therefore, the production of reliable reactive systems became a fundamental concern to computer scientists. Different techniques like static analysis, testing, and various formal methods were developed by academic and industrial communities to ensure the quality and reliability of reactive systems.

Static analysis [131] is a broad term covering a wide range of analysis techniques evaluating programs without executing them. Traditionally, static analysis was aimed at the optimization of programs. Typical applications include detecting *redundant* computations, e.g. loop invariants, and detecting *superfluous* computations that lead to results that are not used or results that are known already at compile-time. Static analysis is also applied for type checking, performance analysis and partial evaluation. Moreover, there are approaches integrating static analysis with formal methods. For example, static analysis is often used prior to model checking (cf. Section 1.2) to slice programs into smaller parts or to identify independent fragments of a program that can be executed in parallel.

Testing [172, 33] is still one of the most popular techniques that is used in industry to ensure the reliability of systems. In testing approaches, the system is simulated with certain inputs called stimuli, and the reaction of the system to stimuli is compared with the one defined by the requirement specification. Exhaustive testing covering all possible system scenarios is practically impossible, hence testing allows to gain confidence about the correctness of the system by looking at *some* of them but it cannot guarantee the absence of errors. This thesis does not deal with testing.

Formal methods allow to determine system correctness by formal proofs that cover *all* possible scenarios of the system. They not only help to find errors that are missed by testing but can also prove that the system meets its requirements. The usage of formal methods in early phases of system development makes possible early detection of errors, which greatly reduces the costs of their correction.

1.2 Formal Methods

The term *formal methods* covers different approaches to specification and verification based on mathematical formalisms. Formal methods are aimed at establishing system correctness with mathematical precision. Every formal approach to system verification usually involves a mathematical *model* of a system, a *formal language* to specify properties of the system, and a *method* to check whether the model satisfies the specification.

Labelled transition systems (LTSs) are a common basis for modelling the behaviour of reactive systems. LOTOS, Estelle and SDL share LTSs as the formal basis of their operational semantics. Therefore, we have chosen LTSs as mathematical model for reactive systems.

Various logics have been proposed to specify properties of systems, e.g. computation tree logic (*CTL* [62]), linear temporal logic (*LTL* [137]) and μ-calculus [112]. In this thesis, we use *LTL* and some subsets of μ-calculus to express properties of reactive systems.

Verifying (or checking) whether a model of a system satisfies certain properties may be partially or completely *automated*. Two well-established formal approaches to computer-aided verification are *theorem proving* and *model checking*.

Theorem Proving. In theorem proving, a system and its properties are expressed in terms of some mathematical logic. This logic is defined by a formal system that provides a set of axioms and a set of inference rules. Theorem proving consists in finding a proof of a property from the axioms of the system. Steps in the proof appeal to the axioms, rules, and, possibly, to derived definitions and intermediate lemmas. There are tools supporting machine-assisted theorem proving, e.g. PVS [134]. Theorem provers are increasingly being used today in the mechanical verification of safety-critical properties of hardware and software designs [43, 149].

Theorem proving can deal directly with infinite state spaces. It relies on techniques like structural induction to construct proofs over possibly infinite domains. Interactive theorem provers, by definition, require a *lot of interaction* with a human, so the theorem proving process is *slow and very expensive* [43]. In the process of finding the proof, however, the user often gains better insight into the system or the property being proved. Theorem proving is out of the scope of this thesis.

Model Checking. Model checking is considered as the method of choice in the verification of reactive systems and is increasingly accepted in industry for its push-button appeal. The term *model checking* designates a collection of formal techniques for the automatic analysis of reactive systems. Subtle errors in the design of reactive systems that often elude conventional verification techniques like testing can be and have been found in this way. Since model checking has been proved cost-effective and integrates well with conventional design methods, model checking has been accepted as a standard procedure to assure the quality of reactive systems [42, 43].

The input to a model checker is usually a finite-state description of the system to be analyzed and a number of properties that are expected to hold for the system, often expressed as formulas of some temporal logic. In contrast to theorem proving, model checking is completely *automatic* and relatively *fast*, often producing an answer in a matter of seconds. The model checker reports either that the property holds or that it is violated. In the latter case, it *provides a counterexample*, a run of the system that violates the property. Such a run can provide a valuable feedback and points to design errors. Model checking can be used to check partial specifications, and thus it can provide useful information about a system's correctness even if the system has not been completely specified.

There are two approaches to model checking: *symbolic* as in *NuSMV* [37] and *COSPAN* [83], and explicit state (or *enumerative*) as in *Spin* [93] and CADP [65]. In the *symbolic* approach, a finite-state system is encoded using a set of binary variables, just as ordinary data types of programming languages are represented in binary form on a computer. The transition relation is expressed as a propositional formula in terms of two sets of variables, one set encoding the old state and the other encoding the new one. Propositional formulas are then represented as binary decision diagrams (BDDs [32]). The model checking algorithm is based on computing fixpoints of predicate transformers that are obtained from the transition relation [124].

Although the symbolic approach can lead to spectacular results, it is not a panacea. Effectiveness of symbolic model checking depends on finding a "good" variable ordering for the representation of a BDD. However, finding a "good" variable ordering is very difficult [127].

State space *enumeration* methods consider each reachable state of a model (a finite-state system) to determine whether the model satisfies a given property.

The main obstacle in model checking of industrial reactive systems is the state explosion problem. The size of the model often grows exponentially with the number of system components working in parallel. The research questions considered in this thesis mainly deal with the development of techniques to cope with the state explosion problem in enumerative model checking.

A number of techniques have been developed to mitigate the state explosion problem: *abstraction* [122, 41, 50], *compositional verification* [148], *partial-order reduction* [74, 163, 95], and *on-the-fly* techniques [93].

Abstraction. The size of systems that can be analyzed by model checking directly remains rather limited. It is still far from the size of real reactive systems, which are often not only large but infinite. Therefore, model checking must be performed on abstract models.

Abstraction methodologies are concerned with the following question [48]: Given a concrete system and a property to be checked, how to get a suitable abstract system of finite (smaller) size? To answer this question, an abstraction framework must provide three things: a method for obtaining abstract models, a method for relating abstract and concrete models, and a logic stating properties so that properties satisfied on the abstract system can be related to properties of the concrete system. In general, when we say that a system T^{α} is an abstraction of a system T, we mean that the observable behaviour of T is contained in the observable behaviour of T^{α}.

Methods for obtaining abstract models range from slicing and variable hiding to more general algorithmic approaches like program transformation based on Abstract Interpretation [46]. The relationship between abstract and concrete models is usually defined in terms of bisimulation, homomorphism, Galois connection or simulation (cf. Chapter 2). Given a property of some logic, two types of preservation are considered: *weak preservation*, when every property that is *true* on the abstract model is also *true* on the concrete one, and *strong preservation*, when the same properties hold on the abstract and concrete models [121].

Compositional Verification. State explosion can be alleviated by decomposing a system into components and considering the components of the system one at a time. As in the case of abstraction, compositional verification requires additional input from the user who must specify appropriate properties of individual components. The components do not necessarily function properly in an arbitrary environment. Their behaviour relies on the properties of the rest of the system. Thus, corresponding assumptions have to be introduced in the statement of components' properties [122].

Partial-Order Reduction is aimed at reducing the size of a system by exploiting the commutativity of concurrently executed actions that result in the same state when executed in different orders. The effectiveness of partial-order reduction methods in general depends on the structure of the system: they are useless for tightly synchronized systems, while they may dramatically reduce the number of states and transitions explored during model checking of loosely coupled, asynchronous systems [42].

On-the-fly techniques allow to minimize the memory demands of model checkers by constructing only those parts of systems that are necessary to verify or refute a given property [93].

Verification of Timed Systems

Quantitative time aspects are often important for correct functionality of reactive systems. Various formalisms such as timed transition systems [89], timed automata [1] and logics [3] have been proposed to model them. When modelling time, two time domains, discrete and dense, are usually differentiated. In the case of a *dense time domain*, time is modelled by real numbers and time progression has a continuous nature. In the case of a *discrete time domain*, time is modelled by non-negative integers and time progresses by discrete steps.

Dense time often allows a more adequate representation of reality than discrete time but it also leads to verification algorithms with higher complexity. UPPAAL [119] is a leading toolset produced in the academic community for the verification of timed systems with dense time. Various verification options like bit-hashing, inactive clock reduction, compact memory management, counter-example generation, etc. are provided for the verification [15]. Designed for the analysis of timed aspects of reactive systems, UPPAAL is not aimed at the verification of the data aspects.

In [88], Henzinger, Manna and Pnueli showed that discrete time suffices for a large and important class of systems and properties, including all systems that can be modelled as timed transition systems, and such properties as time-bounded invariance and time-bounded response. In [29], the authors state that discrete-time automata can be analyzed using any representation scheme used for dense time, and *additionally* can benefit from enumerative and symbolic techniques, which are not naturally applicable to the systems with dense time.

In this thesis, we limit our attention to reactive systems with discrete time. *DTSpin* [24, 55], a discrete time extension of the *Spin* model checker, was used for the majority of experiments mentioned in this thesis.

1.3 Research Questions

In this section, we formulate the research questions that are considered in this thesis. All research questions are related to the state explosion problem, which can be caused by various factors like interpretation of time, data aspects, asynchronous communication, etc.

Modelling time aspects

Reactive systems are usually timed systems that must respond within certain time limits. Interpretation of time and time constraints in a specification language is very much affected by the intended mode of its use. In implementation-orientated languages, time is modelled as an infinitely growing variable of integer or real type. One infinitely growing variable immediately leads to an infinite system.

Timers are usually employed to express time constraints imposed on a reactive system. They can be used for several purposes: to control the release of a limited resource, to control answers from unreliable resources, to issue actions on a regular basis, etc. Timers are modelled as alarm clocks that either send a signal or throw an exception at the right moment of time. Timers are set to moments in time when they should expire. Since reactive systems are usually not supposed to terminate, settings of timers are unbounded. This interpretation, natural for implementation purposes, is, however, not the best choice for verification.

Taking SDL as an instance of the class of implementation-oriented languages, our objective is to provide an interpretation of time and timers that alleviates the state explosion problem and to show that systems with this interpretation can safely be used for verification.

Abstracting timers

Correctness of reactive systems often depends on right timer settings. Model checkers can only verify a single finite-state system at a time. Direct model checking whether a system works for all settings of a timer larger than or equal to some k would require one iteration for each setting larger than some k, i.e., we would need infinitely many iterations. In some cases, it would be more convenient to reason automatically about a family of similar systems. The verification problem in this case can be formulated as follows: given a family of systems whose timer settings satisfy condition "larger than or equal to some k" and a property, check whether the property holds for each system in the family. This problem is undecidable for model checking [7]. In some cases, an abstraction that treats settings of a timer as a system parameter can be used to solve this problem.

Abstractions, however, introduce infinite traces that do not correspond to any behaviour of the real system that can lead to false negatives. Our objective is to show how to exclude nuisance behaviour in case of a timer abstraction and how to do that in the most efficient way.

Closing open systems

Most model checkers cannot handle open systems. Therefore, the next step following the decomposition of a system into components is *closing* components with an environment.

Closing open systems is commonly done by adding an environment that is an abstraction of the real environment. The simplest safe abstraction of the environment thus behaves *chaotically*. When done manually, this closing, as simple as it is, is tiresome and error-prone for large systems, for instance due to the sheer amount of signals.

For model checking, the approach to closing should be well-considered, to counter the state explosion problem. This is especially true in the context of model checking reactive systems where components communicate *asynchronously*. Sending arbitrary message streams to unbounded input queues will immediately lead to an infinite system, unless some assumptions restricting the environment behaviour are incorporated in the closing process. Even so, external chaos results in a combinatorial explosion caused by all combinations of messages in the input queues.

Another problem addressed by closing is that the *data* carried with the messages coming from the environment is usually drawn from some infinite data domain. Special care must be taken to ensure that chaos also shows more behaviour with regards to *timing* issues such as timeouts and time progress. Our objective is to provide an automatic approach to closing asynchronous open timed systems.

Reuse of untimed verification methods for timed verification

Many formalisms have been proposed for timed verification. Most of them are designed for the analysis of timed aspects of reactive systems, while data aspects are usually not taken into account. On the other side, there exist powerful formalisms that are able to handle both data and behaviour aspects of reactive systems but are originally not aimed at the verification of time issues. Our challenge is to show how to reuse untimed formalisms with good support for data types and behaviours for the verification of reactive systems with discrete time.

1.4 Road Map

Chapter 2 reviews some mathematical notions and some notions from computer science that will be used in the rest of the thesis.

Chapter 3 presents a transformation of SDL timers aimed at reducing the infinite domain of timer values to a finite one. We justify the proposed transformation by proofs that allow us to transfer both negative and positive results of verification from the transformed model to the original one. We show that the transformed model and the original one are related by path equivalence up to stuttering. This guarantees that any $LTL{-}X$-formula satisfied by the transformed model is satisfied by the original one, and that a counterexample trace found in the transformed system can also be found in the original one.

In Chapter 4, we propose a timer abstraction and argue its correctness. The abstraction introduces infinite traces that have no corresponding traces at the concrete level. We show how to exclude them by imposing a strong fairness constraint on the abstract model. By employing the fact that the timer abstraction introduces a self-loop, we render the strong fairness constraint into a weak fairness constraint and embed it into the verification algorithm.

In Chapter 5, we propose an automatic transformation yielding a closed system. By *embedding* the outside chaos into the system, we avoid the state-space penalty in the input queues mentioned above. To capture the chaotic timing behaviour of the environment, we introduce a non-standard three-valued timer abstraction. The transformation is based on *data-flow analysis* that detects instances of chaotic variables and timers. The approach is implemented in a tool that automatically closes DTPROMELA translations of SDL-specifications. To corroborate the usefulness of our approach, we compare the state space of a system closed by embedding chaos with the state space of the same system closed with chaos as external environment process on a case study for a wireless ATM medium-access protocol.

In Chapter 6, we propose a manner of introducing discrete time into the μCRL language without extending the language. The specification language μCRL [78] (micro Common Representation Language) is a process algebraic language that was especially developed to take account of data in the study of communicating processes. The μCRL toolset [19] together with the CADP toolset [65] provides support for enumerative model checking. The semantics of discrete time we use makes it possible to reduce the *time progress* problem to the diagnostics of "no action is enabled" situations. The synchronous nature of μCRL facilitates this task. We show some experimental verification results obtained on a timed communication protocol.

Each of these chapters contains an introduction giving an elaborated motivation and an overview of related work.

Chapter 7 discusses how our research questions are answered in this thesis.

1.5 Origins of Chapters

Chapter 3, "Timer Transformation to Verify SDL Specifications", was co-authored with Natalia Sidorova. It was published earlier as:

N. Ioustinova and N. Sidorova. Transformation of SDL specifications - a step towards the verification. In D. Bjorner, M. Broy, and A. Zamulin, editors, *Post-proceedings of Andrei Ershov Fourth International Conference Perspectives of System Informatics (PSI 01)*, volume 2244 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2001.

Chapter 4, "Using Fairness to Make Abstractions Work", was co-authored with Dragan Bošnački and Natalia Sidorova. It was published earlier as:

D. Bošnački, N. Ioustinova, and N. Sidorova. Using fairness to make abstractions work. In S. Graf and L. Mounier, editors, *Proc. of the 11th Int. Spin Workshop on Model Checking of Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 198–215. Springer, 2004.

Chapter 5, "Closing and Flow Analysis for Model Checking Reactive Systems", was co-authored with Natalia Sidorova and Martin Steffen. It was published earlier as:

N. Ioustinova, N. Sidorova, and M. Steffen. Abstraction and flow analysis for model checking open asynchronous systems. In P. Strooper and P. Muenchaisri, editors, *Proc. of the 9th Asia Pacific Software Engineering Conference (APSEC 2002)*, pages 227–235. IEEE Computer Society, 2002.

N. Ioustinova, N. Sidorova, and M. Steffen. Closing open SDL-systems for model checking with *DTSpin*. In L. H. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods - Getting IT Right, Proc. of International Symposium of Formal Methods Europe, FME 2002*, volume 2391 of *Lecture Notes in Computer Science*, pages 531–548. Springer, 2002.

N. Ioustinova, N. Sidorova, and M. Steffen. Synchronous closing and flow abstraction for model checking timed systems. In *Proc. of the Second International Symposium on Formal Methods for Components and Objects (FMCO'03)*, volume (to appear) of *Lecture Notes in Computer Science*. Springer, 2004.

Chapter 6, "Timed Verification with μCRL", was co-authored with Stefan Blom and Natalia Sidorova. It was published earlier as:

S. Blom, N. Ioustinova, and N. Sidorova. Timed verification with μCRL. In M. Broy and A. Zamulin, editors, *Proc. of the 5th Int. Conf. Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2003.

2

Preliminaries

This chapter reviews some mathematical notions and some notions from computer science that will be used in the rest of the thesis.

2.1 Partially Ordered Sets and Lattices

The notions of a partially ordered set, a complete lattice and fixed points play a crucial role in static analysis. Here we review basic definitions and some results about partial orders, lattices, least and greatest fixed points [53].

Definition 2.1. [PARTIAL ORDER]
Let S be a set. An order *(or* partial order*) on S is a binary relation $\sqsubseteq$ on S such that for all $s, s_1, s_2, s_3 \in S$,*

– $s \sqsubseteq s$,
– $s_1 \sqsubseteq s_2 \wedge s_2 \sqsubseteq s_1 \Rightarrow s_1 = s_2$,
– $s_1 \sqsubseteq s_2 \wedge s_2 \sqsubseteq s_3 \Rightarrow s_1 \sqsubseteq s_3$.

These conditions are referred to as *reflexivity*, *antisymmetry* and *transitivity*, respectively. A set S equipped with an order relation $\sqsubseteq$ is called an *ordered set* (or *partially ordered set*) denoted $(S; \sqsubseteq)$. Further we use the shorthand *poset*.

Many important posets are expressed in terms of existence of certain upper and lower bounds of subsets of S. The most important classes of posets defined in this way are lattices and complete lattices. $s \in S$ is the *least element* of S if $s \sqsubseteq s'$ for all $s' \in S$. The *greatest element* of S is defined dually. A subset S' of S has $s \in S$ as an *upper bound* if for all $s' \in S'$: $s' \sqsubseteq s$. A subset S' of S has $s \in S$ as an *lower bound* if for all $s' \in S'$: $s \sqsubseteq s'$. A *least upper bound* s of S' is an upper bound of S' such that $s \sqsubseteq s''$ for all upper bounds s'' of S'. A *greatest lower bound* s of S' is a lower bound of S' such that $s'' \sqsubseteq s$ for all lower bounds s'' of S'.

Definition 2.2. [COMPLETE LATTICE]
A complete lattice *is a poset S such that all its subsets have a least upper bound and a greatest lower bound.*

Further, greatest lower bound and least upper bound of S, when they exist, are denoted as $lub(S)$ and $glb(S)$, respectively.

Definition 2.3. [FIXPOINT]
Let S be a poset and $f\colon S \to S$ be a function. We say $s \in S$ is a fixpoint *of f if $f(s) = s$. The set of all fixpoints of f is denoted $fix(f)$. The least element of $fix(f)$, when it exists, is called the* least fixpoint *of f. Similarly we define the* greatest fixpoint *of f.*

Let $(S, \sqsubseteq)$ be a complete lattice and $f\colon S \to S$ be a function. We say that f is *monotonic* if $f(s) \sqsubseteq f(s')$ whenever $s \sqsubseteq s'$.

Theorem 2.1. [KNASTER/TARSKI]
Let $(S, \sqsubseteq)$ be a complete lattice and $f\colon S \to S$ be a monotonic function. Then f has a greatest fixpoint $gfp(f)$ and a least fixpoint $lfp(f)$.

Definition 2.4. [GALOIS CONNECTION]
Let $(S; \sqsubseteq)$ and $(A; \preceq)$ be posets. A pair of mappings $\alpha : S \longrightarrow A$, $\gamma : A \longrightarrow S$ is a Galois connection *(α, γ) from S to A iff for all $s \in S$ and $a \in A$, $\alpha(s) \preceq a \Leftrightarrow s \sqsubseteq \gamma(a)$.*

2.2 Transition Systems and Behavioural Equivalences

Definition 2.5. [TRANSITION SYSTEM]
A transition system *T is a tuple (S, R) where S is a set of* states *and $R \subseteq S \times S$ is a* transition relation.

A transition system can have various attributes. Often a subset $S_0 \subseteq S$ is designated to represent the initial states. A transition system may come with an *interpretation function* $\mathcal{I}\colon \mathcal{P} \to 2^S$ that specifies interpretation of atomic propositions from $\mathcal{P}$ over the states (see Section 2.3). Alternatively, valuation of literals in states may be given by a *labelling function* $\mathcal{L}\colon S \to 2^{\mathcal{P}}$ specifying the propositions that hold in a state. A transition system with initial states and an interpretation function $\mathcal{L}$ is also called *Kripke structure* [113] (see Section 2.3). Not only states but also transitions of the system can be labelled.

Definition 2.6. [LTS]
A labelled transition system (LTS) *T is a tuple $(S, Lab, \to, s_0)$ where*

- *S is a set of* states *or* locations*;*
- *Lab is a set of* labels*;*
- *$\to \subseteq S \times Lab \times S$ is a labelled* transition relation*;*
- *$s_0 \in S$ is an* initial state.

Further we write $s \to_\lambda s'$ for a triple $(s,\ \lambda,\ s') \in\to$. A triple $(s,\ \lambda,\ s')$ is also referred to as a λ-*step* of T.

Definition 2.7. [TRACE]
Let $T = (S, Lab, \to, s_0)$ be an LTS*. A* trace *ζ of T is a pair of mappings $\zeta_\gamma : N \longrightarrow S$ and $\zeta_\lambda : N \setminus \{0\} \longrightarrow Lab$, where either $N = \{0, 1, 2, \dots, n\}$ or $N = \mathbb{N}$, and $(\zeta_\gamma(i) \to_{\zeta_\lambda(i+1)} \zeta_\gamma(i+1)) \in\to$ for all $i, (i+1) \in N$. If $N = \mathbb{N}$, trace ζ is called an* infinite *trace; otherwise, it is called a* finite *trace. The* length *of ζ is defined as $|N \setminus \{0\}|$ and referred to as $|\zeta|$.*

We use $\zeta^{(m)}$ to denote the prefix of ζ ending at $\zeta_\gamma(m)$. We also use $\zeta_{(m)}$ to denote the suffix of ζ starting at $\zeta_\gamma(m)$. Of course $\zeta^{(m)}$ is defined iff ζ has a length at least m. $\zeta_{(m)}$ is defined only in case ζ has a length at least $m + 1$.

Definition 2.8. [PARTITIONING]
Let $\zeta = (\zeta_\gamma, \zeta_\lambda)$ be a trace of an LTS *T. Let N be the domain of ζ_γ. Let I be a connected subset of N, i.e., it is either $I = \{z \mid k \leq z \leq m\}$, where $k < m$, or $I = \{z \mid z \geq k\}$. A pair $\zeta^I = (\zeta^I_\gamma, \zeta^I_\lambda)$ of mappings $\zeta^I_\gamma : I \longrightarrow S$ and $\zeta^I_\lambda : I \setminus \{k\} \longrightarrow Lab$ is called a* part *of trace ζ iff the following conditions are satisfied:*

- *for all $i \in I$: $\zeta^I_\gamma(i) = \zeta_\gamma(i)$;*
- *for all $j \in I \setminus \{k\}$: $\zeta^I_\lambda(j) = \zeta_\lambda(j)$.*

Let ζ^{I_1} and ζ^{I_2} be two parts of ζ. $\zeta^{I_1 \cup I_2}$ is called the concatenation *of ζ^{I_1} and ζ^{I_2} iff $I_1 = \{k, \ldots, m\}$ and $min_{i \in I_2} i = m$.*

A partitioning *of trace ζ is a finite or infinite sequence $\zeta^{I_1}\zeta^{I_2} \ldots$ of parts of ζ such that the concatenation of the parts coincides with ζ.*

Definition 2.9. [REACHABLE STATE]
Let $T = (S, Lab, \rightarrow, s_0)$ be an LTS. *A state $s \in S$ is* reachable *from the initial state of the system if there is trace ζ of T such that $\zeta_\gamma(0) = s_0$ and $\zeta_\gamma(i) = s$ for some $i \geq 0$.*

A wide range of behavioural equivalences and relations has been developed to distinguish two systems. Often the notion of equivalence between two systems is based upon the idea that we only distinguish between system T_1 and system T_2 if the distinction can be detected by an external system interacting with each of them. [71] provides an overview and comparison of existing behavioural equivalences. Further we review notions of trace equivalence, simulation, bisimulation and isomorphism for LTSs. Trace equivalence requires that systems can execute the same traces but does not require that systems have the same branching structure. Bisimulation and isomorphism allow to distinguish systems that have different branching structures. Let $T_1 = (S_1, Lab_1, \rightarrow^1, s_0^1)$ and $T_2 = (S_2, Lab_2, \rightarrow^2, s_0^2)$ be two LTSs.

Definition 2.10. [STRONG EQUIVALENCE OF TRACES]
Let ζ and ρ be traces of LTSs *T_1 and T_2 respectively. We say that $\zeta \equiv_{tr} \rho$ iff $|\zeta| = |\rho|$ and $\zeta_\lambda(i+1) = \rho_\lambda(i+1)$ for all $i = 0..|\zeta|$.*

Definition 2.11. [STRONG TRACE INCLUSION]
We say that the set of traces generated by LTS *T_2* includes *the set of traces generated by* LTS *T_1, written as $T_1 \preceq_{tr} T_2$, iff for every trace ζ of T_1 there exists a trace ρ in T_2 such that $\zeta \equiv_{tr} \rho$.*

Definition 2.12. [STRONG TRACE EQUIVALENCE]
Two LTSs *T_1 and T_2 are* trace equivalent*, written as $T_1 \equiv_{tr} T_2$, iff both $T_1 \preceq_{tr} T_2$ and $T_2 \preceq_{tr} T_1$.*

Definition 2.13. [STRONG SIMULATION]
A relation $R \subseteq S_1 \times S_2$ is a strong simulation *relation between* LTS *T_1 and* LTS *T_2 iff $s_0^1 R s_0^2$, and pRq together with $p \rightarrow_\lambda^1 p'$ implies $q \rightarrow_\lambda^2 q'$ and $p'Rq'$, for some $q' \in S_2$.*

We write $T_1 \preceq T_2$, if there exists a strong simulation relation R between T_1 and T_2.

Definition 2.14. [STRONG BISIMULATION]
A relation $R \subseteq S_1 \times S_2$ is a strong bisimulation *relation between* LTS *T_1 and* LTS *T_2 iff both R and R^{-1} are strong simulations.*

We write $T_1 \underline{\leftrightarrow} T_2$, if there exists a strong bisimulation relation R between T_1 and T_2.

It is straightforward to check that $\underline{\leftrightarrow}$ is an equivalence relation, i.e., $\underline{\leftrightarrow}$ is reflexive, symmetric and transitive.

Definition 2.15. [ISOMORPHISM]
Two LTSs T_1 *and* T_2 *are isomorphic iff there is a bijection* $g\colon S_1 \to S_2$ *such that* $\forall s, s' \in S_1,\ s \to^1_\lambda s'$ *iff* $g(s) \to^2_\lambda g(s')$.

Isomorphism implies strong bisimulation; strong bisimulation implies strong trace equivalence; strong simulation implies strong trace inclusion.

[72] provides an overview of existing process equivalences and relations in context of process algebras with *silent moves*. There, only a set of external actions of a system is visible to an observer and the internal structure of the system is hidden. Which activities of the system are hidden is a matter of choice that depends on the level of detail at which one wants to analyse the system. The hidden activities of the system are denoted as τ.

Weak trace equivalence relates two systems that can perform exactly the same sequence of observable actions.

Definition 2.16. [WEAK EQUIVALENCE OF TRACES]
Let ζ *and* ρ *be traces of* LTSs T_1 *and* T_2 *respectively. We say that* $\zeta \equiv_{wtr} \rho$ *iff* ζ *and* ρ *can be partitioned as* $\zeta^1\zeta^2\ldots$ *and* $\rho^1\rho^2\ldots$ *respectively, so that the following conditions are satisfied:*

- *every* ζ^k, ρ^k *has at most one step labelled with* $\lambda \neq \tau$*;*
- ζ^k *contains a step labelled by* $\lambda \neq \tau$ *iff* ρ^k *contains a step labelled by* $\lambda \neq \tau$.

Definition 2.17. [WEAK TRACE INCLUSION]
We say that the set of traces generated by LTS T_2 weakly includes *the set of traces generated by* LTS T_1*, written as* $T_1 \preceq_{wtr} T_2$*, iff for every trace* ζ *of* T_1 *there exists a trace* ρ *in* T_2 *such that* $\zeta \equiv_{wtr} \rho$.

Definition 2.18. [WEAK TRACE EQUIVALENCE]
Two LTSs T_1 *and* T_2 *are* weakly trace equivalent*, written as* $T_1 \equiv_{wtr} T_2$*, iff both* $T_1 \preceq_{wtr} T_2$ *and* $T_2 \preceq_{wtr} T_1$.

Intuitively, a τ-step is not truly silent if it results in a change of "potential" of the system. For example, consider the LTS on Fig. 1. After execution of the τ-step the system loses the option to execute step a. Therefore, the τ-step of the system is not truly silent. The intuition of a truly silent τ-step is formalized in the notions of branching simulation and branching bisimulation [70].

Definition 2.19. [BRANCHING SIMULATION]
A relation $R \subseteq S_1 \times S_2$ *is a* branching simulation *relation between* LTSs T_1 *and* T_2 *iff* $s^1_0 R s^2_0$ *and* pRq *together with* $p \to^1_\lambda p'$ *implies that one of the following conditions holds:*

1. $q \to_\tau q'_1 \to_\tau \ldots \to_\tau q'_n \to_\lambda q'$*, for some* $n \geq 0$ *and* $q', q'_1, \ldots, q'_n \in S_2$ *such that* pRq'_i *for all* $i = 1..n$ *and* $p'Rq'$*;*

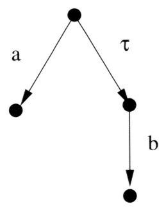

Fig. 1. Not truly silent τ

2. $\lambda = \tau$ *and* $p'Rq$.

We write $T_1 \preceq_{br} T_2$ *if there exists a branching simulation relation* R *between* T_1 *and* T_2.

Definition 2.20. [BRANCHING BISIMULATION]
A relation $R \subseteq S_1 \times S_2$ *is a branching bisimulation relation between* LTSs T_1 *and* T_2 *iff both* R *and* R^{-1} *are branching simulations.*

We write $T_1 \leftrightarrow_{br} T_2$ *if there exists a branching bisimulation relation* R *between* T_1 *and* T_2.

In [13], it was shown that branching bisimulation is indeed an equivalence relation.

2.3 Temporal and Modal Logic

Various logics have been developed to specify properties of systems and programs, e.g., computation tree logic ($CTL^\star$) [62], linear temporal logic (LTL) [137] and μ-calculus [112]. They allow to express universal and existential properties that hold for all or some traces of a system respectively. Various safety ("nothing bad will ever happen") and liveness ("something good must eventually happen") properties can be expressed as well.

Definition 2.21. [$CTL^\star$]
Given a set of atomic proposition $\mathcal{P}$, *the logic* $CTL^\star$ *consists of state* φ *and path* ψ *formulas defined by the following grammar:*

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathbf{A}\psi \mid \mathbf{E}\psi, \text{ where } p \in \mathcal{P}$$

$$\psi ::= \varphi \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \mathbf{X}\psi \mid \mathbf{F}\psi \mid \mathbf{G}\psi \mid \psi\mathbf{U}\psi \mid \psi\mathbf{R}\psi$$

In definition 2.21, **A** and **E** are *path quantifiers* meaning "for all paths" and "for some paths" respectively; **F**, **G**, **X**, **U**, **R** are temporal operators expressing properties of a single path. Formally, the $CTL^\star$ semantics is defined by Def. 2.24. The "eventually" operator **F** specifies that a property holds at some state of the path. The "always" operator **G** requires that a property holds at every state of the path. The "next" operator **X** expresses that a property holds in the second state of the path. The "unless" operator **U** holds if there is a state on the path where the second property holds and the first property holds at every state preceding this state. The "release" operator **R** is used to specify that the second property holds in all states along a path up to and including the first state that satisfies the first property. The first property is not required to be eventually satisfied.

Linear Temporal Logic, *LTL*, is a subset of *CTL** that consists of formulas having the form $\mathbf{A}\psi$ where ψ is a path formula in which the only state subformulas permitted are atomic propositions.

Definition 2.22. [*LTL*]
Given a set of atomic prepositions $\mathcal{P}$, the logic LTL consists of path formulas of the form defined by the following grammar:

$\psi ::= p \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \mathbf{F}\psi \mid \mathbf{G}\psi \mid \mathbf{X}\psi \mid \psi\mathbf{U}\psi \mid \psi\mathbf{R}\psi$
where $p \in \mathcal{P}$.

In *LTL* context, $\Box$ is often used instead of **G** to denote the "always" operator and $\Diamond$ is used instead of **F** to denote the "eventually" operator.

CTL* formulas are interpreted over Kripke structures.

Definition 2.23. [KRIPKE STRUCTURE]
A Kripke structure *K is a tuple $(T, S_0, \mathcal{L})$, where T is a transition system $(S,\ R)$, S_0 is a set of initial states, $\mathcal{L}\colon S \to 2^{\mathcal{P}}$ is an interpretation function and $\mathcal{P}$ is a set of atomic propositions.*

A *sequence* in the Kripke structure $K = (S, R, S_0, \mathcal{L})$ is $\pi = s_0 s_1 s_2 \ldots$ such that $(s_i, s_{i+1}) \in R$ holds for all $i \geq 0$. A *path* in the Kripke structure $K = (S, R, S_0, \mathcal{L})$ is an *infinite* sequence. We use $\pi_{(i)}$ to denote the suffix of π starting at state s_i; $\pi^{(i)}$ is used to denote the prefix of π starting at s_0 and ending at s_i. In case a state s has no outgoing transitions, we say that there is a *deadlock* in this state. Both paths and finite sequences $\pi = s_0 s_1 \ldots s_k$ ending at a state s_k that has a deadlock are called *computations* of K. The *length* of π (denoted $|\pi|$) is the number of states on it. The satisfaction of a formula is defined inductively.

Definition 2.24. [*CTL** SEMANTICS]*[42]*
Let $K = (S, R, S_0, \mathcal{L})$ be a Kripke structure over $\mathcal{P}$. Let $p \in \mathcal{P}$, ϕ, ϕ_1, ϕ_2 be state formulas and ψ, ψ_1, ψ_2 be path formulas, $s \in S$ and π be a path in K. The relation $\models$ is defined inductively as follows:

- $K, s \models p$ *iff* $p \in \mathcal{L}(s)$;
- $K, s \models \neg\phi$ *iff* $K, s \not\models \phi$;
- $K, s \models \phi_1 \vee \phi_2$ *iff* $K, s \models \phi_1$ *or* $K, s \models \phi_2$;
- $K, s \models \phi_1 \wedge \phi_2$ *iff* $K, s \models \phi_1$ *and* $K, s \models \phi_2$;
- $K, s \models \mathbf{E}\psi$ *iff there exists a path* π *starting in* s *such that* $K, \pi \models \psi$;

- $K, s \models \mathbf{A}\psi$ *iff for every path* π *starting in* s, $K, \pi \models \psi$;
- $K, \pi \models \phi$ *iff* π *starts in* s *and* $K, s \models \phi$;
- $K, \pi \models \neg\psi$ *iff* $K, \pi \not\models \psi$;
- $K, \pi \models \psi_1 \vee \psi_2$ *iff* $K, \pi \models \psi_1$ *or* $K, \pi \models \psi_2$;
- $K, \pi \models \psi_1 \wedge \psi_2$ *iff* $K, \pi \models \psi_1$ *and* $K, \pi \models \psi_2$;
- $K, \pi \models \mathbf{X}\psi$ *iff* $K, \pi_{(1)} \models \psi$;
- $K, \pi \models \mathbf{F}\psi$ *iff there exists a* $k \geq 0$ *such that* $K, \pi_{(k)} \models \psi$;
- $K, \pi \models \mathbf{G}\psi$ *iff for all* $i \geq 0, K, \pi_{(i)} \models \psi$;
- $K, \pi \models \psi_1\mathbf{U}\psi_2$ *iff there exists a* $k \geq 0$ *such that* $K, \pi_{(k)} \models \psi_2$ *and for all* $0 \leq j < k$, $K, \pi_{(j)} \models \psi_1$;
- $K, \pi \models \psi_1\mathbf{R}\psi_2$ *iff for all* $k \geq 0$, $\forall i < k\ K, \pi_{(i)} \not\models \psi_1$ *implies* $K, \pi_{(k)} \models \psi_2$.

There are several approaches to the interpretation of $CTL^\star$ path formulas. Although in the original definition of $CTL^\star$ in [62] both finite and infinite computations are taken into consideration, definition of $CTL^\star$ in [63] and in [60] revise the original definition by quantifying over paths only. A similar definition is given in [41]. In [61, 39], a transition relation of a Kripke structure is required to be *total* (i.e., every state must have an outgoing transition so that all computations are infinite).

In case only paths are taken in consideration, no properties about finite computations of the system can be expressed. If a Kripke structure contains a deadlock, this will not be caught by checking some $CTL^\star$ formula.

This deficiency is often repaired by assuring that all computations of the system are infinite before checking any other properties. In [47], a Kripke structure is extended by adding an extra state s and a transition leading to s from every state having no outgoing transitions (including s itself). New atomic proposition is_in_s, which is *true* only in s, is added to the set of atomic propositions. The check of deadlocks can be performed by checking $\mathbf{AG}\neg is\_in\_s$ over system transformed as described above. If this check succeeds, this implies that the original system is deadlock free. Once the system has been checked to be free from deadlocks, the extra transitions and state s can be removed again.

Similar approach is taken in [129]. There, the transition relation of Kripke structure does not have to be total. The authors propose a livelock extension for Kripke structures that is obtained by applying the following transformation: For each state that has no outgoing transitions or occurs in a cycle of states with the same labels, a new outgoing transition is added. The transition leads to a new state s that is labelled by a new proposition that occurs in no other labels on states.

Further in this thesis, we deal with transition systems (LTSs) where the transition relation is *total* due to the fact that a system is *timed* and time can progress even if the system can not do anything useful.

Next-free logic

Reactive systems are usually developed by a number of successive steps. At each step, the system is described in more detail and closer to the implementation level. Refinement allows the replacement of a higher level system specification by a lower, more detailed one. Refinement often changes the granularity of actions, i.e., high-level actions are mapped to multiple low-level actions, and therefore, high-level actions lose their atomicity.

The next operator **X** is closely related to the notion of a next state that can be reached by one step of the system. The **X** operator can be useful to express system properties, but it should be used with caution. The intuitive meaning of the **X** operator is not associated with the granularity of actions that the system can perform. In case the **X** operator is employed to express properties of the system, a change of the granularity can lead to a situation where properties satisfied by a higher level model become *false* for a lower level one.

The necessity of the **X** operator was already questioned in [117]. The main objection against the **X** operator was that it allows the designer to express irrelevant properties of the model. Using the **X** operator one can write a specification of a queue that includes a requirement: "removing an element from the queue should take exactly 17 steps". This property is not meaningful if one gives a high-level specification of a queue. It is a property of an implementation of the "remove" operation, but not a property of the "remove" operation itself. Therefore, Lamport proposed to drop the **X** operator from temporal logics. Further we refer to next-free $CTL^\star$ and to next-free LTL as $CTL^\star{-}X$ and $LTL{-}X$ respectively.

The work of Lamport is related to developments in the field of process equivalences, namely to the research on comparative concurrency semantics [71] in the context of process algebras with silent moves [72]. In [73], it is argued that branching bisimulation equivalence is the coarsest equivalence that respects the branching structure of a process with silent moves. In [129], De Nicola and Vaandrager showed that $CTL^\star{-}X$ induces on LTSs the same identification as branching bisimulation. According to [73], considering $CTL^\star$ would induce an equivalence that is too fine for processes with silent moves. In [47], Dams considers the development from strong bisimulation to branching bisimulation as the parallel of the shift of attention from $CTL^\star$ to $CTL^\star{-}X$ in specification logics.

Formulas of temporal logic are usually interpreted over Kripke structures; LTSs are mainly used for modelling purposes. In [129], De Nicola and Vaandrager introduced a new kind of structure that can be naturally projected on both LTSs and Kripke structures. The structure is called *doubly labelled transition systems.*

Definition 2.25. [DOUBLY LABELLED TRANSITION SYSTEMS] *[129]*
A doubly labelled transition system *(*L^{2}TS*) is a structure* $\mathcal{D} = (S, Lab, \rightarrow, s_0, \mathcal{L})$*, where* $(S, Lab, \rightarrow, s_0)$ *is an LTS and* $\mathcal{L} \colon S \rightarrow 2^{\mathcal{P}}$ *is a labelling function that associates a set of atomic propositions to each state. With* $LTS(\mathcal{D})$ *we denote the substructure* $(S, Lab, \rightarrow, s_0)$ *and* $KS(\mathcal{D})$ *denotes the substructure* $(S, R, s_0, \mathcal{L})$ *such that* $\forall s, s' \in S$, $(s, s') \in R$ *iff* $s \rightarrow_\lambda s' \in\rightarrow$ *for some* $\lambda \in Lab$.

Equivalences defined on LTSs or Kripke structures can be naturally lifted to L^{2}TSs by ignoring state or transition labels respectively. The notions of a path and a computation in an L^{2}TS are defined analogously to the notion of a path and a computation in a Kripke structure. The notion of a trace in an L^{2}TS is defined analogously to the notion of a trace in an LTS. For an L^{2}TS $\mathcal{D}$ and a formula φ of *LTL*, we write $\mathcal{D} \models \varphi$ iff $\mathrm{KS}(\mathcal{D}) \models \varphi$. Further we give a definition of path equivalence up to stuttering and an overview of results from [135, 136] relating path equivalence up to stuttering with *LTL–X*.

Definition 2.26. [STUTTERING-FREE PROJECTION] *[135, 136]*
Let $\pi = s_0 s_1 s_2 \ldots$ *be a sequence in* L^{2}TS $\mathcal{D} = (S, \rightarrow, S_0, \mathcal{L})$. *The stuttering-free projection* $Pr(\pi)$ *of* π *is defined coinductively as follows:*

- $Pr(s_0 s_1 s_2 \ldots) = s_0$ *if* $\forall i > 0$ *(or* $\forall 0 < i \leq |\pi|$*),* $\mathcal{L}(s_i) = \mathcal{L}(s_0)$*;*
- $Pr(s_0 \ldots s_k s_{k+1} \ldots) = Pr(s_0 \ldots s_k) Pr(s_{k+1} \ldots)$ *if* $\mathcal{L}(s_k) \neq \mathcal{L}(s_{k+1})$.

Definition 2.27. [EQUIVALENCE UP TO STUTTERING] *[135, 136]*
Let π *and* ρ *be paths in* L^2TSs $\mathcal{D}_1 = (S_1, \rightarrow^1, s_0^1, \mathcal{L}_1)$ *and* $\mathcal{D}_2 = (S_2, \rightarrow^2, s_0^2, \mathcal{L}_2)$ *respectively, where the range of labelling functions* $\mathcal{L}_1$ *and* $\mathcal{L}_2$ *is* $2^{\mathcal{P}}$. *We say that* π *and* ρ *are* equivalent up to stuttering, *written as* $\pi \equiv_{st} \rho$, *iff* $\mathcal{L}_1(Pr(\pi)(i)) = \mathcal{L}_2(Pr(\rho)(i))$ *for all* $i \geq 0$ *(or* $\forall 0 < i \leq |Pr(\pi)|$ *and* $|Pr(\rho)| = |Pr(\pi)|$*), i.e., the interpretations of the stuttering-free projections of* π *and* ρ *are the same.*

Definition 2.28. [PATH EQUIVALENCE UP TO STUTTERING] *[135, 136]*
Let $\mathcal{D}_1 = (S_1, \rightarrow^1, s_0^1, \mathcal{L}_1)$ *and* $\mathcal{D}_2 = (S_2, \rightarrow^2, s_0^2, \mathcal{L}_2)$ *be two* L^{2}TSs *where the range of labelling functions* $\mathcal{L}_1$ *and* $\mathcal{L}_2$ *is* $2^{\mathcal{P}}$.

We write $\mathcal{D}_1 \preceq_{st} \mathcal{D}_2$ *iff for each path* π *in* $\mathcal{D}_1$ *there is a path* ρ *in* $\mathcal{D}_2$ *such that* $\pi \equiv_{st} \rho$.

$\mathcal{D}_1$ *and* $\mathcal{D}_2$ *are* path equivalent up to stuttering, *written as* $\mathcal{D}_1 \equiv_{st} \mathcal{D}_2$, *iff* $\mathcal{D}_1 \preceq_{st} \mathcal{D}_2$ *and* $\mathcal{D}_2 \preceq_{st} \mathcal{D}_1$.

Theorem 2.2. *[135]*
Let $\mathcal{D}_1$ *and* $\mathcal{D}_2$ *be two* L^{2}TSs, *where the range of labelling functions* $\mathcal{L}_1$ *and* $\mathcal{L}_2$ *is* $2^{\mathcal{P}}$.

Let $\mathcal{D}_1 \preceq_{st} \mathcal{D}_2$. *Then* $\mathcal{D}_1 \models \varphi$ if $\mathcal{D}_2 \models \varphi$ *for any LTL–X formula* φ *over* $\mathcal{P}$.

Let $\mathcal{D}_1 \equiv_{st} \mathcal{D}_2$. *Then* $\mathcal{D}_1 \models \varphi$ iff $\mathcal{D}_2 \models \varphi$ *for any LTL–X formula* φ *over* $\mathcal{P}$.

Modal μ-calculus

Definition 2.29. [MODAL μ-CALCULUS, L_μ] *[112, 51]*
Let Var be a set of propositional variables. *Let* $\mathcal{P}$ *be the set of* atomic propositions. *Moreover, let* $p \in \mathcal{P}$, $x \in Var$, $\varphi \in L_\mu$. *The logic* L_μ *is the set of formulas that is defined by the following grammar:*

$$\varphi ::= p \mid \neg p \mid x \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \Box \varphi \mid \Diamond \varphi \mid \mu x.\varphi \mid \nu x.\varphi$$

Formula $\Box\varphi$ expresses that φ is true for every immediate successor, while $\Diamond\varphi$ expresses that there is at least one successor for which φ is true. $\mu x.\varphi$ and $\nu x.\varphi$ are the least and greatest fixpoint operators respectively. Their meaning is the smallest (respectively, greatest) set x of states in which φ holds.

Here we consider only formulas in *positive normal form*, where all negations occurring in the formula are applied to atomic propositions and no variable is quantified twice [112]. The *universal* and *existential* fragments $\Box L_\mu$ and $\Diamond L_\mu$ are subsets of L_μ in which the only allowed next-state operators are $\Box$ and $\Diamond$ respectively. L_μ^+ denotes the *positive fragment* of L_μ, where the use of negation is forbidden even on the atomic propositions.

L_μ formulas are interpreted over a transition system (see Def. 2.5) with an interpretation function associated to the transition system. Intuitively, $\mathcal{I}(p)$ is the set of states where p holds. A function $\|\cdot\|$ interprets an L_μ formula over a given transition system with an interpretation function.

Definition 2.30. [L_μ SEMANTICS] *[51]*
Let $T = (S, R)$ be a transition system and $\mathcal{I}\colon \mathcal{P} \to 2^S$ be an interpretation function. The function $\|\cdot\|\colon (L_\mu \times (Var \to 2^S)) \to 2^S$ is defined as follows. Let $p \in \mathcal{P}$, $x \in Var$, φ, φ_1, $\varphi_2 \in L_\mu$ and $e\colon Var \to 2^S$.

$$\begin{aligned}
\|p\|_e &= \mathcal{I}(p)\\
\|\neg p\|_e &= \overline{\mathcal{I}(p)}\\
\|x\|_e &= e(x)\\
\|\varphi_1 \vee \varphi_2\|_e &= \|\varphi_1\|_e \cup \|\varphi_2\|_e\\
\|\varphi_1 \wedge \varphi_2\|_e &= \|\varphi_1\|_e \cap \|\varphi_2\|_e\\
\|\Box\varphi\|_e &= \{s \in S \mid \forall\, s' \in S\ sRs' \Rightarrow s' \in \|\varphi\|_e\}\\
\|\Diamond\varphi\|_e &= \{s \in S \mid \exists\, s' \in S\ sRs' \wedge s' \in \|\varphi\|_e\}\\
\|\mu x.\varphi\|_e &= \bigcap\{S' \subseteq S \mid\ \|\varphi\|_{e[x\mapsto S']} \subseteq S'\}\\
\|\nu x.\varphi\|_e &= \bigcup\{S' \subseteq S \mid\ S' \subseteq \|\varphi\|_{e[x\mapsto S']}\}
\end{aligned}$$

$e[x\mapsto S']$ is the same as e except that x is mapped to S'. We write $s \models \varphi$ for $s \in \|\varphi\|$. For a set of states S', the notation $S' \models \varphi$ abbreviates $\forall\, s \in S',\ s \models \varphi$. When there may be confusion between different systems and interpretation functions we write $T, (I), s \models \varphi$ to denote that $s \models \varphi$ in T with interpretation function $\mathcal{I}$.

The existential and universal fragments of μ-calculus subsume the existential and universal fragments of $CTL^\star$ ([87]).

2.4 Model Checking and Automata Theory

Model checking [38, 120, 165, 35, 42] is a formal technique for verifying finite-state systems with respect to their specification. The properties of a system are expressed as formulas in some temporal logic [137, 60]. A model checker either confirms that the system satisfies the properties or reports that they are violated. In case a property gets violated, the model checker produces a counter-example that is a system run that violates the property.

Model checking comes in two fashions: symbolic model checking as e.g. in *NuSMV* [37] and *COSPAN* [83], where Ordered Binary Decision Diagrams [32] are used to represent states of the system symbolically, and explicit state model checking as in *Spin* [93] and CADP [65], where the states of the system are explicitly enumerated. Here we give an overview of the explicit-state automata-based approach to model checking [165, 93]. In this approach, both the system and the negation of a property are turned into a finite automaton on infinite words [158]. The verification consists in checking whether the language recognized by the synchronous product of the above automata is empty.

Further we focus on finite automata over infinite words. The simplest automata over infinite words are Büchi automata [34].

Definition 2.31. [BÜCHI AUTOMATON] *[127]*
A Büchi automaton *$B = (Q, I, \delta, F)$ over an alphabet Σ is given by a finite set Q of states, a non-empty set $I \subseteq Q$ of initial states, a transition relation $\delta \subseteq Q \times \Sigma \times Q$ and a set $F \subseteq Q$ of accepting states.*

A run *of B over an ω-word $w = a_0 a_1 \ldots \in \Sigma^\omega$ is an infinite sequence $\rho = q_0 q_1 q_2 \ldots$ such that $q_0 \in I$ and $(q_i, a_i, q_{i+1}) \in \delta$ holds for all $i \in \mathbb{N}$. The run ρ is* accepting *iff there exists some $q \in F$ such that $q_i = q$ holds for infinitely many $i \in \mathbb{N}$.*

A language $L(B) \subseteq \Sigma^\omega$ is the set of ω-words for which there exists some accepting run ρ of B. A language $L \subseteq \Sigma^\omega$ is called ω-regular *iff $L = L(B)$ for some Büchi automaton B.*

Finite automata can be used to model concurrent and interactive systems. A Kripke structure directly corresponds to a finite automaton over infinite words where all states are accepting. Specifically, a Kripke structure $K = (S, R, s_0, \mathcal{L})$ where $\mathcal{L} \colon S \to 2^{\mathcal{P}}$ can be seen as an automaton $A = (S, S_0, \delta, S)$ over $2^{\mathcal{P}}$, where $(s, a, s') \in \delta$ for $s, s' \in S$ iff $(s, s') \in R$ and $a = L(s)$ [168]. (Note that in the general case there is a set of initial states, but for model checking we only require a single initial state.) The specification can be transformed into an automaton B, over the same alphabet. The system A *satisfies* the specification B when $L(A) \subseteq L(B)$. That can be rewritten as $L(A) \bigcap \overline{L(B)} = \emptyset$, i.e., there is no behaviour of A that is disallowed by B. If the intersection is not empty, any behaviour in it corresponds to a counter-example.

Büchi automata are closed under intersection and complement [34]. This means that there exists an automaton that accepts exactly the intersection of the languages of two automata and an automaton that recognizes exactly the complement of the language of a given automaton. In some implementations such as *Spin* [93], the automaton for the complement of the specification is used instead of the automaton for the specification. In this case, not the good behaviour is specified, but the bad behaviour.

We show how to construct an automaton that recognises the intersection of two languages accepted by a pair of Büchi automata. Since all the states of the automaton for the modelled system are accepting, we give the definition of the product for the case when all the states of one of the automata are accepting.

Definition 2.32. [SYNCHRONOUS PRODUCT] *[42]*
Let $B_1 = (Q_1,\ \delta_1,\ Q_1^0,\ Q_1)$ *and* $B_2 = (Q_2,\ \delta_2,\ Q_2^0,\ F_2)$ *be Büchi automata over* Σ*. The synchronous product of* B_1 *and* B_2 *is the automaton* B *over* Σ *given by* $(Q_1 \times Q_2,\ \delta,\ Q_1^0 \times Q_2^0,\ Q_1 \times F_2)$ *such that* $((q_1, q_2), a, (q_1', q_2')) \in \delta$ *iff* $(q_1, a, q_1') \in \delta_1$ *and* $(q_2, a, q_2') \in \delta_2$.

It is straightforward to show that the synchronous product of automata B_1 and B_2 accepts $L(B_1) \bigcap L(B_2)$.

Let ρ be an accepting run of a Büchi automaton $B = (Q, I, \delta, F)$ over an alphabet Σ. Then ρ contains infinitely many accepting states from F. Since Q is a finite set, there is some suffix ρ' of ρ such that every state of Q that is met along the suffix appears infinitely many times. Each state on ρ' is reachable from any other state on ρ', i.e., the states in ρ' form a cycle. This component is reachable from some initial state. It contains an accepting state of the automaton and generates an accepting run of it. Checking non-emptiness of $L(B)$ is equivalent to finding a strongly connected component that is reachable from an initial state and contains an accepting state. In other words, the language $L(B)$ is non-empty if and only if there is a reachable accepting state that is on a cycle [42].

The automaton for the specification can have as many as $2^{O(n)}$ states where n is the number of subformulas in the specification [165]. The size of the product automaton which determines the overall complexity of the method is proportional to $N \cdot 2^{O(n)}$, where N is the number of reachable states of the modelled system. Model checking on-the-fly allows to detect a property violation by constructing and visiting only some part of the search space containing a counter-example.

The *nested depth first search* (*ndfs*) algorithm (Fig.2) is used for finding cycles with accepting states (*accepting cycles*) "on-the-fly" [96, 45, 127]. Given a property φ and a Kripke structure K, the model checking problem is reformulated as follows: Does there exist a run of K that does not satisfy φ? We ask whether the language of the automaton defined by the product of the automaton for K and the automaton for $\neg\varphi$ is empty or not.

ndfs is an "on-the-fly" algorithm because the exploration of reachable states is interleaved with the search for acceptance cycles. The algorithm keeps the stack of all states whose successors need to be explored and the set of states that have already been visited. Starting from an initial state, the procedure *ndfs* generates reachable states until an accepting state is met. The search

Procedure 2.3 (*emptyness*) *[127]*

```
\\ initialization
stack = emptystack(); visited = emptyset(); seed = nil;
for each $q_0 \in Q_0$ {
  push $q_0$ onto stack;
  enter $(q_0, false)$ into visited;
  ndfs(false)
}
```

Procedure 2.4 (*ndfs(boolean search_cycle)*)

```
q = top(stack);
for each $q' \in successors(q)$ {
  if $(search\_cycle \wedge (q' == seed))$
    report acceptance cycle and exit;
  if $((q', search\_cycle) \notin visited)$ {
    push $q'$ onto stack;
    enter $(q', search\_cycle)$ into visited;
    ndfs(search_cycle);
    if $(\neg search\_cycle \wedge (q'$ is accepting$))$ {
      seed: $= q'$; ndfs(true);
      }
    }
}
pop(stack);
}
```

Fig. 2. Nested depth-first search algorithm

then switches to the cycle search mode (indicated by the boolean variable *search_cycle*) and tries to find a path that leads back to the accepting state. The algorithm reports an acceptance cycle if one exists, although it does not guarantee to find all cycles, because the exploration stops as soon as an error has been found [127].

Lemma 2.1. [CORRECTNESS OF *ndfs* ALGORITHM] *[45]*
The ndfs algorithm returns a counterexample for the emptiness of the automaton B exactly when the language $L(B)$ is not empty.

If the acceptance cycle is found, the sequence of system states in the stack represents a path of K that violates property φ. The algorithm needs to store only the path from the current state back to the initial state and the set of visited states. If no acceptance cycle is found, all reachable states have to be visited.

Spin and *DTSpin*

For the majority of experiments mentioned in this thesis, we used *DTSpin* [24, 55], a discrete time extension of the *Spin* model checker.

Spin [93] is a state-of-the-art, enumerative model-checker with an expressive input-language PROMELA. In an extensive list of industrial applications, *Spin* and PROMELA have proven to be useful for the verification of industrial systems. *Spin* can be used not only as a simulator for rapid prototyping that supports random, guided and interactive simulation, but also as a powerful state space analyzer for proving user-specified correctness properties of the system. As standard *Spin* does not deal with timing aspects of protocols, *DTSpin*, a discrete time extension of *Spin*, has been developed [24, 55], that can be used for verification of properties depending on timing parameters. The extension is compatible with the standard untimed version of the *Spin* validator, except for the timeout statement, which has different semantics and its usage is no longer allowed (nor necessary) in discrete-time models.

Fairness

The behavior of a reactive system depends not only on the properties of the individual components running in parallel, but also on the interactions among those components. These interactions depend on external factors such as the relative speed of processors or the particular scheduler implementation whose details can be complex or even unknown. By introducing appropriate fairness assumptions stating that every sufficiently enabled component eventually proceeds, we can abstract away from these details without ignoring them completely.

Most of the common notions of fairness share the same general form: Every entity that is enabled sufficiently often will eventually make progress. Depending on the interpretations of "entity" and "sufficiently often" we get different notions of fairness. In the context of communicating processes, there are many different kinds of entities to consider, each choice leading to a different notion of fairness. In particular, Francez [68] and Kuiper and de Roever [114] have identified a hierarchy of fairness notions for CSP that includes the following forms of fairness: process fairness, channel fairness, guard fairness, and communication fairness. Each of these fairness notions have *weak* and *strong* varieties, which differ in the interpretation of "sufficiently often": *weak* forms of fairness concern with *continuously enabled* entities, whereas *strong* forms of fairness concern with *infinitely* (but not necessarily continuously) *enabled* entities.

Process fairness is one of the most common notions of fairness, due to its applicability in the context of communicating processes. *Weak (process) fairness* (also known as *justice* [110]) states that every *continuously* enabled process will eventually make progress. Intuitively, weak fairness ensures that the scheduler will never forget the process forever. It is straightforward to implement weak fairness as a scheduling policy, using a simple round-robin scheduling queue. An alternative to weak fairness, *strong (process) fairness* (also known as *compassion* [110]), states that every *infinitely* enabled process makes progress infinitely often.

2.5 Verification by Abstraction

Model checking can be applied to programs that have a relatively small finite state space. State space explosion remains a stumbling block of model checking. Various techniques were developed to solve this problem. One of them is *verification by abstraction.* Abstraction means replacing a semantical model by an abstract, in general simpler (finite) one. In addition to the requirement that an abstract (verification) model should have a smaller state space than the concrete (implementation) one, the abstraction needs to be *safe*, which means that every property checked to be true on the abstract model, holds for the concrete one as well. A safe abstract system is, intuitively, a system whose behaviour contains at least the behaviour of the concrete system [47]. This allows the transfer of positive verification results from the abstract model to the concrete one.

Let M be a system, whose semantics is given by transition system $T = (S, R)$, and let ${}_{\alpha}T = ({}_{\alpha}S, {}_{\alpha}R)$. *Description* relation $\rho \subseteq S \times {}_{\alpha}S$ gives for the states from T their "descriptions" in ${}_{\alpha}T$.

Definition 2.33. [PRE-IMAGE AND POST-IMAGE FUNCTIONS] *[121]*
Given a relation $\rho \subseteq S \times {}_{\alpha}S$ we define $pre[\rho]\colon 2^{{}_{\alpha}S} \to 2^S$ and $post[\rho]\colon 2^S \to 2^{{}_{\alpha}S}$ by:

$$pre[\rho](X) = \{s \in S \mid \exists s^{\alpha} \in X, (s, s^{\alpha}) \in \rho\}$$

$$post[\rho](Y) = \{s^{\alpha} \in {}_{\alpha}S \mid \exists s \in Y, (s, s^{\alpha}) \in \rho\}$$

The duals $\widetilde{pre}[\rho]\colon 2^{{}_{\alpha}S} \to 2^S$ *and* $\widetilde{post}[\rho]\colon 2^S \to 2^{{}_{\alpha}S}$ *are defined by* $\widetilde{pre}[\rho](X) = \overline{pre[\rho](\overline{X})}$ *and* $\widetilde{post}[\rho](Y) = \overline{post[\rho](\overline{Y})}$.

Given a description relation ρ, the functions $\alpha = post[\rho]$ and $\gamma = \widetilde{pre}[\rho]$, called *abstraction* and *concretization* respectively, form a Galois connection (see Def. 2.4) from 2^S to $2^{{}_\alpha S}$.

Lemma 2.2. [GALOIS CONNECTION GENERATED BY RELATION] *[121]*
If $\rho \subseteq S \times S^\alpha$, then $(post[\rho], \widetilde{pre}[\rho])$ is a Galois connection from 2^S to $2^{{}_\alpha S}$ and $(pre[\rho], \widetilde{post}[\rho])$ is a Galois connection from $2^{{}_\alpha S}$ to 2^S.

Further we review the definition of simulation parameterized by a Galois connection and the definition of simulation parameterized by a description relation ρ. Lemma 2.3 and Lemma 2.4 from [121] show that these two notions of simulation coincide. Let $T = (S, R)$ and ${}_\alpha T = ({}_\alpha S, {}_\alpha R)$ be two transition systems.

Definition 2.34. [SIMULATION PARAMETERIZED BY CONNECTION] *[121]*
Let (α, γ) be a Galois connection from 2^S to $2^{{}_\alpha S}$. Define $T \sqsubseteq_{(\alpha,\gamma)} {}_\alpha T$ iff $\alpha \circ pre[R] \circ \gamma \subseteq pre[{}_\alpha R]$.

Definition 2.35. [SIMULATION PARAMETERIZED BY DESCRIPTION] *[121]*
Let ρ be a description relation, $\rho \subseteq S \times {}_\alpha S$. Define $T \sqsubseteq_\rho {}_\alpha T$ iff $R^{-1}\ \rho\ \subseteq \rho\ {}_\alpha R^{-1}$.

Lemma 2.3. [FROM $\sqsubseteq_{(\alpha,\gamma)}$ TO $\sqsubseteq_\rho$] *[121]*
For any description relation $\rho \subseteq S \times {}_\alpha S$, there exists a Galois connection (α, γ) from 2^S to $2^{{}_\alpha S}$ such that $T \sqsubseteq_\rho {}_\alpha T$ iff $T \sqsubseteq_{(\alpha,\gamma)} {}_\alpha T$.

Lemma 2.4. [FROM $\sqsubseteq_\rho$ TO $\sqsubseteq_{(\alpha,\gamma)}$] *[121]*
For any Galois connection (α, γ) from 2^S to $2^{{}_\alpha S}$, there exists a description relation $\rho \subseteq S \times {}_\alpha S$ such that $T \sqsubseteq_{(\alpha,\gamma)} {}_\alpha T$ iff $T \sqsubseteq_\rho {}_\alpha T$.

Definition 2.36. [ABSTRACTION]
We say that ${}_\alpha T$ is an abstraction *of T iff there exists a description relation $\rho \subseteq S \times {}_\alpha S$ such that $T \sqsubseteq_\rho {}_\alpha T$.*

The notion of consistency defined below shows when an abstraction function $\alpha \colon 2^S \to 2^{{}_\alpha S}$ preserves the meaning of the atomic propositions defined by an interpretation function $\mathcal{I}$ (see Sec. 2.3) on 2^S. The abstraction function α is consistent with $\mathcal{I} \colon \mathcal{P} \to 2^S$, where $\mathcal{P}$ is a set of atomic propositions, if for all atomic propositions p the abstractions of $\mathcal{I}(p)$ and $\overline{\mathcal{I}(p)}$ by α are disjoint, i.e., abstractions of interpretation of p and $\neg p$ are not contradictory. In case (α, γ) is a Galois connection, consistency of α with $\mathcal{I}$ expresses the fact that $\widetilde{\gamma} \circ \alpha$ strongly preserves the interpretation of all atomic propositions (see Lemma 2.5).

Definition 2.37. [CONSISTENT ABSTRACTION FUNCTION] *[121]*
Let $\mathcal{I} \colon \mathcal{P} \to 2^S$ be an interpretation function. $\alpha \colon 2^S \to 2^{{}_\alpha S}$ is consistent *with $\mathcal{I}$ if $\forall\ p \in \mathcal{P}$, $\alpha(\mathcal{I}(p)) \cap \alpha(\overline{\mathcal{I}(p)}) = \emptyset$.*

Lemma 2.5. [CHARACTERIZATION OF CONSISTENCY] *[121]*
If (α, γ) is a Galois connection from 2^S to $2^{{}_\alpha S}$, then α is consistent with $\mathcal{I}$ iff

$$\forall p \in \mathcal{P},\ \gamma(\alpha(\mathcal{I}(p))) = \mathcal{I}(p)$$

We give here an overview of the preservation results for the $\Box L_\mu^+$ and $\Box L_\mu$ fragments of modal μ-calculus from [121]. The preservation results allow the use of the following verification method: Given a concrete system T and some Galois connection (α, γ), compute an abstract system ${}_\alpha T$, such that $T \sqsubseteq_{(\alpha,\gamma)} {}_\alpha T$. In order to verify a property expressed as a formula φ of $\Box L_\mu$, verify the property on ${}_\alpha T$. If φ holds on ${}_\alpha T$, it also holds on T.

Theorem 2.5. [PRESERVATION OF $\Box L_\mu^+$ AND $\Box L_\mu$] *[121]*
Let $T \sqsubseteq_{(\alpha,\gamma)} {}_\alpha T$. Let $\mathcal{I} \colon \mathcal{P} \to 2^S$ and ${}_\alpha\mathcal{I} \colon \mathcal{P} \to 2^{{}_\alpha S}$ be two interpretation functions. Then $\widetilde{\gamma}$ preserves the formulas of $\Box L_\mu^+$ from ${}_\alpha T$ to T and if $\widetilde{\gamma}$ is consistent with ${}_\alpha\mathcal{I}$ then $\tilde{\gamma}$ preserves the formulas of $\Box L_\mu$ from ${}_\alpha T$ to T.

3

Timer Transformation to Verify SDL Specifications

Industrial-size specifications/models, whose state spaces are often infinite, can in general not be model checked in a direct way. Program transformation is a way to build a finite-state verification model that can be submitted to a model checker. This chapter presents a transformation of SDL timers aimed at reducing the infinite domain of timer values to a finite one while preserving system behaviours.

The chapter is based on [101].

3.1 Introduction

The development of a specification language and its semantical concept are greatly affected by the intended mode of its use, its application domain. Often, the final objective is to provide an executable specification/implementation. In that case, the specification language and its semantics should provide a framework for constructing faithful and detailed descriptions of systems. No wonder that specifications written in these implementation-oriented languages are harder to verify than the ones written in the languages developed as input languages for model checkers. In this chapter, we concentrate on some aspects of modelling time in implementation-oriented languages, taking SDL (Specification and Description Language) [133] as an instance of this class of languages.

SDL is used for the specification of real-time systems like telecommunication software as well as aircraft and train control, medical and packaging systems, all of which must respond within certain time limits. Time-supervision is employed to control response time from unreliable resources and release of shared or limited resources. It can also be used to establish activities that must be repeated on a regular basis. Behaviour of a system specified in SDL is scheduled with the help of timers declared in the specification. The model of SDL timers was induced by manners of implementation of timers in real systems. An SDL timer can be activated by setting it to a value $(\texttt{NOW} + \delta)$ where the expression `NOW` provides an access to the current system time and δ is a delay after which this timer expires, i.e., the timer expires when the system time (system clock) reaches the point $(\texttt{NOW} + \delta)$. Such an implementation of timers immediately means that the state space of SDL specifications is infinite just because timer variables take an infinite number of values due to the value of `NOW` which grows during the system run. An inverse timer model is normally employed in verification-oriented languages: a timer indicates the delay left until its expiration, i.e., a timer is set to a value δ instead of $(\texttt{NOW} + \delta)$, and this value is decreased at every tick of the system clock. When the timer value reaches zero, the timer expires. This model of timers guarantees that every timer variable takes only a finite (and relatively small) number of values.

Another SDL peculiarity that adds to the complexity of verification is the manner in which timers expire. SDL is based on the Communicating Extended State Machines; communication is organized via message passing. For the uniformity of communication, timers are considered as a special kind of signals and a process learns about a timer expiration by dint of a signal with the name of the expired timer, inserted in the input port of the process. The timeout message can be added in the input port at any point of the time slice, in which the timer is ready to expire. From the verification point of view it would be better if a timer expiration had been diagnosed by a simple check of the timer value. Treating timeouts as messages, one gets all possible combinations of timeouts with messages exchanged by the processes, which increases the state space of the system.

Though formal verification of SDL specifications is an area of rather active investigations [25, 94, 91, 147, 160], for a long time the time-concerned difficulties were being got around by means of abstracting out time and timers. Due to engineering rather than formal approaches to constructing abstractions, some proposed abstractions turned out to be unsafe (see [25] for details).

In [25], a toolset and a methodology for the verification of *time-dependent* properties of SDL-specifications are described. The SDL-specifications are first translated into DTPROMELA, the input language of the *DTSpin* (Discrete Time Spin) model checker [24], and then verified against *LTL* formulas. Some informal arguments in favour of correctness of the DTPROMELA translation to the original specification are given, but no formal proof that the results of the verification of the transformed system are transferable to the original one is provided.

In this chapter, we propose a transformation that substitutes the traditional SDL timers by timer variables. In the transformed model, the timers are inverse and timeouts are not put into the input queue, but modelled directly by guards. The underlying idea is similar to the one in [25], but we provide a formal proof of path equivalence up to stuttering, which substantiates that the transformed system can be *safely* used for verification. The semantics of transformed systems will be further used in Chapter 5 to present an approach to automatic closing SDL specifications.

The chapter is organized as follows. In Section 3.2, we give an overview of syntax and define the semantics of the subset of SDL we concentrate on. The transformation substituting timeouts-as-messages by timeouts-by-guards is given in Section 3.3. This section also gives the semantics of transformed systems. The proof that the results of the verification of the transformed system are transferable to the original one is provided in Section 3.4. We conclude this chapter with Section 3.5.

3.2 SDL

The development of SDL started in 1972 when a study group within the telecommunications union ITU-T (CCITT at this time) representing several countries and large telecommunication companies began research on a specification language for the telecommunication industry. SDL is standardized by ITU-T as standard Z.100. The first version of the language issued in 1976 was followed by new versions every fourth year since. The formal semantics of SDL, defined in 1988 and further updated for subsequent versions of the SDL language in 1992 and 1996, is based on the combination of the VDM meta language Meta-IV with a CSP-like communication mechanism. It provides a formalization of the static [143] and dynamic semantics [145] of SDL. The semantics is documented by more than 500 pages of Meta-IV descriptions and hardly manageable because of its size. The latest version of SDL called SDL-2000 was approved as an international standard in 1999 [142, 144, 146].

Besides an official semantics developed within ITU-T there are other approaches to the formalization of the semantics of SDL. In [30, 90, 92], semantics of various subsets of SDL are defined based on stream processing functions of FOCUS [31]. SDL processes are modelled as discrete streams of signals. This stream based semantics neither supports the concept of states and transitions nor provides an adequate treatment of time aspects of SDL. It restricts the fundamental notion of system time that makes it not suitable for our purposes. The transformation proposed in this chapter deals mainly with time aspects, so the stream based semantics is not suitable for proving the correctness of the transformation.

In [66], SDL Time Nets (an extended Petri Net model) are proposed as a basis for the formal verification of communication protocols specified in SDL. A process algebra semantics of a restricted version of SDL is defined in [16]. The authors admit that the extension of discrete time process algebra with relative timing, used to describe the meaning of language constructs of SDL, is fairly large and rather intricate. A compiler-oriented semantics of SDL-2000 [64] is defined by an SDL semantics group. The key issues of this approach that uses abstract state machines as a formal basis are maintenance and *executability* of an SDL specification.

Further in this section, we provide an example of an SDL process, give an overview of the syntax, define the set of specifications we work with, and define their semantics. The transformation that we propose is related to behavioural aspects of SDL systems, rather than to structural ones. Therefore, we concentrate on a subset of SDL that is used for specification of behaviour, without considering structuring concepts used for describing large system. We also do not deal with abstract data types, but assume a specification to be well-typed and a few data types together with their interpretations to be predefined.

3.2.1 Syntax Overview

Systems described by SDL consist of many processes that run simultaneously and communicate with each other by exchanging signals via channels. An SDL system is specified by a *system diagram* that consists of a system text area, where channels and signals are defined, and a process interaction area. Further, we refer to the set of channel names defined in a system text area as *Chan*. The set *Chan* is partitioned into $Chan_i$ and $Chan_o$ of input and output channels, and we write $c_i, c'_o, \ldots$ to denote membership of a channel in one of these classes. The set of signal names defined by the signal declarations is denoted as *Sig*. A process interaction area is formed by one or more process diagrams.

A *process diagram* is formed by a process heading, a process text area and a process graph area enclosed by a frame symbol (see Fig. 3). A process text area contains declarations of process variables and timers. For variables, we assume data types **Integer**, **Boolean**, **Time** and **Duration** together with their natural interpretations $\mathbb{Z}$, *Bool*, $\mathbb{Z}$ and $\mathbb{Z}$ to be predefined.

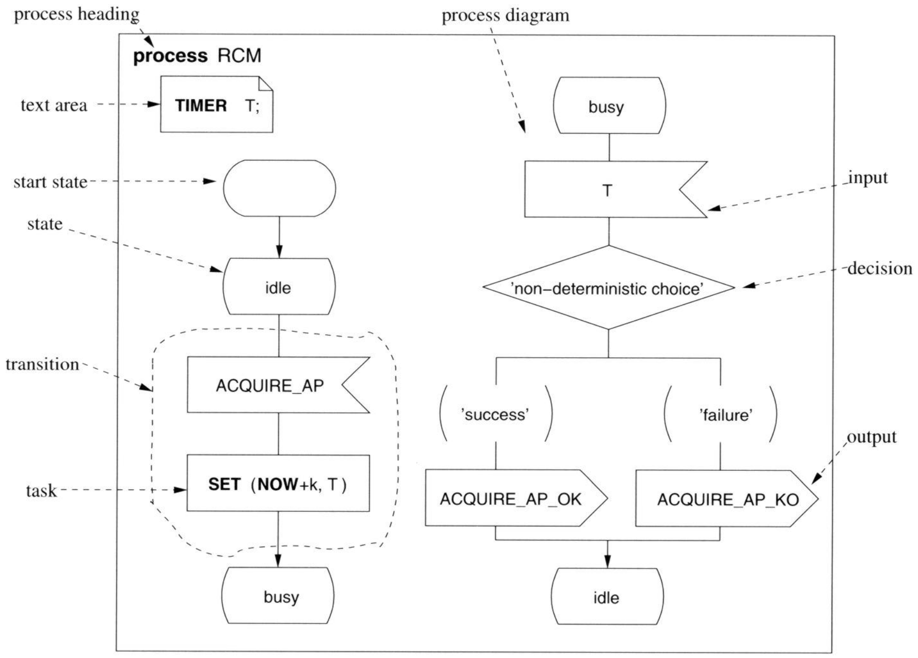

Fig. 3. An SDL process

A *process graph area* contains a graph defining process *behaviour* in terms of *states* and *transitions* with a start state as its root. Transitions are decorated by *input*, *output*, *task* and *decision* symbols. The process diagram in Fig. 3 defines process RCM. The process stays in state idle until it gets the ACQUIRE_AP signal as input. After this input, the process sets timer T and waits in state busy until timer T expires. Upon expiration of the timer, the process makes a non-deterministic decision between sending either the ACQUIRE_AP_OK or the ACQUIRE_AP_KO signal and returns to the idle state.

Formally, we define a specification of a process P by a tuple (pid, In, Out, Var, $Timer$, Loc, Act, Edg, l_0), where pid is a unique process identity, In is a finite set of *input* channel names of the process, Out is a finite set of *output* channel names, Var denotes a finite set of variables, $Timer$ denotes a finite set of timers, and Loc denotes a finite set of *locations* or control states; Act is the finite set of actions; the set $Edg \subseteq Loc \times Act \times Loc$ denotes the set of edges. For an edge $(l, \alpha, \hat{l}) \in Edg$ of P, we write more suggestively $l \longrightarrow_{\alpha} \hat{l}$.

The set Loc consists of states declared by the process diagram and intermediate control states between statements decorating arrows of transitions. In the set Loc, we distinguish the subset Loc_i of *input locations*, i.e. the locations where input actions are allowed. Note that only input actions are allowed in these locations. The initial location l_0 does not necessarily belong to Loc_i.

The set *Edg* defines possible changes of locations by performing *actions* from the set *Act*. As *untimed* actions, we distinguish (1) *input* of a signal s containing a value to be assigned to a local variable, (2) *sending* over a channel c a signal s together with a value described by an expression, and (3) *assignments*. In SDL, each transition starts with an input action, hence we assume the inputs to be unguarded, while output and assignment are *guarded* by a boolean expression g, its guard. The three classes of actions are written as $?s(x)$, $g \rhd c!s(e)$, and $g \rhd x := e$, respectively, and we use α, $\alpha' \ldots$ when leaving the class of actions unspecified. We use *env* to denote environment, the world outside the system. We define the set of internal signals Sig_{int} as the set of all signals sent by the processes within the system. The set of signals exchanged with environment is denoted as Sig_{ext}. Note that it can be the case that the same signal can come both from the environment and from a process of the system.

Time aspects of a system behaviour are specified by actions dealing with *timers*. A timer is a stopwatch, defined by a local declaration of a process. Each process has the finite set *Timer* of timers with typical elements $t, t_1, \ldots$. A timer can be either *set* to a value, i.e., activated until the system time reaches a certain point, or deactivated by a *reset* action. Actions setting and resetting a timer t are denoted as $g \rhd \mathtt{SET}(e,t)$ and $g \rhd \mathtt{RESET}(t)$, respectively. There is a timeout signal associated with each timer t. Further, we assume T to be the set of timeout signals defined by the specification.

Definition 3.1. [PROCESS SPECIFICATION]
A process specification $Spec_P$ *is a tuple* $(pid, In, Out, Var, Timer, Loc, Act, Edg, l_0)$, *where pid is a unique process identity, In is a finite set of input channel names, Out is a finite set of output channel names, Var is a finite set of variables, Timer is a finite set of timers, Loc is a finite set of locations with an initial location* l_0, *Act is a finite set of actions, and* $Edg \subseteq Loc \times Act \times Loc$ *is a finite set of edges.*

We assume the sets of variables and timers, sets of locations and sets of input channels of processes P_i in a specification to be disjoint. A mapping from variables to values is called a *valuation*; we denote the set of valuations by $Val = \{\phi \mid \phi\colon Var \to D\}$. We write D when leaving the data domain unspecified, and we silently assume all expressions to be well-typed.

Definition 3.2. [SYSTEM SPECIFICATION]
A system specification *Spec is given by a finite set of channels Chan, a finite set of signals Sig and a finite set of process specifications,* $\{Spec_{P_1}, \ldots, Spec_{P_n}\}$ *such that the following conditions hold for all* $j, k = 1..n$:

- $In_j \bigcap In_k = \emptyset$ *if* $j \neq k$;
- $Var_j \bigcap Var_k = \emptyset$ *and* $Timer_j \bigcap Timer_k = \emptyset$ *if* $j \neq k$;
- $Loc_j \bigcap Loc_k = \emptyset$ *if* $j \neq k$;
- $\bigcup_{j=1}^{n} In_j \subseteq Chan_i$ *and* $\bigcup_{j=1}^{n} Out_j \subseteq Chan_o$.

Note, that *Spec* specifies an open system. The channel names used by the environment are not necessarily in $\bigcup_{j=1}^{n} In_j$. Further we refer to the the channels and processes specified by a system specification as *entities*.

3.2.2 SDL Semantics

Here, we define a small step structural operational semantics [131] of a specification in terms of *configurations* and a *transition relation* expressing how a configuration is changed by *one step* of computations. First, we consider the local semantics of separate entities like a process and a channel. The semantics of a process (channel) is an LTS, which is defined with the help of the rules of Table 1 and Table 2 mapping an edge from *Edg* to a transition. Rule EXPIRATION of Table 1 and rules IN, OUT of Table 2 have no edges in premises. Further, we give a definition of n-ary composition that allows to put n processes into communication with each other.

Local semantics

Behaviour of a single process is given by sequences of transitions $\sigma_0 \rightarrow_{\lambda_1} \sigma_1 \rightarrow_{\lambda_2} \ldots$ starting from the initial state.

A process *state*, denoted as σ, consists of an actual process location, a valuation of process variables, a valuation of timers and an input queue of the process. The process starts from the initial location with the default valuation of variables, all timers being deactivated and an empty input queue. The set of process states Σ is a subset of the Cartesian product $Loc \times Val \times TVal \times Q$, where Val denotes the valuations of process variables, $TVal$ denotes the valuations of timers and Q denotes the contents of the input queue.

An input queue receives and holds signals (both timeout and nontimeout) until they are consumed by the process. We write ϵ for the empty queue; $s(pid, v) :: q$ denotes a queue with message $s(pid, v)$ (consisting of a signal s, an identity of a sender pid and a value v) at the head of the queue, i.e., $s(pid, v)$ is the message to be input next; likewise the queue $q :: s(pid, v)$ contains $s(pid, v)$ most recently entered. We use M to denote a set of messages that can be sent in the system, $M = Sig \times Id \times D$. The set of input queue contents is defined as $Q = Seq(M \cup T)$ where $Seq(X)$ denotes the set of all sequences over X.

Definition 3.3. [PROCESS STATE]
A state σ of a process P is a tuple (l, ϕ, θ, q), where l is a location, ϕ is a valuation of process variables, θ is a valuation of timers and q is an input queue of the process. Σ denotes the set of process states.

The *step semantics* is given as a labelled transition relation between states. The labels differentiate between internal τ-steps, *tick*-steps, which represent time progress, and communication steps (either input $c?s(pid, v)$ or output $c!s(pid, v)$) which are labelled by a quadruple of a channel name, a signal, an identity of a sender and a value being transmitted, so $Lab_P = \{\tau,\ tick,\ c_i?s(pid, v),\ c_o!s(pid, v) \mid s \in Sig,\ v \in D,\ pid \in Id\}$.

Depending on location, valuation of process variables, valuation of timers, the possible next actions, and the content of the input queue, the possible successor states are given by the rules of Table 1. In the table, the notation $\phi[x \mapsto v]$ stands for the valuation equalling ϕ for all $y \in Var \backslash \{x\}$ and mapping variable x to the value v.

An *input* of a signal, $l \longrightarrow_{?s(x)} \hat{l} \in Edg$, is enabled if the signal at the head of the queue matches signal s expected by the process. Input $?s(x)$ results in removing the signal $s(v)$ from the head of the queue and updating the valuation ϕ of process variables to $\phi_{[x \mapsto v]}$ (rule INPUT).

Discard is a specific feature of SDL92: if the signal from the head of the queue does not match any input defined as possible for the current input location, then the signal is removed from the queue, and the location, the valuation of process variables and the valuation of timers remain the same (rule DISCARD).

Receiving and sending are communication steps of a process. Denoted by a label $c_i?s(pid, v)$, receiving a signal s with a value v via a channel c leads to adding the message $s(v, pid)$ to the input queue and does not influence process variables, timers and current location (rule RECEIVE).

Output is guarded, so sending a message involves evaluating the guard and the expression to be sent according to the current valuation of variables. It leads to a change of location of the process that sends the message. The message is sent along the channel with name c and the output step is labelled by $c_o!s(pid, v)$ (rule OUTPUT).

An *assignment* $l \longrightarrow_{g \rhd x:=e} \hat{l} \in Edg$ is enabled if the guard g evaluates to *true*. It results in a change of the current location and an update of the valuation $\phi_{[x \mapsto v]}$, where $[\![e]\!]_\phi = v$ (rule ASSIGN).

Modelling time in SDL

In SDL, the concept of timers is employed to specify timing conditions imposed on a system. Two predefined data types, **Time** and **Duration**, are used to specify values related to time. A **Time** value indicates some point of time, whereas a **Duration** value denotes a time delay. A process can access the current system time by means of the `NOW` expression. A valuation $[\![\texttt{NOW}]\!]$ maps this expression to a value of the predefined type **Time** representing current system time. SDL is intended to specify distributed systems with asynchronous communication, so no assumption on the temporal ordering of events in different processes can be based on reading `NOW`.

Each timer is related to a process; a timer is either active (set to a value) or inactive (reset). Two operations are defined on timers: `SET` and `RESET` (rules SET and RESET). A timer can be activated by setting it to a value $on(v)$, where v is the *time* when the timer should expire. The value v is given by an expression

$$\frac{l \longrightarrow_{?s(x)} \hat{l} \in \mathit{Edg} \qquad s \notin T}{(l, \phi, \theta, s(\mathit{pid}, v) :: q) \rightarrow_{\tau} (\hat{l}, \phi[x \mapsto v], \theta, q)}\ \textsc{Input}$$

$$\frac{s' \notin \{s \mid l \longrightarrow_{?s(x)} \hat{l} \in \mathit{Edg}\} \qquad l \in \mathit{Loc}_i \qquad s' \notin T}{(l, \phi, \theta, s'(\mathit{pid}, v) :: q) \rightarrow_{\tau} (l, \phi, \theta, q)}\ \textsc{Discard}$$

$$\frac{v \in D \qquad c \in \mathit{In}}{(l, \phi, \theta, q) \rightarrow_{c_i?s(\mathit{pid}, v)} (l, \phi, \theta, q :: s(\mathit{pid}, v))}\ \textsc{Receive}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, c!(s,e)} \hat{l} \in \mathit{Edg} \qquad \llbracket g \rrbracket_{\phi} = \mathit{true} \qquad \llbracket e \rrbracket_{\phi} = v}{(l, \phi, \theta, q) \rightarrow_{c_o!s(\mathit{pid}, v)} (\hat{l}, \phi, \theta, q)}\ \textsc{Output}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, x := e} \hat{l} \in \mathit{Edg} \qquad \llbracket g \rrbracket_{\phi} = \mathit{true} \qquad \llbracket e \rrbracket_{\phi} = v}{(l, \phi, \theta, q) \rightarrow_{\tau} (\hat{l}, \phi[x \mapsto v], \theta, q)}\ \textsc{Assign}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, \mathtt{SET}(e,t)} \hat{l} \in \mathit{Edg} \qquad \llbracket g \rrbracket_{\phi} = \mathit{true} \qquad \llbracket e \rrbracket_{\phi} = v}{(l, \phi, \theta, q) \rightarrow_{\tau} (\hat{l}, \phi, \theta[t \mapsto \mathit{on}(v)], \pi_t(q))}\ \textsc{Set}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, \mathtt{RESET}(t)} \hat{l} \in \mathit{Edg} \qquad \llbracket g \rrbracket_{\phi} = \mathit{true}}{(l, \phi, \theta, q) \rightarrow_{\tau} (\hat{l}, \phi, \theta[t \mapsto \mathit{off}], \pi_t(q))}\ \textsc{Reset}$$

$$\frac{\llbracket t \rrbracket_{\theta} = \mathit{on}(v) \qquad v \leq \mathbf{now}}{(l, \phi, \theta, q) \rightarrow_{\tau} (l, \phi, \theta[t \mapsto \mathit{off}], q :: t)}\ \textsc{Expiration}$$

$$\frac{l \longrightarrow_{?t} \hat{l} \in \mathit{Edg} \qquad t \in T}{(l, \phi, \theta, t :: q) \rightarrow_{\tau} (\hat{l}, \phi, \theta, q)}\ \textsc{TInput}$$

$$\frac{t' \notin \{t \mid l \longrightarrow_{?t} \hat{l} \in \mathit{Edg}\} \qquad l \in \mathit{Loc}_i \qquad t' \in T}{(l, \phi, \theta, t' :: q) \rightarrow_{\tau} (l, \phi, \theta, q)}\ \textsc{TDiscard}$$

$$\frac{\mathit{blocked}(\sigma)}{\mathbf{now} \rightarrow_{\mathit{tick}} \mathbf{now} + 1}\ \textsc{Tick}_P$$

Table 1. Step semantics of process specification Spec_P

$(\mathtt{NOW}+e)$. A `RESET` action sets the timer to *off*. So the set $TVal$ of valuations of timers is defined as follows: $TVal = \{\theta \mid \theta\colon Timer \to \{off, on(v) \mid v \in \mathbf{Time}\}\}$.

If a `SET` or `RESET` operation is performed on an expired timer after adding the timer signal to the process queue but before the signal is consumed from the queue, the timer signal is removed from the queue. We write $\pi_t(q)$ for the queue obtained from q by projecting out the timeout signal t.

There are a pseudo-signal and an implicit transition, called a timeout transition, associated with each timer. A timer expires when the system time **now** reaches the value to which the timer was set, i.e., the timeout transition of the timer set to $on(v)$ becomes enabled and may occur when **now** is larger than or equal to v. Execution of the timeout transition, captured by the EXPIRATION rule, adds the corresponding pseudo-signal to the process input queue and resets the timer to *off*. The TINPUT rule captures consumption of a timeout signal from the input queue. The TDISCARD is similar to DISCARD.

Rule TICK$_P$ allows time progression by action *tick* that increases the value of system time **now** by 1 and does not change the state of the process. Time can progress only when the process is *blocked*, i.e., it cannot do anything except receiving signals from the outside world. Due to the discarding feature, this implies that the input queue of processes should be empty. This situation is determined by a predicate *blocked* that is *true* if the process is in a state $(l, \phi, \theta, \epsilon)$ and $l \in Loc_i$. None of the other steps can change the system time.

Definition 3.4. [PROCESS P]
A process P is an LTS $(\Sigma \times Time, Lab_P, \rightarrow_\lambda, (\sigma_0, 0), In, Out)$ *where* $\sigma_0 = (l_0, \phi_0, \theta_0, \epsilon)$ *is an initial state,* 0 *is the initial system time, In is a set of input channel names, Out is a set of output channels names and* $\rightarrow \subseteq (\Sigma \times Time) \times Lab \times (\Sigma \times Time)$ *is a labelled transition relation derived by applying the rules in Table 1 to some process specification $Spec_P$.*

We say that a process specification is *well-formed* iff at least one guard evaluates to *true* in each non-input state. At the SDL source language, this assumption corresponds to the natural requirement that each conditional construct must cover all cases, for instance by having at least a default branch. The system should not block because of a non-covered alternative in a decision-construct [133]. In the sequel, we assume that we work *only* with well-formed process specifications.

Channels

In SDL's asynchronous communication model, a process receives messages via channels into a single input port associated with the process. Input ports as well as asynchronous channels are modelled as queues. A channel is represented explicitly by a separate entity consisting of the channel name together with a queue modelling the channel. To allow a uniform presentation of parallel composition below, we use the symbol σ not only for typical element of process states, but also for states (c, q) of queues. Note that timeout signals may appear only in an input queue of a process, but not in a queue modelling a channel. Only nontimeout signals are transferred via channels, so the set of possible channel states of channel c is defined as $\Sigma = \{(c, q) \mid q \in Seq(M)\}$.

Definition 3.5. [CHANNEL STATE]
A state σ of a channel c is a pair (c, q), where c is the name of the channel and q is a FIFO queue.

We require for the input and the output names of a channel that $In_c = \{c_o\}$ and $Out_c = \{c_i\}$. The step semantics of a channel c is given by a labelled transition relation $\rightarrow \subseteq \Sigma \times Lab_c \times \Sigma$ defined by the operational rules IN and OUT of Table 2. Rule IN enables a step $c_o?s(pid, v)$ that adds a message $s(pid, v)$ to the channel queue, whereas rule OUT makes it possible to take a step $c_i!s(pid, v)$ that removes the message $s(pid, v)$ from the channel queue in order to deliver it to a destination process or an environment. The set of labels Lab_c is defined as $\{c_i!s(pid), c_o?s(pid, v) \mid s \in Sig, pid \in Id, v \in D\}$, so the labels marking the communication steps of channels differ from those labelling communication steps of processes.

A channel process may also *tick*, if it can not do anything else except receiving messages. This is possible in case the queue modelling the channel is empty. Otherwise, the channel still can deliver a message to some destination process or environment. So *blocked* is *true* only in state (c, ϵ).

Definition 3.6. [CHANNEL c]
A channel process c is an LTS $S = (\Sigma \times Time, Lab_c, \rightarrow, (\sigma_0, 0), In_c, Out_c)$, *where $\sigma_0 = (c, \epsilon)$ is an initial state, 0 is the initial system time, and a transition relation $\rightarrow \subseteq (\Sigma \times Time) \times Lab_c \times (\Sigma \times Time)$ that is defined by the rules in Table 2.*

Time in SDL

We use an interpretation of time progression supported by the commercial SDL design-tools [166, 156]. It regards transitions as instantaneous, i.e., taking zero-time. Time is allowed to pass when SDL processes are in an idle state

$$\frac{}{(c, s(pid, v) :: q) \rightarrow_{c_i!s(pid,v)} (c, q)}\ \textsc{Out} \qquad \frac{}{(c, q) \rightarrow_{c_o?s(pid,v)} (c, q :: s(pid, v))}\ \textsc{In}$$

$$\frac{blocked(c, q)}{\mathbf{now} \rightarrow_{tick} \mathbf{now} + 1}\ \textsc{Tick}_Q$$

Table 2. Step semantics for channel c

and waiting for further signals to arrive, i.e., the input ports are empty. Time proceeds until an active timer expires or a process receives a signal from the environment, i.e., time progress has *least priority*. We refer to a time period between two *tick*-steps as a *time slice*. When the system time becomes equal to a timer value, the timeout transition becomes enabled and can be executed at any point of the time slice. A time slice starts with firing some enabled timeout transition or an input from the environment. This action unblocks the system. In case several timeout transitions become enabled at the same time, one of them is taken non-deterministically to unblock the system, and the rest is taken later at any point of the time slice since they have the same priority as normal transitions. In case none of timeout transition is enabled and there are no inputs from the environment, the system is still blocked and time can progress further.

We say that a system is *blocked* if it can only wait for signals to arrive from the outside world. We use the predicate *blocked* to determine whether the system is in this state. The system time is allowed to pass if all entities of the system are blocked. This interpretation conforms to the interpretation of time given by the dynamic semantics of SDL [145, 146]. The dynamic semantics states that *global system time* is represented by a function *clock* whose value increases monotonically and does not increase as long as a *signal is in transit* on a channel. Moreover, it also fits into the interpretation of time given in Section 11.12.1 of SDL standard Z.100 [140, 142].

In SDL, real numbers are used for **Time** and **Duration**. So, whenever a timer is set, its expiration is given by a real number. Every tick increases system time with a certain amount, so the system time proceeds in a discrete manner. Therefore, we may use a discrete approach to the interpretation of **Time** values, i.e., $\mathbb{N}$ and $\mathbb{Z}$ are used as the interpretations of **Time** and **Duration**, respectively. We assume a global system time represented by a system variable **now** that has a value of the type **Time**. A *tick*-step, which increases the value of this variable, is enabled only if the system is blocked.

Though complicated, such a time semantics is suitable for implementation purposes [133]. It is natural to model a timer as a unit advancing from the current moment of time derived by the evaluation of the `NOW` expression to the time point specified by the expression ($\texttt{NOW} + \delta$), where δ is time left until the expiration of the timer. Expression *NOW* always evaluates to the system time, i.e., $[\![NOW]\!] = \mathbf{now}$.

n-ary parallel composition

A global semantics of an SDL specification is given by an n-ary parallel composition of processes and channels defined by the specification.

The semantics of parallel composition of n processes is given by the rules of Table 3. We call a vector of states of n system entities (processes and channels) a *configuration* and write $\gamma, \gamma_1, \ldots \in \Gamma$ for typical elements. We call a sequence

$$\frac{(\sigma_j, \mathbf{now}) \rightarrow_{\alpha_j} (\hat{\sigma}_j, \mathbf{now}) \qquad (\sigma_k, \mathbf{now}) \rightarrow_{\alpha_k} (\hat{\sigma}_k, \mathbf{now}) \qquad comm(\alpha_j, \alpha_k)}{(\dots, \sigma_j, \dots, \sigma_k, \dots, \mathbf{now}) \rightarrow_{\tau} (\dots, \hat{\sigma}_j, \dots, \hat{\sigma}_k, \dots, \mathbf{now})}\ \textsc{Comm}$$

$$\frac{(\sigma_i, \mathbf{now}) \rightarrow_{c_o?s(env,v)} (\hat{\sigma}_i, \mathbf{now})}{(\dots, \sigma_i, \dots, \mathbf{now}) \rightarrow_{c_o?s(env,v)} (\dots, \hat{\sigma}_i, \dots, \mathbf{now})}\ \textsc{Interleave}_{in}$$

$$\frac{(\sigma_i, \mathbf{now}) \rightarrow_{c_i!s(pid,v)} (\hat{\sigma}_i, \mathbf{now}) \qquad c \in In_{env}}{(\dots, \sigma_i, \dots, \mathbf{now}) \rightarrow_{c_i!s(pid,v)} (\dots, \hat{\sigma}_i, \dots, \mathbf{now})}\ \textsc{Interleave}_{out}$$

$$\frac{(\sigma_i, \mathbf{now}) \rightarrow_{\tau} (\hat{\sigma}_i, \mathbf{now})}{(\dots, \sigma_i, \dots, \mathbf{now}) \rightarrow_{\tau} (\dots, \hat{\sigma}_i, \dots, \mathbf{now})}\ \textsc{Interleave}_{\tau}$$

$$\frac{(\sigma_1, \mathbf{now}) \rightarrow_{tick} (\sigma_1, \mathbf{now}+1) \qquad \dots \qquad (\sigma_n, \mathbf{now}) \rightarrow_{tick} (\sigma_n, \mathbf{now}+1)}{(\sigma_1, \dots, \sigma_n, \mathbf{now}) \rightarrow_{tick} (\sigma_1, \dots, \sigma_n, \mathbf{now}+1)}\ \textsc{Tick}$$

Table 3. n-ary parallel composition

of configurations $\gamma_0 \rightarrow_{\lambda_0} \gamma_1 \rightarrow_{\lambda_2} \dots$ starting from an initial configuration a *run*.

Communication between two system entities is performed by exchanging a common signal and a value over a channel name (c_o or c_i). According to the syntactic restriction on the use of input and output channel names, only synchronization of communication steps between a process and a queue may happen. Sending of a message over the channel consists in synchronizing an output step of the process with an input step that adds the message into the channel queue, i.e., a $c_o!s(pid, v)$-step of the process should be synchronized with a $c_o?s(pid, v)$-step of the channel c. Receiving a message consists in synchronizing an output step that removes the first element from the channel queue with a receiving step that adds the message into the input queue of the process, i.e., a $c_i!s(pid, v)$-step of the channel should be synchronized with a $c_i?s(pid, v)$-step of the destination process. A predicate *comm*, which is *true* when communication steps should be synchronized, is defined as follows:

Definition 3.7.
For two labels α_1 and α_2, the predicate comm is true iff one of the following conditions is satisfied for $j, k \in \{1, 2\}$, $j \neq k$:

$$\textit{(i)}\ \alpha_j = c_o!s(pid, v),\ \alpha_k = c_o?s(pid, v)$$

$$\textit{(ii)}\ \alpha_j = c_i!s(pid, v),\ \alpha_k = c_i?s(pid, v)$$

Otherwise the predicate is false.

The initial configuration of an n-ary parallel composition is a vector of initial configurations of n system entities (processes and channels), i.e., $\gamma_0 = (\gamma_0^1, \ldots, \gamma_0^n)$. As it is defined by the rule COMM, two common steps are glued and relabelled to a τ-step by the synchronization. Inputs and outputs from the environment are interleaved by rules INTERLEAVE$_{in}$ and INTERLEAVE$_{out}$, respectively. As far as τ-steps are concerned, each system entity can act on its own according to the rule INTERLEAVE$_\tau$.

The rule TICK states that time progresses if the system is blocked. A time progression step *tick* increases the system variable **now** modelling system time. Expression *NOW* always evaluates to the system time, i.e., $[\![NOW]\!] = \mathbf{now}$.

Definition 3.8. [PARALLEL COMPOSITION]
The n-ary parallel composition *of n system entities* $S_k = (\Sigma_k, Lab_k, \rightarrow^k, In_k, Out_k, \sigma_0^k)$ *is a LTS* $S = (\Gamma, Lab, \rightarrow, In, Out, \gamma_0)$, *where*

- $\Gamma = \Sigma_1 \times \ldots \times \Sigma_n \times Time$ *is a set of states with an initial state* $\gamma_0 = (\sigma_0^1, \ldots, \sigma_0^n)$ *and projections* $f_k : \Gamma \rightarrow \Gamma_k$, $k \in \{1, \ldots, n\}$*;*
- $Lab = \bigcup_{k=1}^{n} Lab_k$ *is a finite set of labels;*
- $In = \bigcup_{k=1}^{n} In_k$ *is a finite set of input channel names;*
- $Out = \bigcup_{k=1}^{n} Out_k$ *is a finite set of output channel names;*
- $\rightarrow \subseteq (\Gamma \times Time) \times Lab \times (\Gamma \times Time)$ *is a labelled transition relation given by the rules of Table 3.*

The following lemma expresses that the blocked predicate is compositional in the sense that the parallel composition of processes is blocked iff each process is blocked and there are no messages in transit on channels and no messages in input queues.

Lemma 3.1.
For a configuration γ*, blocked*(γ) *iff blocked*(σ) *for all states* σ *that are a part of* γ*.*

Proof. If γ is not blocked, it can perform an output step or a τ-step. The output step must originate from a process, which is not blocked or from a channel which contains a signal in transit in this case.

The τ-step is either caused by a single process or by a synchronizing action of a process and a channel; in both cases at least one entity is not blocked.

For the reverse direction, a τ-step of a single process being thus not blocked, entails that γ is not blocked.

An output step of a single process or a channel causes γ either to do the same output step or, in case of internal communication, to do a τ-step. In both cases, γ is not blocked. □

3.3 Timer Transformation

A state of an SDL process is given by its current location, the valuations of timers and variables, and the content of the input queue. There are several reasons why the way of modelling timers in SDL is not quite suitable for model checking purposes. Since `NOW` gives access to the current system time, executing $\texttt{SET}(\texttt{NOW}+e,t)$ with different (infinitely growing) values of `NOW`, we get different process configurations. Moreover, a timeout transition can add a timer signal at any point of the time slice. This blows up the state space due to the number of possible interleaving sequences of events. Furthermore, keeping a timeout signal in a process queue adds to the length of the state vector. In this section, we solve this problem by a transformation replacing the SDL concept of timers with a new one.

The transformation of process specifications is given by the rules of Table 4. A new syntax is introduced for setting and resetting timer variables. Note that the transformation rules are developed under the assumption that the `NOW` operator appears in the original system specification in the scope of `SET` operations only, and all the `SET` operations are of the form $\texttt{SET}(\texttt{NOW}+\delta,t)$. In the sequel, we consider only systems of this type.

To avoid the state explosion due to the interpretation of timers and the overhead caused by the management of timeout signals, we substitute the SDL concept of timeouts as a special kind of signals by a concept of timeouts as guards. A declaration of a timer t is transformed to the declaration of a timer variable t. We use *off* to represent inactive timers. The value of a timer variable representing an active timer shows the *delay* left until timer expiration.

$$\frac{l \longrightarrow_{?t} \hat{l} \in Edg \qquad t \in T}{l \longrightarrow_{g_t \rhd\, reset\ t} \hat{l} \in Edg'} \quad \textsc{TInput to Timeout}$$

$$\frac{l \longrightarrow_{g \rhd \texttt{SET(NOW+}e,t\texttt{)}} \hat{l} \in Edg}{l \longrightarrow_{[g \bigwedge (e\ \geq\ 0)] \rhd\, set\ t:=e} \hat{l} \in Edg'} \quad \textsc{Set to Set}_1$$

$$\frac{l \longrightarrow_{g \rhd \texttt{SET(NOW+}e,t\texttt{)}} \hat{l} \in Edg}{l \longrightarrow_{[g \bigwedge (e<0)] \rhd\, set\ t:=0} \hat{l} \in Edg'} \quad \textsc{Set to Set}_2$$

$$\frac{l \longrightarrow_{g \rhd \texttt{RESET(}t\texttt{)}} \hat{l} \in Edg'}{l \longrightarrow_{g \rhd\, reset\ t} \hat{l} \in Edg'} \quad \textsc{Reset to Reset}$$

Table 4. Transformation rules

We use the integer domain $\mathbb{Z}$ as a natural interpretation of **Duration**, since the delay specified by an expression e in a setting action $\texttt{SET}(\texttt{NOW} + e, t)$ may take positive and negative values. Setting a timer to a time value less than the actual system time results in an immediate expiration of the timer and adding a timeout signal to the input queue of the process. In this case, immediate means that the expiration must take place within the current time slice. Therefore, an action $g \rhd \texttt{SET}(\texttt{NOW}+e, t)$ on timers is substituted by the choice between setting timer variable t to the value of the expression e and setting t to zero. The first action is allowed if the expression has a nonnegative value (rule SET TO SET$_1$ of Table 4). Otherwise, the second action is enabled and the timer variable is set to zero (rule SET TO SET$_2$ of Table 4). The $\texttt{RESET}(t)$ action is transformed into an assignment of the *off* value to the timer variable t, denoted by *reset* t (rule RESET TO RESET of Table 4).

In the original system, a timer whose value is larger than or equal to the current system time may expire. The transformed system should demonstrate the same behaviour. Since we suppose the value of a transformed timer to be the delay left until its expiration, only timers whose values are equal to 0 may expire. Therefore, we replace each input of a timeout signal t by resetting a timer t that is guarded by the *timeout guard* g_t, namely, $(t = on(0))$. Resetting guarantees deactivation of the timer (rule TINPUT TO TIMEOUT of Table 4). The set of actions of the transformed process coincides with the set of actions of the original one, except for SET and RESET actions on timers that are substituted by *set* and *reset* actions on timer variables.

Applying the rules of Table 4 to a process specification $Spec_P$, we get the process specification $Spec_{P'}$. Given $Spec_P = (pid,\ In,\ Out,\ Var,\ Timer,\ Loc,\ Act,\ Edg,\ l_0)$, we get $Spec_{P'} = (pid,\ In,\ Out,\ Var',\ Loc,\ Act',\ Edg',\ l_0)$, where $Var' = Var \cup \{t \mid t \in Timer\}$. Settings, timeouts and resettings of timers are substituted by settings, timeouts and resettings of timer variables according to the rules of Table 4. The other actions and edges are left unmodified. Further we refer to the set of timer variables as $TVar$.

Table 5 gives the local step semantics of the transformed process specification. Further, we refer to an LTS derived from some transformed process specification $Spec_{P'}$ by the rules of Table 5 as a transformed process P'. A state of a transformed process is given by a location, a valuation of process and timer variables and a process input queue. A valuation of process and timer variables is denoted as η. Since timeout signals are not put into an input queue anymore, we do not distinguish between timeout and non-timeout signals in rule INPUT. Setting a timer and resetting a timer do not influence the input queue of a process (rules SET and RESET). The set Σ' of process states is defined by the Cartesian product $Loc \times Val \times Seq(M)$.

Definition 3.9. [STATE OF THE TRANSFORMED PROCESS]
A state σ' of a process P' is a triple (l, η, q), where l is a location, η is a valuation of variables and q is a content of the input queue of the process. Σ' denotes the set of states of a transformed process.

$$\frac{l \longrightarrow_{?s(x)} \hat{l} \in \mathit{Edg}'}{(l, \eta, s(pid, v) :: q) \rightarrow_{\tau} (\hat{l}, \eta_{[x \mapsto v]}, q)}\ \textsc{Input}$$

$$\frac{s' \not\in \{s \mid l \longrightarrow_{?s(x)} \hat{l} \in \mathit{Edg}\} \qquad l \in \mathit{Loc}_i}{(l, \phi, \theta, s'(pid, v) :: q) \rightarrow_{\tau} (l, \phi, \theta, q)}\ \textsc{Discard}$$

$$\frac{v \in D \qquad c \in \mathit{In}_P}{(l, \eta, q) \rightarrow_{c_i?s(pid,v)} (l, \eta, q :: s(pid, v))}\ \textsc{Receive}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, c!(s,e)} \hat{l} \in \mathit{Edg}' \qquad [\![g]\!]_{\eta} = \mathit{true} \qquad [\![e]\!]_{\eta} = v}{(l, \eta, q) \rightarrow_{c_o!s(pid,v)} (\hat{l}, \eta, q)}\ \textsc{Output}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in \mathit{Edg}' \qquad [\![g]\!]_{\eta} = \mathit{true} \qquad [\![e]\!]_{\eta} = v}{(l, \eta, q) \rightarrow_{\tau} (\hat{l}, \eta_{[x \mapsto v]}, q)}\ \textsc{Assign}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, set\ t:=e} \hat{l} \in \mathit{Edg}' \qquad [\![g]\!]_{\eta} = \mathit{true} \qquad [\![e]\!]_{\eta} = v}{(l, \eta, q) \rightarrow_{\tau} (\hat{l}, \eta_{[t \mapsto on(v)]}, q)}\ \textsc{Set}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, reset\ t} \hat{l} \in \mathit{Edg}' \qquad [\![g]\!]_{\eta} = \mathit{true}}{(l, \eta, q) \rightarrow_{\tau} (\hat{l}, \eta_{[t \mapsto \mathit{off}]}, q)}\ \textsc{Reset}$$

$$\frac{l \longrightarrow_{g_t \,\triangleright\, reset\ t} \hat{l} \in \mathit{Edg}' \qquad [\![t]\!]_{\eta} = on(0)}{(l, \eta, q) \rightarrow_{\tau} (\hat{l}, \eta_{[t \mapsto \mathit{off}]}, q)}\ \textsc{Timeout}$$

$$\frac{t' \not\in \{t \mid l \rightarrow_{g_t \,\triangleright\, reset\ t} \hat{l} \in \mathit{Edg}'\} \qquad [\![t']\!]_{\eta} = on(0) \qquad l \in \mathit{Loc}_i}{(l, \eta, q) \rightarrow_{\tau} (l, \eta_{[t' \mapsto \mathit{off}]}, q)}\ \text{T}\textsc{Discard}$$

$$\frac{\mathit{blocked}(\sigma)}{(l, \eta, q) \rightarrow_{\mathit{tick}} (l, \eta_{dec}, q)}\ \textsc{Tick}_P$$

Table 5. Step semantics of transformed process specification $\mathit{Spec}_{P'}$

$$\frac{}{(c, \epsilon) \rightarrow_{tick} (c, \epsilon)}\ \textsc{Tick}_c$$

$$\frac{\sigma_j \rightarrow_{\alpha_j} \hat{\sigma}_j \qquad \sigma_k \rightarrow_{\alpha_k} \hat{\sigma}_k \qquad j \neq k \qquad comm(\alpha_j, \alpha_k)}{(\ldots, \sigma_j, \ldots, \sigma_k, \ldots) \rightarrow_{\tau} (\ldots, \hat{\sigma}_j, \ldots, \hat{\sigma}_k, \ldots)}\ \textsc{Comm}$$

$$\frac{\sigma_i \rightarrow_{c_o?s(env,v)} \hat{\sigma}_i}{(\ldots, \sigma_i, \ldots) \rightarrow_{c_o?s(env,v)} (\ldots, \hat{\sigma}_i, \ldots)}\ \textsc{Interleave}_{in}$$

$$\frac{\sigma_i \rightarrow_{c_i!s(pid,v)} \hat{\sigma}_i \qquad c \in In_{env}}{(\ldots, \sigma_i, \ldots) \rightarrow_{c_i!s(pid,v)} (\ldots, \hat{\sigma}_i, \ldots)}\ \textsc{Interleave}_{out}$$

$$\frac{\sigma_i \rightarrow_{\tau} \hat{\sigma}_i}{(\ldots, \sigma_i, \ldots) \rightarrow_{\tau} (\ldots, \hat{\sigma}_i, \ldots)}\ \textsc{Interleave}_{\tau}$$

$$\frac{\sigma_1 \rightarrow_{tick} \hat{\sigma}_1 \qquad \ldots \qquad \sigma_n \rightarrow_{tick} \hat{\sigma}_n}{(\sigma_1, \ldots, \sigma_n) \rightarrow_{tick} (\hat{\sigma}_1, \ldots, \hat{\sigma}_n)}\ \textsc{Tick}$$

Table 6. *tick*-step of a channel and n-ary parallel composition

Definition 3.10. [TRANSFORMED PROCESS]
A transformed process P' is an LTS $(\Sigma', Lab', \rightarrow', In, Out, \sigma'_0)$ *where* $\sigma'_0 = (l_0, \eta_0, \epsilon)$ *is the initial state and* $\rightarrow' \subseteq \Sigma' \times Lab' \times \Sigma'$ *is a labelled transition relation derived by applying the rules in Table 5 to some transformed process specification* $Spec_{P'}$.

Note that the *system time is not present* in the transformed system explicitly — one infinitely growing variable would be enough to cause state explosion. Instead of increasing the system time, the tick transition (rule TICK_P of Table 5) decreases the values of timer variables. Like the TICK transition of the original system, this transition can take place only if the system is blocked, and "blocked" has exactly the same meaning as before. The *dec* operation decreases all the positive values of timer variables by one and leaves the other variables unchanged. The evaluation obtained by applying the *dec* operation is denoted η_{dec}.

Table 6 defines n-ary parallel composition of channels and transformed processes. Here, we need to define a *tick*-step not only for processes but also for channels. (Note that IN and OUT rules for channels in the transformed system coincide with the same rules in Table 2.) A *tick*-step of a channel does not change the state of the channel and becomes enabled only if the channel is blocked, i.e. it has no message to deliver and it may receive only messages from the environment (rule TICK_c of Table 6). The definition of n-ary parallel composition for transformed systems coincides with the one for original systems except the TICK rule. According to the TICK rule of Table 6, all components of the transformed system are synchronized on their *tick*-steps decreasing values of active timer variables.

3.4 Model Equivalence

The goal of the transformation described in Section 3.3 is to overcome state explosion caused by the traditional interpretation of timers and time in SDL. In this section, we will show that the results of the verification of the transformed system are transferable to the original one. Namely, we show that there is a branching simulation relation (see Def. 2.19) relating the original system to its transformation. The branching simulation relation guarantees that the transformed system contains at least the behaviour of the original system. We also show that there is a weak trace inclusion relation (see Def. 2.17) relating the transformed system to the original one. We also show that that the original system and the transformed one are path equivalent up to stuttering (see Def. 2.28).

The transformed system does not give a straightforward reflection of the original system behaviour. While actions that are not related to timers are left unchanged, sendings of timeout signals are projected out, and consumption of timeout signals from the process queue are mimicked by the corresponding TIMEOUT, whose enabling conditions are guaranteed to be true in this case.

Such a projection is not harmful from verification point of view, because not the presence or absence of a timeout signal but the consumption of it and the choice of the following actions are important. The same concerns process queues: by saying that the content of some queue in the transformed system is the same as the content of the corresponding queue in the original system, we mean that the projections of the queues on *Sig* coincide (note that *Sig* contains only nontimeout signals). The main requirement imposed on configurations is that the valuations of variables should be equal. Since the sets of process variables are disjoint, we may use $[\![x]\!]_\gamma$ to denote the valuation of variable x in configuration γ. Further, we define a relation $\approx$ on configurations.

Definition 3.11. [RELATION $\approx$]
We write $\gamma \approx \gamma'$ *iff* $[\![x]\!]_\gamma = [\![x]\!]_{\gamma'}$ *for all process variables* x.

Since a configuration is defined as a parallel composition of one or more local states of processes and channels, the definition of $\approx$ on configurations is defined analogously.

The transformation ought to preserve timing aspects of the behaviour of the original system. In order to guarantee this, the timers of the original system should be related to the timer variables of the transformed one in such a way that whenever a timeout is possible in the original system, it is also enabled in the transformed one. It means that if a timer expires in the original system, the timer variable representing the timer should carry the value zero. Further, we define a relation $\approx^\star$, that relates timing and input/output aspects of system configurations. The relation connects a configuration of the original system with a configuration of the transformed one:

- the process variables have the same values;
- both systems have the same input possibilities wrt. nontimeout signals;
- timeouts enabled in the original system are also enabled in the transformed one;
- the transformed system can establish the same communication steps as the original one, i.e., the queues representing channels of the transformed system have the same contents as the corresponding queues of the original system.

Definition 3.12. [RELATION $\approx^\star$ ON STATES]
Let $Spec_P$ *be a process specification and* $Spec_{P'}$ *be the process specification obtained from* $Spec_P$ *by applying the rules from Table 4. Let* P *and* P' *be processes derived from* $Spec_P$ *and* $Spec_{P'}$ *by applying rules of Table 1 and Table 5, respectively. Let* $\sigma = (l, \phi, \theta, q)$ *and* $\sigma' = (l, \eta, q')$ *be states of* P *and* P'*, respectively. Let* **now** *be the system time related to process* P*. We write* $(\sigma, \mathbf{now}) \approx^\star \sigma'$ *iff the following conditions are satisfied:*

1. $\sigma \approx \sigma'$;
2. $q' = \pi_T(q)$, *where* $\pi_T(q)$ *is obtained from* q *by projecting out the timeout signals from* T;

3. *for all $t \in TVar$, $t \in Timer$: if $[\![t]\!]_\theta = on(v)$ and $[\![t]\!]_\eta = on(w)$, then $w + \mathbf{now} = \max\{\mathbf{now}, v\}$;*
4. *for all $t \in TVar$, $t \in Timer$: if $[\![t]\!]_\theta = off$ and the timeout signal t is not in q, then $[\![t]\!]_\eta = off$;*
5. *for all $t \in TVar$, $t \in Timer$: if $[\![t]\!]_\theta = off$ and the timeout signal t is in q, then $[\![t]\!]_\eta = on(0)$.*

*Let **now** be the system time related to channel c. For channel states (c, q) and (c, q'), we write $((c, q), \mathbf{now}) \approx^\star (c, q')$ iff $q = q'$.*

Now consider a specification *Spec* that consists of n components (channels and processes) and the specification *Spec′* obtained from *Spec* by applying the transformation rules of Table 4. It is straightforward that the specification *Spec′* specifies n components as well. Let S be the LTS derived from *Spec* by applying rules from Tables 1, 2, 3, and S' be the LTS derived from *Spec′* by applying rules from Tables 5, 6. Suppose $\gamma = (\sigma_1, \ldots, \sigma_i, \ldots, \sigma_n)$ is a configuration of S consisting of n entities (processes and channels), and let $\gamma' = (\sigma'_1 \ldots, \sigma'_i, \ldots, \sigma'_n)$ be a configuration of S'. Moreover, σ_i denotes a state of P_i and σ'_i a state of P'_i.

Definition 3.13. [RELATION $\approx^\star$ ON CONFIGURATIONS]
*We write $(\gamma, \mathbf{now}) \approx^\star \gamma'$ for configuration γ of LTS S and system time **now** related to S and configuration γ' of S' iff $(\sigma_i, \mathbf{now}) \approx^\star \sigma'_i$ for all $i \in \{1, \ldots, n\}$.*

Lemma 3.2.
Let γ and γ' be configurations of S and S' respectively, $(\gamma, \mathbf{now}) \approx^\star \gamma'$ and e be a non-timed expression. Then $[\![e]\!]_\gamma = v$ iff $[\![e]\!]_{\gamma'} = v$.

Proof. The process variables have the same values both in γ and γ' by Def. 3.11, and hence the non-timed expression e has the same value v both in γ and in γ'. □

Lemma 3.3.
*Let γ and γ' be configurations of S and S' respectively, **now** be the system time of S and $(\gamma, \mathbf{now}) \approx^\star \gamma'$. Then blocked($\gamma$) iff blocked($\gamma'$).*

Proof.
blocked(γ) $\Longrightarrow$ *blocked*(γ')
If S is blocked in configuration γ, then (i) all queues (modelling both channels and input ports) are empty, as otherwise communication or input steps might take place (rules DISCARD, INPUT, TDISCARD, TINPUT and RECEIVE of Table 1 and rules OUT and IN of Table 2); (ii) guards of guarded steps are *false*, as otherwise there would be a guarded step possible (rules NONEINPUT, OUTPUT, ASSIGN, SET, RESET of Table 1); (iii) there are no active timers ready to expire in S, as otherwise an expiration step would be enabled (rule EXPIRATION of Table 1).

According to Def. 3.12, queues in γ' are projections of corresponding queues in γ on timeout signals. Since all queues in γ are empty, the queues in γ' are empty as well and no input or communication step is possible in S'.

There is no timer in γ that satisfies condition 5 of Def. 3.12 and so there is no timer variable in γ' having a zero value, as otherwise, $(\gamma, \mathbf{now})$ and γ' are not related by $\approx^\star$. Therefore, no timeout step is possible in S'.

According to Lemma 3.2, untimed guards that are *false* in γ are also *false* in γ'. Since all guards in S valuate to *false*, there is no guarded step enabled in the configuration γ' of S'.

Communication, input, discard, timeout, guarded steps are disabled in γ', so the only steps that may happen are inputs from the environment and time progression. Therefore, $\mathit{blocked}(\gamma')$ is *true*.

$\mathit{blocked}(\gamma') \implies \mathit{blocked}(\gamma)$
If the transformed system is blocked, then (i) all queues of S' are empty, as otherwise an internal communication step or an input would be enabled (rules Discard, Input, Receive of Table 5 and rules Out and In of Table 2); (ii) none of the timer variables of S' has the value zero, as otherwise a timeout or a discard of a timeout would be enabled (rules Timeout and TDiscard of Table 5); (iii) none of the non-timeout conditions imposed on guarded steps in S' is valuated to *true*, as otherwise one of the guarded steps would be enabled (rules Output, Assign, Set, Reset of Table 5).

Since all queues in γ' are empty and none of the time variables has the value zero, all queues are also empty in γ, according to conditions 2 and 5 of Def. 3.12. All nontimeout guards that are *false* in γ' are also *false* in γ by Lemma 3.2. So all guarded steps possible in γ are disabled.

Since all queues are empty and none of the guarded steps is enabled in γ, the only steps that may happen are inputs from the environment and time progression, i.e., $\mathit{blocked}(\gamma)$ is *true*. □

To show that the transformed system shows at least the behaviour of the original one, we demonstrate that the relation $\approx^\star$ on their configurations is a branching simulation relation (Def. 2.19). To prove that $\approx^\star$ is a branching simulation relation on the configurations of S and S', we first check this relation on the rules of Table 1 and Table 5 for a process P of the system S and its counterpart P' in the system S'. Then, we proceed similarly by the case analysis on the rules of Table 2. Finally, we check the relation on rules for n-ary parallel composition of Table 3.

Lemma 3.4.
Let $Spec_P$ be a process specification and $Spec_{P'}$ be the process specification obtained from $Spec_P$ by applying rules from Table 4. Let P and P' be processes derived from $Spec_P$ and $Spec_{P'}$ by applying rules of Table 1 and Table 5, respectively. Then there exists a relation $R \subseteq (\Sigma \times Time) \times \Sigma'$ such that $(\sigma, \mathbf{now}) R \sigma'$ implies $(\sigma, \mathbf{now}) \approx^\star \sigma'$ and R is a branching simulation.

Proof. Let σ_0 be the initial configuration of the process P, σ_0' be the initial configuration of the process P' and **now** be the system time of P. The transformation rules defined in Table 4 do not change initial locations of system entities. Application of transformation rules does not modify default valuation of process variables. Initially, the timers are deactivated and the input queue is empty in P, thus the queues are empty and the timer variables are *off* for P'. The condition of Def. 3.11 and the conditions of Def. 3.12 are satisfied and $(\sigma_0, \mathbf{now}) \approx^\star \sigma_0'$ holds for the initial states.

Now assume that $(\sigma, \mathbf{now}) \approx^\star \sigma'$ holds for some σ, σ' and **now**. To show that $P \preceq_{br} P'$ we proceed with a case analysis on the rules of Table 1.

Case: INPUT
Let transition $(l, \phi, \theta, s(pid, v) :: q) \rightarrow_\tau (\hat{l}, \phi[x \mapsto v], \theta, q)$ be enabled in P. Let $((l, \phi, \theta,\ s(pid, v) :: q), \mathbf{now}) \approx^\star \sigma'$. According to rule INPUT of Table 1, there exists an edge $l \longrightarrow_{?s(x)} \hat{l} \in Edg$. By the transformation rules of Table 4, there is an edge $l \longrightarrow_{?s(x)} \hat{l} \in Edg'$.

Since $(\sigma, \mathbf{now}) \approx^\star \sigma'$, the input queue of P' is a projection of $s(pid, v) :: q$ on timeout signals, $s(pid, v)$ is the head element of the queue and transition $(l, \eta, s(pid, v) :: q') \rightarrow_\tau (\hat{l}, \eta[x \mapsto v], q')$ is enabled in P' (rule INPUT of Table 5).

The input of P can be mimicked by the input of P'. The valuation of variable x is changed to v in both systems. Timers, timer variables and system time are not influenced in this case. q' is the projection of q on *Sig*. So the conditions of Def. 3.12 are satisfied and $((\hat{l}, \phi[x \mapsto v], \theta, q), \mathbf{now}) \approx^\star (\hat{l}, \eta[x \mapsto v], q')$ holds. Input steps are labelled by τ in both systems, so condition 1 of Def. 2.19 is satisfied.

Case: DISCARD
Let transition $(l, \phi, \theta, s(pid, v) :: q) \rightarrow_\tau (l, \phi, \theta, q)$ be enabled in P. We also assume that $((l, \phi, \theta, s(pid, v) :: q), \mathbf{now}) \approx^\star \sigma'$.

Rule DISCARD of Table 1 shows that signal s is not expected by process P in location l. According to the transformation rules of Table 4, $l \longrightarrow_{?s(x)} \hat{l} \notin Edg'$. So, the discard in P can be mimicked by a discard in P' (rule DISCARD of Table 5). Further, this case is analogous to the case INPUT and $(\hat{\sigma}, \mathbf{now}) \approx^\star \hat{\sigma}'$ for the resulting states. Moreover, the condition 1 of Def. 2.19 is satisfied.

Case: RECEIVE
A receiving step $(l, \phi, \theta, q) \rightarrow_{c_i?s(pid,v)} (l, \phi, \theta, q :: s(pid, v))$ of P (rule RECEIVE of Table 1) can always be mimicked by the receiving step $(l, \eta, q') \rightarrow_{c_i?(s,v)} (l, \eta, q' :: s(pid, v))$ of P' (rule RECEIVE of Table 5). Both the step of the original system and the mimicking step of the transformed system add $s(pid, v)$ to the input queue. So $q' :: s(pid, v)$ is the projection of $q :: s(pid, v)$ to the timeout signals. Nothing else is changed by the steps. Therefore, $((l,\ \phi,\ \theta,\ q :: s(pid, v),$ $\mathbf{now}) \approx^\star (l,\ \eta,\ q' :: s(pid, v)$ holds for the resulting states. Both steps are labelled by $c_i?s(pid, v)$, so condition 1 of Def. 2.19 holds.

Case: OUTPUT
Let $(l, \phi, \theta, q), \mathbf{now}) \approx^\star (l, \eta, q')$. Let $(l, \phi, \theta, q) \rightarrow_{c_o!s(pid,v)} (\hat{l}, \phi, \theta, q)$ be a step

of P. By rule OUTPUT of Table 1, we get $l \longrightarrow_{g \,\triangleright\, c!(s,e)} \hat{l} \in \mathit{Edg}$. According to the transformation rules of Table 4, $l \longrightarrow_{g \,\triangleright\, c!(s,e)} \hat{l} \in \mathit{Edg}'$.

The guard g, which valuates to *true* in σ, also valuates to *true* in σ' by Lemma 3.2. According to the rule OUTPUT of Table 5, this edge is mapped to the step $(l, \eta, q) \rightarrow_{c_o!s(pid,v)} (\hat{l}, \eta, q)$ of P'. Both the step of the original system and the mimicking step of the transformed one change the location of the process only. Therefore, $((\hat{l}, \phi, \theta, q), \mathbf{now}) \approx^{\star} (\hat{l}, \eta, q)$ for the resulting states. Moreover, both steps are labelled by $c_o!s(pid, v)$, so condition 1 of Def. 2.19 is satisfied.

Case: ASSIGN
Let $(l, \phi, \theta, q) \rightarrow_{\tau} (\hat{l}, \phi[x \mapsto v], \theta, q)$ in P and $(l, \phi, \theta, q), \mathbf{now}) \approx^{\star} (l, \eta, q')$. By rule ASSIGN, $l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in \mathit{Edg}$. If this edge is present in Spec_P, then it is also available in $\mathit{Spec}_{P'}$ by the transformation rules of Table 4.

The guard g, which valuates to *true* in σ, also valuates to *true* in σ' by Lemma 3.2. An expression e having a value v in σ has the same value in σ'. So the assignment is mapped to the $(l, \eta, q') \rightarrow_{\tau} (\hat{l}, \eta[x \mapsto v], q')$-step of P' (rule ASSIGN of Table 5).

The value of the variable x is changed to v both in the original system and in the transformed one. Neither timers, nor queues, nor system time are influenced by these steps. Therefore, $((\hat{l}, \phi[x \mapsto v], \theta, q), \mathbf{now}) \approx^{\star} (\hat{l}, \eta[x \mapsto v], q')$ holds for the resulting states. Both steps are labelled by τ, so condition 1 of Def. 2.19 is satisfied.

Case: SET
Let $(l, \phi, \theta, q) \rightarrow_{\tau} (\hat{l}, \phi, \theta[t \mapsto on(v)], \pi_t(q))$ in P and $(l, \phi, \theta, q), \mathbf{now}) \approx^{\star} (l, \eta, q')$. By rule SET of Table 1, we get $l \longrightarrow_{g \,\triangleright\, \mathtt{SET}(exp,t)} \hat{l} \in \mathit{Edg}$ where exp is an expression of form $(\mathtt{NOW} + e)$.

According to the rules SET TO SET$_1$ and SET TO SET$_2$ of Table 4, there are two edges corresponding to the edge in $\mathit{Spec}_{P'}$:
(i) $l \longrightarrow_{[g \bigwedge (e \,\geq\, 0)] \,\triangleright\, set\ t:=e} \hat{l} \in \mathit{Edg}'$ and
(ii) $l \longrightarrow_{[g \bigwedge (e<0)] \,\triangleright\, set\ t:=0} \hat{l} \in \mathit{Edg}'$.
The guard g, which valuates to *true* in σ, also valuates to *true* in σ' by Lemma 3.2. An expression e has the same value w both in σ and in σ'.

If $w \geq 0$ then the step $(l, \eta, q') \rightarrow_{\tau} (\hat{l}, \eta[t \mapsto on(w)], q')$ is enabled in P'. Otherwise, the step $(l, \eta, q') \rightarrow_{\tau} (\hat{l}, \eta[t \mapsto on(0)], q')$ is enabled in P' (rule SET of Table 5). In both cases, condition 3 of Def. 3.12 is satisfied for timer t of P and timer variable t of P'. Moreover, q' is a projection of $\pi_t(q)$ on timeout signals, so condition 2 of Def. 3.12 holds. So $((\hat{l}, \phi, \theta[t \mapsto on(v)], \pi_t(q)), \mathbf{now}) \approx^{\star} (\hat{l}, \eta[t \mapsto on(w)], q')$ and $((\hat{l}, \phi, \theta[t \mapsto on(v)], \pi_t(q)), \mathbf{now}) \approx^{\star} (\hat{l}, \eta[t \mapsto on(0)], q')$ hold.

In both cases the step of the original system and the mimicking step are labelled by τ. Hence, condition 1 of Def. 2.19 is satisfied.

Case: RESET
Assume $(l,\ \phi,\ \theta,\ q),\ \mathbf{now}) \approx^{\star} \sigma'$ and $(l, \phi, \theta, q) \rightarrow_{\tau} (\hat{l}, \phi, \theta[t \mapsto \mathit{off}], \pi_t(q))$ of P.

By rule Reset of Table 1, $l \longrightarrow_{g \,\rhd\, \mathtt{RESET}(t)} \hat{l} \in \mathit{Edg}$. According to the rule Reset to Reset, $l \longrightarrow_{g \,\rhd\, \mathit{reset}\ t} \hat{l} \in \mathit{Edg}'$.

The guard g, which valuates to *true* in σ, also valuates to *true* in σ' by Lemma 3.2. So the reset step of P' is mapped to $(l, \eta, q') \rightarrow_{\tau} (\hat{l}, \eta_{[t \mapsto \mathit{off}]}, q')$ that resets timer t in P' (rule Reset of Table 5). So the resetting step of P can be mimicked by the resetting step of P'.

Both the step of the original system and the mimicking step of the transformed one change the value of the timer t (the timer variable t resp.) to *off* i.e., condition 4 of Def. 3.12 is satisfied. Moreover, q' is a projection of $\pi_t(q)$ on timeout signals. It means that condition 2 of Def. 3.12 holds. So $((\hat{l},\ \phi,\ \theta_{[t \mapsto \mathit{off}]},\ \pi_t(q)),\ \mathbf{now}) \approx^{\star} (\hat{l},\ \eta_{[t \mapsto \mathit{off}]},\ q')$. Both steps are labelled by τ, hence condition 1 of Def. 2.19 is satisfied.

Case: Expiration
Let $(l, \phi, \theta, q) \rightarrow_{\tau} (l, \phi, \theta_{[t \mapsto \mathit{off}]}, q :: t)$ in P and $((l, \phi, \theta, q), \mathbf{now}) \approx^{\star} \sigma'$. By rule Expiration of Table 1, this step may take place only if $\mathbf{now} = v$ and $[\![t]\!]_{\sigma} = \mathit{on}(v)$. Since we also have $(\sigma, \mathbf{now}) \approx^{\star} \sigma'$, $((l, \phi, \theta_{[t \mapsto \mathit{off}]}, q :: t), \mathbf{now}) \approx^{\star} \sigma'$ holds (see conditions 3 and 5 of Def. 3.12). Expiration is a τ-step, so we have case 2 of Def. 2.19.

Case: TInput
Let $(l, \phi, \theta, t :: q) \rightarrow_{\tau} (\hat{l}, \phi, \theta, q)$ in P and $((l, \phi, \theta, q), \mathbf{now}) \approx^{\star} (l, \eta, q')$. By rule TInput of Table 1, $l \longrightarrow_{?t} \hat{l} \in \mathit{Edg}$. According to the transformation rule TInput to Timeout of Table 4, $l \longrightarrow_{g_t \,\rhd\, \mathit{reset}\ t} \hat{l} \in \mathit{Edg}'$.

Since $(\sigma, \mathbf{now}) \approx^{\star} \sigma'$, timer variable t has the value zero in σ' by condition 5 of Def. 3.12. So $(l, \eta, q') \rightarrow_{\tau} (\hat{l}, \eta_{[t \mapsto \mathit{off}]}, q')$ is enabled in P' and inputting the timeout signal in P can be mimicked by the timeout in P'.

Both the step of the original system and the mimicking step of the transformed one are setting the timer t (the timer variable t resp.) to *off*, so condition 4 of Def. 3.12 is satisfied. The input queue q' is the projection of q on timeout signals, hence condition 2 of Def. 3.12 holds. Therefore, $((\hat{l}, \phi, \theta, q), \mathbf{now}) \approx^{\star} (\hat{l}, \eta_{[t \mapsto \mathit{off}]}, q')$. The input step and the timeout step are labelled by τ, thus condition 1 of Def. 2.19 is satisfied.

Case: TDiscard
Let $(l, \phi, \theta, t :: q) \rightarrow_{\tau} (l, \phi, \theta, q)$ in P, $l \in \mathit{Loc}_i$ and $(l, \phi, \theta, q), \mathbf{now}) \approx^{\star} (l, \eta, q')$. By rule TDiscard of Table 1, the timeout signal t is not expected in location l of the process P.

Since $(\sigma, \mathbf{now}) \approx^{\star} \sigma'$, timer variable t has the value zero in σ' wrt. condition 5 of the Def. 3.12. By TDiscard of Table 5, $(l, \eta, q') \rightarrow_{\tau} (l, \eta_{[t \mapsto \mathit{off}]}, q')$ is enabled in P' and discarding the timeout signal in P can be mimicked by the timeout in P'. Further, this case coincides with the TInput case above. Therefore, $((l, \phi, \theta, q), \mathbf{now}) \approx^{\star} (l, \eta_{[t \mapsto \mathit{off}]}, q')$. The discard step and the timeout step are labelled by τ, so condition 1 of Def. 2.19 is satisfied.

We showed that there exists a relation $R \subseteq \approx^{\star}$ on states of P and P' and that this relation is branching simulation. □

Checking the rules of Table 1, we have demonstrated that there is $R \subseteq \approx^\star$ that is a branching simulation relation on states of P and P'. Using the rules of n-ary parallel composition (Table 3), we show the same for S and S'.

Theorem 3.1. [BRANCHING SIMULATION]
Let Spec be a specification and Spec′ be the result of the transformation of Spec wrt. the rules of Table 4. Let S be an LTS derived from Spec by applying rules of Tables 1, 2, 3 to Spec, and and S' be an LTS obtained from Spec′ by applying rules of Tables 5, 2, 6. Then there exists a relation $R \subseteq (\Gamma \times \mathit{Time}) \times \Gamma'$ such that $(\gamma, \mathbf{now}) R \gamma'$ implies $(\gamma, \mathbf{now}) \approx^\star \gamma'$ and R is a branching simulation.

Proof. By Lemma 3.4, for all $i \in \{1, \ldots, n\}$, there exists $\approx_i^\star \subseteq (\Sigma_i \times \mathit{Time}) \times \Sigma_i'$ such that $(\sigma_{i0}, \mathbf{now}) \approx_i^\star \sigma_{i0}'$ where σ_{i0} and σ_{i0}' are the initial states of P_i and P_i', respectively. Let S be the LTS built by applying the n-ary parallel composition (Table 3) to $P_1, \ldots, P_n$. Let S' be the LTS built by applying the n-ary parallel composition (Table 6) to $P_1', \ldots, P_n'$. Now given $\approx_i^\star$ for all $i \in \{1, \ldots, n\}$, $\approx^\star \subseteq (\Sigma_1 \times \ldots \times \Sigma_n \times \mathit{Time}) \times (\Sigma_1' \times \ldots \times \Sigma_n')$ is defined by Def. 3.13. It remains to be shown that $\approx^\star$ is a branching simulation relation. First, we show that a channel process of the transformed system can simulate a channel process of the original one. Finally, we check the relation on the rules for parallel composition (Table 3).

Case: IN, OUT
The transformation does not change the semantics of channels except for adding the TICK_c rule. Suppose that $c_i!s(\mathit{pid}, v)$ is enabled in state $\sigma = (c, s(\mathit{pid}, v) :: q)$ of channel process P_c, and moreover, $(\sigma, \mathbf{now}) \approx^\star \sigma'$, where σ' is the state of process channel P_c' of the transformed system. According to Def. 3.12, the queue modelling channel c in system S' has the same content. Moreover, $\sigma' = (c, s(\mathit{pid}, v) :: q)$. So, the $c_i!s(\mathit{pid}, v)$-step of P_c can be mimicked by the $c_i!s(\mathit{pid}, v)$-step of P_c'. In both cases, message (s, v) is removed from the queue modelling the channel, so $(\hat{\sigma}, \mathbf{now}) \approx^\star \hat{\sigma}'$ and condition 1 of Def. 2.19 is satisfied. The OUT case is analogous to the IN case.

Case: TICK
In this case, we have $(\sigma_1, \ldots, \sigma_n, \mathbf{now}) \rightarrow_{tick} (\sigma_1, \ldots, \sigma_n, \mathbf{now} + 1)$. By rule TICK of Table 3, we obtain $\mathit{blocked}(\sigma_1, \ldots, \sigma_n)$. According to Lemma 3.3, $\mathit{blocked}(\ \sigma_1', \ \ldots, \ \sigma_n')$ is *true* as well. So the *tick*-step of the original system can be mimicked by some *tick*-step of the transformed system (see rule TICK_P of Table 5, rules TICK_c and TICK of Table 6).

In S, the *tick*-step increases the system time **now**; the mimicking step of S' decreases all active timers. Suppose that timer t evaluates to $on(v)$ in S. Note that $v > \mathbf{now}$, as otherwise timer t can expire and S is not blocked. According to Def. 3.12, timer variable t should evaluate to $on(w)$, where $\mathbf{now} + w = \max(\mathbf{now}, v)$ and $w > 0$. After *tick*-steps condition 3 of Def. 3.12 still holds because $(w - 1) + (\mathbf{now} + 1) = \max(\mathbf{now}, v)$. Therefore, $(\gamma, \mathbf{now} + 1) \approx^\star \hat{\gamma}$, where $\hat{\gamma}$ is γ with all active timers decreased by 1, is valid and condition 1 of Def. 2.19 is satisfied.

Case: COMM
Assume that $(\ldots, \sigma_i, \ldots, \sigma_j, \ldots) \rightarrow_\tau (\ldots, \hat{\sigma}_i, \ldots, \hat{\sigma}_j, \ldots)$. By rule COMM of Table 3 we get $\sigma_i \rightarrow_{\alpha_i} \hat{\sigma}_i$, $\sigma_j \rightarrow_{\alpha_j} \hat{\sigma}_j$, $i \neq j$ and $comm(\alpha_i, \alpha_j)$ is *true*.

By Lemma 3.4, $\sigma_i \rightarrow_{\alpha_i} \hat{\sigma}_i$ and $\sigma_j \rightarrow_{\alpha_j} \hat{\sigma}_j$ can be mimicked by steps with the same label in P_i' and P_j'. So we have $\sigma_i' \rightarrow_{\alpha_i} \hat{\sigma}_i'$, $\sigma_j' \rightarrow_{\alpha_j} \hat{\sigma}_j'$, $i \neq j$ and $comm(\alpha_i, \alpha_j)$ is *true*. By rule COMM, which is valid both for the original systems and for the transformed ones, we obtain the following τ-step: $(\ldots, \sigma_i', \ldots, \sigma_j', \ldots) \rightarrow_\tau (\ldots, \hat{\sigma}_i', \ldots, \hat{\sigma}_j', \ldots)$. So the communication of P_i with P_j can be mimicked by the communication of P_i' with P_j'.

By Lemma 3.4, $(\hat{\sigma}_i, \mathbf{now}) \approx_i^\star \hat{\sigma}_i'$ and $(\hat{\sigma}_j, \mathbf{now}) \approx_j^\star \hat{\sigma}_j'$. So, we obtain $((\ldots, \hat{\sigma}_i, \ldots, \hat{\sigma}_j, \ldots), \mathbf{now}) \approx^\star (\ldots, \hat{\sigma}_i', \ldots, \hat{\sigma}_j', \ldots)$. Moreover, condition 1 of Def. 2.19 is satisfied.

Case: INTERLEAVE$_{in}$
Here we have $(\ldots, \sigma_i, \ldots) \rightarrow_{c_o?s(pid,v)} (\ldots, \hat{\sigma}_i, \ldots)$. By rule INTERLEAVE$_{in}$ of Table 3, $\sigma_i \rightarrow_{c_o?s(pid,v)} \hat{\sigma}_i$ and $s \in Sig_{ext}$. According to the case IN, OUT above, the $c_o!s(pid, v)$-step of P_i can be mimicked by a $c_o!s(pid, v)$-step of P_i'. So we have $\sigma_i' \rightarrow_{c_o?s(pid,v)} \hat{\sigma}_i'$ and $s \in Sig_{ext}$. Using rule INTERLEAVE$_{in}$ of Table 3, we get $(\ldots, \sigma_i', \ldots) \rightarrow_{c_o?s(pid,v)} (\ldots, \hat{\sigma}_i', \ldots)$. According to the case IN, OUT above, $(\hat{\sigma}_i, \mathbf{now}) \approx_i^\star \hat{\sigma}_i'$, so $((\ldots, \hat{\sigma}_i, \ldots), \mathbf{now}) \approx^\star (\ldots, \hat{\sigma}_i', \ldots)$ and condition 1 of Def. 2.19 is satisfied.

Case: INTERLEAVE$_{out}$
This case is analogous to the INTERLEAVE$_{in}$ case.

Case: INTERLEAVE$_\tau$
Assume we have $(\ldots, \sigma_i, \ldots) \rightarrow_\tau (\ldots, \hat{\sigma}_i, \ldots)$ in S. By rule INTERLEAVE$_\tau$ of Table 3, $\sigma_i \rightarrow_\tau \hat{\sigma}_i$ for some P_i in S. If the τ-step corresponds to one of the cases (DISCARD, INPUT NONEINPUT, ASSIGN, EXPIRATION, SET, RESET) considered by Lemma 3.4, it can also be mimicked by a τ-step of P_i'. So we may conclude that there is a mimicking step $(\ldots, \sigma_i', \ldots) \rightarrow_\tau (\ldots, \hat{\sigma}_i', \ldots)$ in S' and that $((\ldots, \hat{\sigma}_i, \ldots), \mathbf{now}) \approx^\star (\ldots, \hat{\sigma}_i', \ldots)$. Moreover, the condition 1 of Def. 2.19 is satisfied.

We have shown that $(\gamma_0, 0) \approx^\star \gamma_0'$ holds for initial configurations of S and S'. Assumed that the same relation holds for some $(\gamma, \mathbf{now}) \approx^\star \gamma'$, we also have demonstrated that every step that is possible in S can be mimicked by a step of S' so that the conditions of Def 2.19 are satisfied, so $S \preceq_{br} S'$. □

Ideally, we would like $\approx^{\star-1}$ to be a branching simulation relation as well, and establish by that a branching bisimulation between S and S'. However, this is not the case—an attempt to establish a simulation in the reverse direction fails while considering the TIMEOUT case in the transformed system. In principle, TIMEOUT should be mimicked by taking EXPIRATION and TINPUT. But the EXPIRATION step in the original system cannot be taken earlier than the decision about TIMEOUT in the transformed system. On the other hand, when TIMEOUT is taken, it could be already too late to take the EXPIRATION step, since the process queue of the original system can be non-empty, which would mean that the timeout signal could not be immediately consumed from the process queue. However, $S' \preceq_{wtr}$ (Def. 2.17).

Later we also show that the original system and the transformed system are path equivalent up to stuttering (Def. 2.28). To link the "equivalent" traces of the original and transformed systems, we introduce a relation requiring that each step of trace χ of the transformed system can be mimicked either by the same step of trace ζ of the original system or by the same step of ζ preceded by one or more expiration steps, which do not change the valuation of process variables.

Definition 3.14.
Let ζ and χ be traces of S and S' respectively. We write $\zeta \equiv_{\mathcal{U}} \chi$ iff there exists a relation $\mathcal{U} \subseteq \mathbb{N} \times \mathbb{N}$ such that:

1. *$(i, j) \in \mathcal{U}$ implies $(\zeta_\gamma(i), \textbf{now}) \approx^\star \chi_\gamma(j)$.*
2. *ζ and χ can be partitioned as $\zeta^{I_1}\zeta^{I_2}\ldots$ and $\chi^{J_1}\chi^{J_2}\ldots$, respectively, so that for all $k > 0$ $I_k = \{m, \ldots, m+n\}$, $J_k = \{j, j+1\}$ and the following conditions are satisfied:*
   (a) (m, j) and $(m+n, j+1)$ are in $\mathcal{U}$, and $\zeta_\lambda(m+n) = \chi_\lambda(j+1)$;
   (b) for all $m < i < m+n$: $(i, j) \in \mathcal{U}$ and $\zeta_\lambda(i) = \tau$.

We write $S' \preceq_{\mathcal{U}} S$ iff for every trace χ of S' there exists a trace ζ in S such that $\zeta \equiv_{\mathcal{U}} \chi$ for some $\mathcal{U} \subseteq \mathbb{N} \times \mathbb{N}$.

Lemma 3.5.
Let ζ and χ be traces of S and S' respectively. Let $\zeta \equiv_{\mathcal{U}} \chi$ for some $\mathcal{U} \subseteq \mathbb{N} \times \mathbb{N}$. Then $\zeta \equiv_{wtr} \chi$.

Proof. Follows directly from Def. 3.14 and Def. 2.16. □

Let π_ζ and π_χ be paths corresponding to traces ζ and χ respectively, i.e., $\pi_\zeta = \zeta_\gamma(0)\zeta_\gamma(1)\ldots$ and $\pi_\chi = \chi_\gamma(0)\chi_\gamma(1)\ldots$

Lemma 3.6.
Let ζ and χ be traces of S and S' respectively. Let $\zeta \equiv_{\mathcal{U}} \chi$ for some $\mathcal{U} \subseteq \mathbb{N} \times \mathbb{N}$. Let $\mathcal{L}$, $\mathcal{L}'$ be interpretation functions with the range $2^{\mathcal{P}}$, where $\mathcal{P}$ is the set of atomic propositions mentioning only process variables. Then $\pi_\zeta \equiv_{st} \pi_\chi$.

Proof. The transformation defined by the rules of Table 4 does not influence process variables. So *Spec* and *Spec'* obtained by applying the transformation to *Spec* have the same set of process variables.

Since $\zeta \equiv_{\mathcal{U}} \chi$ for some $\mathcal{U} \subseteq \mathbb{N} \times \mathbb{N}$, $\mathcal{L}(\zeta_\gamma(i)) = \mathcal{L}'(\chi_\gamma(j))$ for all $(i, j) \in \mathcal{U}$ by condition 1 of Def. 3.14 and condition 1 of Def. 3.12. By condition 2 of Def. 3.14, each step of trace χ of the transformed system is mimicked either by the same step of trace ζ of the original system or by the same step of ζ preceded by one or more τ steps, which do not change the valuation of process variables. Hence, it is straightforward to show that $\mathcal{L}(Pr(\pi_\zeta)(k)) = \mathcal{L}'(Pr(\pi_\chi)(k))$ for all $k \geq 0$ (cf. Def. 2.27). □

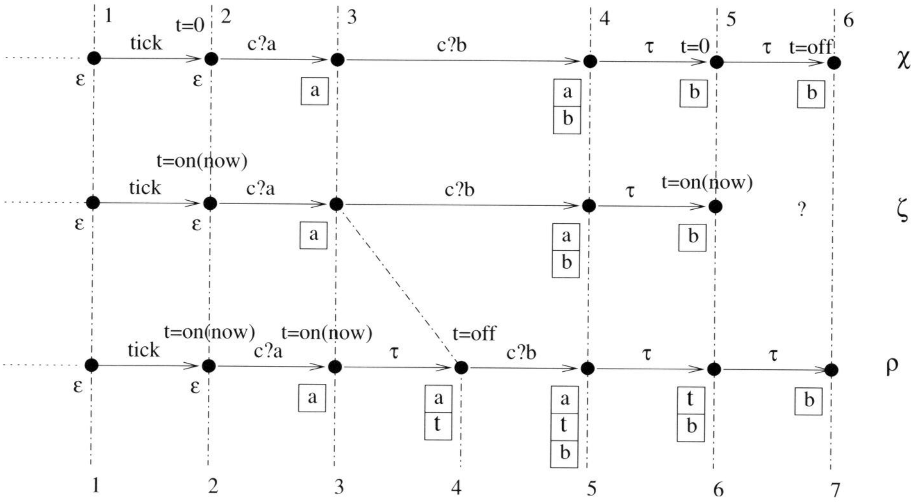

Fig. 4. Treatment of timeout

Further, we show that for each trace χ of S' there is a trace ζ of S such that $\zeta \equiv_{\mathcal{U}} \chi$. As we already mentioned, we do not take an expiration step in S until we meet a timeout in S', so S cannot mimic each step of S'. However, we demonstrate that each time slice (all the steps between two *tick*-steps) of S' can be mimicked by a time slice of S. First, we consider a special treatment for the TIMEOUT case. We take a trace χ of length $n+1$ of S' which has a timeout as the $(n+1)$th-step, assume a trace ζ of S that is equivalent to the $\chi^{(n)}$, and show how to transform ζ into a trace ρ of S such that $\rho \equiv_{\mathcal{U}} \chi$. Since timeout and expiration are local steps, it is enough if we consider the special treatment for timeout only for processes P and P' given by process specifications $Spec_P$ and $Spec_{P'}$, respectively.

Fig. 4 gives an illustration for the special treatment of TIMEOUT. There we consider a suffix of some trace χ of P' starting with a *tick*-step. The *tick*-step is followed by receive steps $c?a$ and $c?b$, step τ that inputs signal a from the input queue and a timeout τ-step for timer variable t. ζ is some trace of P that is equivalent to the prefix of χ up to the timeout step, and relation $\mathcal{U} = \{(1,1,),(2,2),(3,3),(4,4),(5,5)\}$ (this is showed by the dashed lines in Fig. 4).

At the state $\zeta_\gamma(5)$, the input queue of P contains signal b only, so we cannot mimic the timeout step of χ. We transform trace ζ into trace ρ by inserting an expiration step for timer t before the step receiving signal b. By definition of $\mathcal{U}$, timer t may expire at state $\rho_\gamma(3)$, and thus we can mimic the timeout step of χ at state $\rho_\gamma(6)$. Since timer t may expire at any point of the time slice, trace ρ obtained in the result of the above mentioned transformation is still a valid trace of P. The step $\chi_\lambda(4) = c?b$ is mimicked by steps $\rho_\lambda(4) = \tau$ and $\rho_\lambda(5) = c?b$, and $\mathcal{U}$ is modified to $\{(1,1),(2,2),(3,3),(4,3),(5,4),(6,5),(7,6)\}$.

Lemma 3.7.
Let χ be a trace of P' of length $(n+1)$ and the last step of χ be a timeout of timer variable t. Let ζ be a trace of P of the length m such that $\zeta \equiv_{\mathcal{U}} \chi^{(n)}$. Then ζ can be transformed into a trace ρ of P such that $\rho \equiv_{\mathcal{U}} \chi$ for some $\mathcal{U} \subseteq \mathbb{N} \times \mathbb{N}$.

Proof. The $(n+1)$th-step of χ is the timeout of timer variable t. Since in P we cannot take the expiration of t before the timeout is taken in P', there is no timeout signal t at the head of the queue of P in the state $\zeta_\gamma(m)$ for some $m > 0$. Further we consider two cases.

Case: The queue of P is empty at $\zeta_\gamma(m)$.
Since $(\zeta_\gamma(m), \mathbf{now}) \approx^\star \chi_\gamma(n)$, we can take an expiration step for timer t at $\zeta_\gamma(m)$. After the expiration step, the step consuming the timeout signal is enabled in P due to rule TInput to Timeout of Table 4. So we can modify $\mathcal{U}$ and ζ as follows.

1. ρ is ζ extended by the expiration step and the step inputting timeout signal t, i.e., $\rho_\gamma(i) = \zeta_\gamma(i)$, $\rho_\lambda(j) = \zeta_\lambda(j)$ $\forall i = 0..m, j = 1..m$. Initially, $\mathcal{U}'$ is defined as $\mathcal{U}$.
2. We add the expiration step, i.e., assuming $\zeta'_\gamma(m) = (l, \phi, \theta, \epsilon)$, $\rho_\gamma(m+1) = (l, \phi, \theta[t \mapsto \mathit{off}], t)$, $\rho_\lambda(m+1) = \tau$, and we add the pair $(m+1, n)$ to $\mathcal{U}'$, i.e., $\mathcal{U}' = \mathcal{U}' \bigcup \{(m+1, n)\}$. In this case, $(\rho_\gamma(m+1), \mathbf{now}) \approx^\star \chi_\gamma(n)$.
3. Next, we add the input step, which consumes timeout signal t. Assuming $\chi_\gamma(n+1) = (\hat{l}, \eta[t \mapsto \mathit{off}], \epsilon)$, we define $\rho_\gamma(m+2) = (\hat{l}, \phi, \theta, \epsilon)$ and $\rho_\lambda(m+2) = \tau$, and we add the pair $(m+2, n+1)$ to $\mathcal{U}'$, i.e., $\mathcal{U}' = \mathcal{U}' \bigcup \{(m+2, n+1)\}$. In this case, $(\rho_\gamma(m+2), \mathbf{now}) \approx^\star \chi_\gamma(n+1)$.

Since $\zeta \equiv_{\mathcal{U}} \chi^{(n)}$, ζ can be partitioned as $\zeta^{I_1} \dots \zeta^{I_v}$ and $\chi^{(n)}$ can be partitioned as ${\chi^{(n)}}^{J_1} \dots {\chi^{(n)}}^{J_v}$ (see condition 2 of Def. 3.14). Since $\mathcal{U}$ is left unmodified for the prefix of ρ that coincides with ζ, ρ and χ can be partitioned as $\rho^{I_1} \dots \rho^{I_v} \rho^{I_{v+1}}$ and $\chi^{J_1} \dots \chi^{J_v} \chi^{J_{v+1}}$, where $\rho^{I_i} = \zeta^{I_i}$ and $\chi^{J_i} = {\chi^{(n)}}^{J_i}$ for all $1 \geq i \geq v$. The last step of χ forms part $\chi^{J_{v+1}}$. The expiration step together with the input step consuming the timeout signal form part $\rho^{I_{v+1}}$. All the conditions of Def. 3.14 are satisfied and $\rho \equiv_{\mathcal{U}} \chi$.

Case: The queue of P is not empty at $\zeta_\gamma(m)$.
Here, we should find a receiving step for an element msg staying at the head of the queue at $\zeta_\gamma(m)$, insert the expiration step before the step receiving msg, modify relation $\mathcal{U}$ with respect to this insertion, and extend ζ by the step mimicking the timeout. Since the expiration can be taken at any point of the time slice, the trace that we get as the result of this insertion is a trace of P. For the sake of readability of the proof we assume that all the messages in the queue are different. In order to include the expiration step, we modify $\mathcal{U}$ and ζ as follows.

- Let $\zeta_\gamma(m) = (l, \phi, \theta, msg :: q)$. Going from m to 1, we find the first k along the trace ζ such that $\zeta_\lambda(k)$ is labelled by $c_i?msg$ (or an expiration step in case msg is a timeout signal).
- We keep the k-prefix of ζ and the corresponding subset of $\mathcal{U}$. Initially, $\mathcal{U}' = \{(i,j) \mid (i,j) \in \mathcal{U},\ 0 \le i < k\}$, $\forall i = 0..k-1$: $\rho_\gamma(i) = \zeta_\gamma(i)$, $\forall i = 1..k-1 : \rho_\lambda(i) = \zeta_\lambda(i)$.
- We add the expiration step in front of the step receiving msg. Let $\rho_\gamma(k-1) = (l, \phi, \theta, q)$. Since $\zeta \equiv_{\mathcal{U}} \chi^{(n)}$, there is $(k, w) \in \mathcal{U}$ such that $\zeta_\lambda(k)$, $\chi_\lambda(w)$ belong to the parts ζ^{I_w}, ${\chi^{(n)}}^{J_w}$ respectively. We define $\rho_\gamma(k) = (l, \phi, \theta_{[t \mapsto \mathit{off}]}, q :: t)$, $\rho_\lambda(k) = \tau$, and we add the pair (k, w) to $\mathcal{U}'$, i.e., $\mathcal{U}' = \mathcal{U}' \bigcup \{(k, w)\}$. Moreover, $(\rho_\gamma(k), \mathbf{now}) \approx^\star \chi_\gamma(l)$.
- The suffix of ζ starting with the step receiving msg is shifted due to the inclusion of the expiration step. The shift is done as follows: $\forall i = k..m$: $\rho_\gamma(i+1) = (l, \phi, \theta, \hat{q})$ where $\zeta_\gamma(i) = (l, \phi, \theta, \tilde{q})$, $\tilde{q} = q_1 :: msg :: q_2$ and $\hat{q} = q_1 :: t :: msg :: q_2$ (i.e., the queue is modified wrt. the inclusion), and $\rho_\lambda(i+1) = \zeta_\lambda(i)$.

  Relation $\mathcal{U}'$ is modified by inclusion as follows: $\forall (i,j) \in \mathcal{U}$ s.t. $k \le i \le m$: we add the pair $(i+1, j)$ to $\mathcal{U}'$, i.e., $\mathcal{U}' = \mathcal{U}' \bigcup \{(i+1, j)\}$. Since condition 2 of Def. 3.12 is still satisfied, $(\zeta_\gamma(i+1), \mathbf{now}) \approx^\star \chi_\gamma(i)$.
- Let $\chi_\gamma(n+1) = (\hat{l},\ \eta_{[t \mapsto \mathit{off}]},\ q)$. Finally, we add the input of the timeout signal t. Then $\rho_\gamma(m+2) = (\hat{l}, \phi, \theta, q')$, and $\rho_\lambda(m+2) = \tau$, and we add the pair $(m+2, n+1)$ to $\mathcal{U}'$, i.e., $\mathcal{U}' = \mathcal{U}' \bigcup \{(m+2, n+1)\}$. Moreover, $(\rho_\gamma(m+2), \mathbf{now}) \approx^\star \chi_\gamma(n+1)$.

Since $\zeta \equiv_{\mathcal{U}} \chi^{(n)}$, ζ can be partitioned as $\zeta^{I_1} \ldots \zeta^{I_v}$ and $\chi^{(n)}$ can be partitioned ${\chi^{(n)}}^{J_1} \ldots {\chi^{(n)}}^{J_v}$. $\rho^{(k-1)}$ coincides with $\zeta^{(k-1)}$, so we can partition $\rho^{(k-1)}$ so that $\rho^{I_i} = \zeta^{I_i}$ and $\chi^{J_i} = {\chi^{(n)}}^{J_i}$ for all $1 \le i < w$, where ζ^{I_w} is a part containing the kth-step of ζ.

The partition I_w of ζ ends with the kth-step. We add the expiration step in front of the kth-step. The partition ζ^{I_w} is modified by the inclusion, so that the part ρ^{I_w}, which corresponds to the expiration step followed by the kth-step, still satisfies the conditions of Def. 3.14.

Further, the suffix of ρ starting in state $\rho_\gamma(k+1)$ can be partitioned in the same way as the corresponding suffix of ζ starting in $\zeta_\gamma(k)$. The parts ρ^{I_j} differ from the parts ζ^{I_j} only by the presence of the timeout signal in the input queue. The conditions of Def. 3.14 are satisfied by ρ^{I_j} and ${\chi^{(n)}}^{J_j}$ for all $w+1 \le j \le v$. The input step consuming the timeout signal forms part $\rho^{I_{v+1}}$ of the original system. The corresponding timeout step in χ forms part $\chi^{J_{v+1}}$. The conditions of Def. 3.14 are satisfied, so $\rho \equiv_{\mathcal{U}} \chi$.

Concluding, we have shown how to construct ρ such that $\rho \equiv_{\mathcal{U}} \chi$ for some $\mathcal{U} \subseteq \mathbb{N} \times \mathbb{N}$ and ρ is a trace of P. □

Lemma 3.8.
Let $Spec_P$ be a process specification and $Spec_{P'}$ be the result of the transformation of $Spec_P$ wrt. the rules of Table 4. Let P and P' be the processes derived

from $Spec_P$ and $Spec_{P'}$ by applying the rules of Table 1 and Table 5, respectively. Then for each trace χ of P' there exists a trace ζ of P such that $\zeta \equiv_{\mathcal{U}} \chi$ for some $\mathcal{U} \subseteq \mathbb{N} \times \mathbb{N}$.

Proof. Here we demonstrate that for each trace χ of P' there exists a trace ζ of P, such that $\zeta \equiv_{\mathcal{U}} \chi$ for some $\mathcal{U} \subseteq \mathbb{N} \times \mathbb{N}$.

In Lemma 3.4, we have already demonstrated that $(\gamma_0, 0) \approx^\star \gamma'_0$ is valid for the initial states of P and P', so $\mathcal{U} = \{(0,0)\}$ and $\zeta_\gamma(0) = \gamma_0$ initially. Further, trace ζ and relation $\mathcal{U}$ are constructed by induction on the length of the trace χ. Each step of P' is mimicked by a step of P, and configurations reached by the step of the transformed system and the mimicking step of the original one are related by $\approx^\star$. Assume that $(i,j) \in \mathcal{U}$. By Def. 3.14, $(\zeta_\gamma(i), \mathbf{now}) \approx^\star \chi_\gamma(j)$. Now we proceed with a case analysis on the rules of Table 5.

Case: INPUT
In this case we have a step $(l, \eta, s(pid, v) :: q') \rightarrow_\tau (\hat{l}, \eta_{[x \mapsto v]}, q')$ of P', and $(\zeta_\gamma(i), \mathbf{now}) \approx^\star \chi_\gamma(j)$, where $\chi_\gamma(j) = (l, \eta, s(pid, v) :: q$, and $\chi_\lambda(j+1) = \tau$ and $\chi_\gamma(j+1) = (\hat{l}, \eta_{[x \mapsto v]}, q')$.

Rule INPUT of Table 5 gives an edge $l \longrightarrow_{?s(x)} \hat{l} \in Edg'$. Since the transformation leaves this edge untouched, there is an edge $l \longrightarrow_{?s(x)} \hat{l} \in Edg$.

Since $(\zeta_\gamma(i), \mathbf{now}) \approx^\star (l, \eta, s(pid, v) :: q)$, $\zeta_\gamma(i) = (l, \phi, \theta, s(pid, v) :: q)$ and the input is enabled in $\zeta_\gamma(i)$. So the input step of P' is mimicked by the following step: $(l,\ \phi,\ \theta,\ s(pid, v) :: q) \rightarrow_\tau\ (\hat{l},\ \phi_{[x \mapsto v]},\ \theta,\ q)$ (rule INPUT of Table 1).

We define $\zeta_\lambda(i+1) = \tau$, $\zeta_\gamma(i+1) = (\hat{l}, \phi_{[x \mapsto v]}, \theta, q)$ and we add the pair $(i+1, j+1)$ to $\mathcal{U}$, i.e., $\mathcal{U} = \mathcal{U} \bigcup \{(i+1, j+1)\}$. Both the step of the transformed system and the mimicking step of the original one form parts satisfying condition 2 of Def. 3.14. Since both the step of the transformed system and the mimicking step of the original one remove $s(pid, v)$ from the input queue and modify the value of x to v, $(\zeta_\gamma(i+1), \mathbf{now}) \approx^\star \chi_\gamma(j+1)$. The other conditions of Def. 3.14 are satisfied as well.

Case: DISCARD
We have $(l, \eta, s(pid, v) :: q) \rightarrow_\tau (l, \eta, q)$ of P' and $(\zeta_\gamma(i), \mathbf{now}) \approx^\star \chi_\gamma(j)$ where $\chi_\gamma(j) = (l, \eta, s(pid, v) :: q')$, $\chi_\lambda(j+1) = \tau$, and $\chi_\gamma(j+1) = (l, \eta, q')$.

By rule DISCARD of Table 5, we derive that signal s is not expected by process P in location l. The transformation rules of Table 4 leave the set of nontimeout signals expected at location l untouched. Therefore, a signal, which is not expected in location l of the transformed system, is also not expected in location l of the original system. Since $(\zeta_\gamma(i), \mathbf{now}) \approx^\star \chi_\gamma(j)$, two cases are possible: (i) signal s is at the head of the input queue of P in $\zeta_\gamma(i)$, i.e., $\zeta_\gamma(i) = (l, \phi, \theta, s(pid, v) :: q)$; (ii) there are one or more timeout signals in front of s at the head of the queue of P in $\zeta_\gamma(i)$, i.e., $\zeta_\gamma(i) = (l, \phi, \theta, t_1 :: \ldots t_n :: s(pid, v) :: q)$.

In the first case, the discard step of P' can be mimicked by the discard step of P (rule DISCARD of Table 1). We define $\zeta_\lambda(i+1) = \tau$, $\zeta_\gamma(i+1) = (l, \phi, \theta, q)$ and we add the pair $(i+1, j+1)$ to $\mathcal{U}$. Both the step of the transformed system and the mimicking step of the original one remove $s(pid, v)$ from the input queue, so $(\zeta_\gamma(i+1), \mathbf{now}) \approx^\star \chi_\gamma(j+1)$. The conditions of Def. 3.14 are satisfied.

Since we do not take an expiration step for timer t in P until the timeout of t is met in P', there is no need to consider the second case.

Case: OUTPUT
Here we have $(l, \eta, q') \rightarrow_{c_o!s(pid,v)} (\hat{l}, \eta, q')$ in S'_P and $(\zeta_\gamma(i), \mathbf{now}) \approx^\star \chi_\gamma(j)$ where $\chi_\gamma(j) = (l, \eta, q')$, $\chi_\lambda(j+1) = c_o!s(pid, v)$ and $\chi_\gamma(j+1) = (\hat{l}, \eta, q')$.

By rule OUTPUT of Table 5, we get $l \longrightarrow_{g \,\triangleright\, c!(s,e)} \hat{l} \in Edg'$. The transformation rules of Table 4 leave output edges unmodified, so $l \longrightarrow_{g \,\triangleright\, c!(s,e)} \hat{l} \in Edg'$. Since guard g evaluates to *true* in $\chi_\gamma(j)$, it also evaluates to *true* in $\zeta_\gamma(i)$ too. Expression e has the same value in $\chi_\gamma(j)$ and in $\zeta_\gamma(i)$ (see Lemma 3.2). So the output step of P' can be mimicked by the output step of P (rule OUTPUT of Table 1).

We define $\zeta_\lambda(i+1)$ as $c_o!s(pid, v)$, $\zeta_\gamma(i+1) = (\hat{l}, \phi, \theta, q)$ and we add the pair $(i+1, j+1)$ to $\mathcal{U}$. Condition 2 of Def. 3.14 is satisfied. Both the step of the transformed system and the mimicking step of the original one change the location only, thus $(\zeta_\gamma(i+1), \mathbf{now}) \approx^\star \chi_\gamma(j+1)$, and condition 1 of Def. 3.14 is satisfied.

Case: RECEIVE
We have $(l, \eta, q') \rightarrow_{c_i?s(pid,v)} (l, \eta, q' :: s(pid, v))$ for some $v \in D$, where c is an input channel of P' (rule RECEIVE of Table 5) and $(\zeta_\gamma(i), \mathbf{now}) \approx^\star \chi_\gamma(j)$ where $\chi_\gamma(j) = (l, \eta, q')$, $\chi_\lambda(j+1) = c_i?s(pid, v)$, and $\chi_\gamma(j+1) = (l, \eta, q :: s(pid, v))$.

The transformation rules of Table 4 do not change the structure of the system, i.e., P' and P have the same set of input channels. By rule RECEIVE of Table 1, the receive step of P' can be mimicked by the following receive step $(l,\ \phi,\ \theta,\ q) \rightarrow_{c_i?s(pid,v)} (l,\ \phi,\ \theta,\ q :: s(pid, v))$ enabled in $\zeta_\gamma(i)$ of P.

We define $\zeta_\lambda(i+1) = c_i?s(pid, v)$, $\zeta_\gamma(i+1) = (l, \phi, \theta, q :: s(pid, v))$ and add the pair $(i+1, j+1)$ to $\mathcal{U}$. Condition 2 of Def. 3.14 is satisfied. Both the step of the transformed system and the mimicking step add $s(pid, v)$ to the input queue, thus $(\zeta_\gamma(i+1), \mathbf{now}) \approx^\star \chi_\gamma(j+1)$, and condition 1 of Def. 3.14 is satisfied.

Case: ASSIGN
We have $(l, \eta, q') \rightarrow_\tau (\hat{l}, \eta[x \mapsto v], q')$ and $(\zeta_\gamma(i), \mathbf{now}) \approx^\star \chi_\gamma(j)$ where $\chi_\gamma(j) = (l, \eta, q')$, $\chi_\lambda(j+1) = \tau$, and $\chi_\gamma(j+1) = (\hat{l}, \eta[x \mapsto v], q')$.

By rule ASSIGN of Table 5, $l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in Edg'$. The transformation rules of Table 4 leave output edges unmodified, so $l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in Edg$. According to Lemma 3.2, guard g and expression e have in $\zeta_\gamma(i)$ the same value as in $\chi_\gamma(j)$. Therefore, the assignment in P' can be mimicked by assignment step $(l, \phi, \theta, q) \rightarrow_\tau (\hat{l}, \phi[x \mapsto v], \theta, q)$ of P.

Here $\zeta_\lambda(i+1) = \tau$, $\zeta_\gamma(i+1) = (\hat{l}, \phi[x \mapsto v], \theta, q)$ and we add the pair $(i+1, j+1)$ to $\mathcal{U}$. Condition 2 of Def. 3.14 is satisfied. The value of variable x is changed to v both by the step of the transformed system and by the mimicking step, thus $(\zeta_\gamma(i+1), \mathbf{now}) \approx^\star \chi_\gamma(j+1)$, and condition 1 of Def. 3.14 is satisfied as well.

Case: SET
Assume $(l, \eta, q') \rightarrow_{\tau} (\hat{l}, \eta[t \mapsto on(v)], q')$ and $(\zeta_{\gamma}(i), \mathbf{now}) \approx^{\star} \chi_{\gamma}(j)$ where we have $\chi_{\gamma}(j) = (l, \eta, q')$, $\chi_{\lambda}(j+1) = \tau$, and $\chi_{\gamma}(j+1) = (\hat{l}, \eta[t \mapsto on(v)], q')$.

By rule SET of Table 5, $l \xrightarrow{}_{g' \rhd set\ t:=e} \hat{l} \in Edg'$. According to rules SET TO SET$_1$ and SET TO SET$_2$ of Table 4, $l \xrightarrow{}_{g \rhd \mathtt{SET(NOW}+e,t)} \hat{l} \in Edg$.

Two cases are possible: (i) g' is a condition of the form $[g \bigwedge(e \geq 0)]$ for some g in P, and (ii) g' is a guard of the form $[g \bigwedge(e < 0)]$.

In both cases, g evaluates to *true* in $\zeta_{\gamma}(i)$ iff g' is *true* in $\chi_{\gamma}(j)$ by Lemma 3.2, and the setting of timer variable t in P' can be mimicked by setting timer t in P, i.e., by the step $(l, \eta, q) \rightarrow_{\tau} (\hat{l}, \eta[t \mapsto on(w)], q)$ where $w = [\![\mathtt{NOW} + e]\!]_{\zeta_{\gamma}(i)}$ (rule SET of Table 1).

Here $\zeta_{\lambda}(i+1) = \tau$, $\zeta_{\gamma}(i+1) = (\hat{l}, \phi, \theta[t \mapsto on(w)], q)$ and we add the pair $(i+1, j+1)$ to $\mathcal{U}$. Condition 2 of Def. 3.14 is satisfied.

If $e \geq 0$ is *true*, then guard $[g \bigwedge(e \geq 0)]$ is *true* in $\chi_{\gamma}(j)$, and $w = \mathbf{now} + v$, i.e., condition 3 of Def. 3.12 is satisfied. Otherwise, guard $[g \bigwedge(e < 0)]$ is *true* in $\chi_{\gamma}(j)$ and e has the value zero, so condition 3 of Def. 3.12 is satisfied as well. Hence, $(\zeta_{\gamma}(i+1), \mathbf{now}) \approx^{\star} \chi_{\gamma}(j+1)$, and condition 1 of Def. 3.14 is satisfied.

Case: RESET
Here we have $(l, \eta, q') \rightarrow_{\tau} (\hat{l}, \eta[t \mapsto off], q')$, $l \notin Loc_i'$ and $(\zeta_{\gamma}(i), \mathbf{now}) \approx^{\star} \chi_{\gamma}(j)$ where $\chi_{\gamma}(j) = (l, \eta, q')$, $\chi_{\lambda}(j+1) = \tau$, and $\chi_{\gamma}(j+1) = (\hat{l}, \eta[t \mapsto off], q')$.

By rule RESET of Table 5, $l \xrightarrow{}_{g \rhd reset\ t} \hat{l} \in Edg'$ for some $l \notin Loc_i$. By transformation rule RESET TO RESET of Table 4, $l \xrightarrow{}_{g \rhd \mathtt{RESET}(t)} \hat{l} \in Edg$.

Since $l \notin Loc_i$, guard g is not a timeout guard. By Lemma 3.2, guard g evaluates to *true* both in $\chi_{\gamma}(j)$ and in $\zeta_{\gamma}(i)$, so the reset step of P' can be mimicked by the reset step $(l, \phi, \theta, q) \rightarrow_{\tau} (\hat{l}, \phi, \theta[t \mapsto off], \pi_t(q))$ of P (rule RESET of Table 1).

Here $\zeta_{\lambda}(i+1) = \tau$, $\zeta_{\gamma}(i+1) = (\hat{l}, \phi, \theta[t \mapsto off], \pi_t(q))$ and we add the pair $(i+1, j+1)$ to $\mathcal{U}$. Condition 2 of Def. 3.14 is satisfied.

Both the reset step of P' and the reset step of P change the value of timer variable (resp. timer) t to *off* and the reset step of P removes all timeout signals of t from the input queue, thus condition 4 of Def. 3.12 is satisfied. Therefore, $(\zeta_{\gamma}(i+1), \mathbf{now}) \approx^{\star} \chi_{\gamma}(j+1)$, and condition 1 of Def. 3.14 is satisfied.

Case: TICK$_P$
Here we have $(l, \eta, q') \rightarrow_{tick} (l, \eta_{[t \mapsto dec(t)]}, q')$ and $(\zeta_{\gamma}(i), \mathbf{now}) \approx^{\star} \chi_{\gamma}(j)$ where $\chi_{\gamma}(j) = (l, \eta, q')$, $\chi_{\lambda}(j+1) = tick$, and $\chi_{\gamma}(j+1) = (l, \eta_{[t \mapsto dec(t)]}, q')$.

The premise of rule TICK$_P$ of Table 5 says that $blocked(\chi_{\gamma}(j))$ is *true*. By Lemma 3.3, $blocked(\zeta_{\gamma}(i))$ also holds. So the *tick*-step of P' can be mimicked by the *tick*-step of P that is $\mathbf{now} \rightarrow_{tick} \mathbf{now} + 1$ (rules TICK of Table 3).

In this case, $\zeta_{\lambda}(i+1) = tick$, $\zeta_{\gamma}(i+1) = \zeta_{\gamma}(i)$, $\zeta_{\mathbf{now}}(i+1) = \zeta_{\mathbf{now}}(i) + 1$ and we add the pair $(i+1, j+1)$ to $\mathcal{U}$. Condition 2 of Def. 3.14 is satisfied.

The *tick*-step of P' decreases values of timer variables and the *tick*-step of P increases the system time, i.e. $(w - 1) + (\mathbf{now} + 1) = v$ where w is the positive value of the timer variable and v is the value of the corresponding timer, thus condition 3 of Def. 3.12 is satisfied. Moreover, $(\zeta_\gamma(i+1), \mathbf{now}+1) \approx^\star \chi_\gamma(j+1)$, and condition 1 of Def. 3.14 is satisfied.

Case: TIMEOUT, TDISCARD
See Lemma 3.7.

We have shown that for trace χ of P' there exists a trace ζ of P such that $\zeta \equiv_{\mathcal{U}} \chi$. Each construction step preserves the relation $\approx^\star$ on resulting states. □

Lemma 3.9.
Let Spec be a specification, Spec′ be the result of transforming Spec according to the rules of Table 4. Let S be the LTS derived from Spec by applying rules of Tables 1, 2, 3 to Spec, and S′ be the LTS obtained from Spec′ by applying rules of Table 5, rules IN *and* OUT *of Table 2, rules of Table 6. Then for each trace χ of S' there exists a trace ζ of S such that $\zeta \equiv_{\mathcal{U}} \chi$ for some $\mathcal{U} \subseteq \mathbb{N} \times \mathbb{N}$.*

Proof. By Lemma 3.8, we have already shown that $P' \preceq_{\mathcal{U}} P$ for single processes. For systems consisting of n components, it is straightforward to prove that $P'_i \preceq_{\mathcal{U}} P_i$ for all $i \in \{1, \ldots, n\}$ implies $S' \preceq_{\mathcal{U}} S$, proceeding similarly by case analysis on the rules of Table 6 and by using Lemma 3.3.

$(i, j) \in \mathcal{U}$ implies $(\zeta_\gamma(i), \mathbf{now}) \approx^\star \chi_\gamma(j)$. Trace ζ of S and relation $\mathcal{U}$ are constructed by induction on the length of trace χ of S'. Initial configurations γ_0 and γ'_0 of S and S' respectively are related by $\approx^\star$, so $\mathcal{U}$ is initially $(0, 0)$ and $\zeta_\gamma(0) = \gamma_0$. Assume that $(i, j) \in \mathcal{U}$. ζ is a trace of the length i of S and equivalent to $\chi^{(j)}$ of S'. Further we proceed by a case analysis on the rules of Table 6.

Case: IN, OUT
Since the transformation does not influence in- and out-behaviour of channels and $(i, j) \in \mathcal{U}$ implies $(\zeta_\gamma(i), \mathbf{now}) \approx^\star \chi_\gamma(j)$, i.e., channel c has in $\zeta_\gamma(i)$ the same contents as in $\chi_\gamma(j)$. So a $c_o?s(pid, v)$-step of channel c in S' can always be mimicked by the same step of channel c in S. Moreover, the configurations reached by the mimicking step of the original system and the step of the transformed one are related by $\approx^\star$. The same is valid for $c_i!s(pid, v)$-steps.

Case: COMM
We have $\chi_\gamma(j) \rightarrow_\tau \chi_\gamma(j+1)$ and $(\zeta_\gamma(i), \mathbf{now}) \approx^\star \chi_\gamma(j)$ where we have $\chi_\gamma(j) = (\chi_\gamma(j)_1, \ldots, \chi_\gamma(j)_n)$, $\zeta_\gamma(i) = (\zeta_\gamma(i)_1, \ldots, \zeta_\gamma(i)_n)$, $\chi_\gamma(j+1) = (\chi_\gamma(j+1)_1, \ldots, \chi_\gamma(j+1)_n)$. By the premises of rule COMM, this implies that there are some $k, m \in \{1, \ldots, n\}$ such that $\chi_\gamma(j)_m \rightarrow_{\alpha_m} \chi_\gamma(j+1)_m$, $\chi_\gamma(j)_k \rightarrow_{\alpha_k} \chi_\gamma(j+1)_k$ and predicate $comm(\alpha_m, \alpha_k)$ is *true*.

Due to Lemma 3.8 and case (IN, OUT) considered above, steps labelled by α_m and α_k can be mimicked by steps with the same labelling in P_m and P_k. So we have $\zeta_\gamma(i)_m \rightarrow_{\alpha_m} \zeta_\gamma(i+1)_m$ and $\zeta_\gamma(i)_k \rightarrow_{\alpha_k} \zeta_\gamma(i+1)_k$. Moreover, $(\zeta_\gamma(i+1)_m, \mathbf{now}) \approx^\star \chi_\gamma(j+1)_m$, $(\zeta_\gamma(i+1)_k, \mathbf{now}) \approx^\star \chi_\gamma(j+1)_k$ and $comm(\alpha_k, \alpha_m)$ holds. By rule COMM, which is valid both for the original system and for the transformed one, we derive for some $k, m \in \{1 \ldots n\}$ transition $(\zeta_\gamma(i)_1, \ldots, \zeta_\gamma(i)_n) \rightarrow_\tau (\zeta_\gamma(i+1)_1, \ldots, \zeta_\gamma(i+1)_n)$. So the communication of P'_m with P'_k can be mimicked by the communication of P_m with P_k.

In this case, $\zeta_\lambda(i+1) = \tau$, $\zeta_\gamma(i+1) = (\zeta_\gamma(i+1)_1, \ldots, \zeta_\gamma(i+1)_n)$ and we add the pair $(i+1, j+1)$ to $\mathcal{U}$. Condition 2 of Def. 3.14 is satisfied.

Since $(\zeta_\gamma(i+1)_k, \mathbf{now}) \approx^\star \chi_\gamma(j+1)_k$ and $(\zeta_\gamma(i+1)_k, \mathbf{now}) \approx^\star \chi_\gamma(j+1)_k$ for all $k \in \{1, \ldots, n\}$, $(\zeta_\gamma(i+1), \mathbf{now}) \approx^\star \chi_\gamma(j+1)$, and condition 1 of Def. 3.14 is satisfied.

Case: TICK
Here we have $\chi_\gamma(j) \rightarrow_{tick} \chi_\gamma(j+1)$ decreasing values of non-zero timer variables of S', and $(\zeta_\gamma(i), \mathbf{now}) \approx^\star \chi_\gamma(j)$.

By rule TICK of Table 6, predicate *blocked* is *true* in $\chi_\gamma(i)$. According to Lemma 3.3, *blocked* is also *true* in $\zeta_\gamma(j)$. So the *tick*-step of S' can be mimicked by the *tick*-step of S.

In this case, $\zeta_\lambda(i+1)$ is defined as *tick*, $\zeta_\gamma(i+1) = \zeta_\gamma(i)$, $\zeta_{\mathbf{now}}(i+1) = \zeta_{\mathbf{now}}(i)+1$ and we add the pair $(i+1, j+1)$ to $\mathcal{U}$. The condition 2 of Def. 3.14 is satisfied.

The *tick*-step of S' decreases values of non-zero timer variables and the *tick*-step of S increases system time, i.e., $(w-1) + (\mathbf{now}+1) = v$ where w is the value of a non-zero timer variable and v is the value of the corresponding timer. Condition 3 of Def. 3.12 is satisfied, hence $(\zeta_\gamma(i+1), \mathbf{now}+1) \approx^\star \chi_\gamma(j+1)$. Moreover, condition 1 of Def. 3.14 is satisfied.

Case: INTERLEAVE$_{in}$
Here we have $(\chi_\gamma(j)_1, \ldots, \chi_\gamma(j)_n) \rightarrow_{c!s(pid,v)} (\chi_\gamma(j+1)_1, \ldots, \chi_\gamma(j+1)_n)$, and $(\zeta_\gamma(i), \mathbf{now}) \approx^\star \chi_\gamma(j)$ where $\chi_\gamma(j) = (\chi_\gamma(j)_1, \ldots, \chi_\gamma(j)_n)$, $\zeta_\gamma(i) = (\zeta_\gamma(i)_1, \ldots, \zeta_\gamma(i)_n)$, and $\chi_\gamma(j+1) = (\chi_\gamma(j+1)_1, \ldots, \chi_\gamma(j+1)_n)$. By the premises of rule INTERLEAVE$_{in}$, $\chi_\gamma(j)_k \rightarrow_{c?s(pid,v)} \chi_\gamma(j+1)_k$ for some $k \in \{1, \ldots, n\}$ and $s \in Sig_{ext}$.

The set of external signals for S coincides with the set of external signals for S', thus there is a step $\zeta_\gamma(i)_k \rightarrow_{c!s(pid,v)} \zeta_\gamma(i+1)_k$. By rule INTERLEAVE$_{in}$, $\zeta_\lambda(i+1) = c!s(pid, v)$, $\zeta_\gamma(i+1) = (\ldots, \zeta_\gamma(i+1)_k, \ldots)$ and we add the pair $(i+1, j+1)$ to $\mathcal{U}$. Condition 2 of Def. 3.14 is satisfied.

Since $(\zeta_\gamma(i)_m, \mathbf{now}) \approx^\star \chi_\gamma(j)_m$ for all $m \in \{1, \ldots, n\}$, $(\zeta_\gamma(i+1), \mathbf{now}) \approx^\star \chi_\gamma(j+1)$, i.e., condition 1 of Def. 3.14 is satisfied.

Case: INTERLEAVE$_{out}$
The proof of this case is analogous to the proof of the INTERLEAVE$_{in}$ case.

Case: INTERLEAVE$_\tau$
Assume that $\chi_\lambda(j+1)$ is a τ-step and that $\chi_\gamma(j) = (\chi_\gamma(j)_1, \ldots, \chi_\gamma(j)_n)$. Let $\chi_\gamma(j)_k \rightarrow_\tau \chi_\gamma(j+1)_k$ for some $k \in \{1, \ldots, n\}$. The τ-step corresponds to one of the cases DISCARD, NONEINPUT, ASSIGN, EXPIRATION, SET, RESET of Table 5 or the case COMM considered above. As we already showed, it can be simulated by a τ-step of P_k in all the cases.

The configurations reached by the τ-step in S' and the mimicking τ-step of S are related by $\approx^\star$ for all cases. Therefore, we add the pair $(i+1, j+1)$ to $\mathcal{U}$

and define $\zeta_\lambda(i+1) = \tau$, $\zeta_\gamma(j+1) = \gamma$ where γ is the configuration reached in S by the mimicking τ-step.

By the construction mechanism described above, we have shown that for each trace χ of S' there is a trace ζ of S such that $\zeta \equiv_{\mathcal{U}} \chi$ for some $\mathcal{U} \subseteq \mathbb{N} \times \mathbb{N}$. □

Lemma 3.10. [WEAK TRACE EQUIVALENCE]
Let Spec be a specification and Spec′ be the result of transforming Spec according to the rules of Table 4. Let S be the LTS derived from Spec by applying the rules of Tables 1, 2, 3 to Spec, and S' be the LTS obtained from Spec′ by applying the rules of Table 5, rules IN *and* OUT *of Table 2, rules of Table 6. Then $S \equiv_{wtr} S'$.*

Proof. Case: $S \preceq_{wtr} S'$
Follows from Theorem 3.1.

Case: $S' \preceq_{wtr} S$
Follows from Lemma 3.9 and Lemma 3.5. □

Lemma 3.11.
Let Spec be a specification and Spec′ be the result of transforming Spec according to the rules of Table 4. Let S be the LTS derived from Spec by applying the rules of Tables 1, 2, 3 to Spec, and S' be the LTS obtained from Spec′ by applying the rules of Table 5, rules IN *and* OUT *of Table 2, rules of Table 6. Let $\mathcal{L}$ and $\mathcal{L}'$ be interpretation functions for the set of atomic proposition $\mathcal{P}$ mentioning only process variables. Then $(S, \mathcal{L}) \equiv_{st} (S', \mathcal{L}')$.*

Proof. Both $(S, \mathcal{L})$ and $(S', \mathcal{L}')$ are doubly LTSs (Def. 2.25). Here we assume that neither system S nor system S' contains deadlocks, because even if the system cannot proceed, time can progress. So all the traces of S and S' are infinite.

Case: $(S, \mathcal{L}) \preceq_{st} (S', \mathcal{L}')$
Follows from Def. 3.12 and Theorem 3.1.

Case: $(S', \mathcal{L}') \preceq_{st} (S, \mathcal{L})$
The expiration steps present in S and absent in S' do not change the valuation of process variables. By Lemma 3.9 and Lemma 3.6, it is straightforward to show that $(S', \mathcal{L}') \preceq_{st} (S, \mathcal{L})$ wrt. formulas mentioning process variables only. $(S, \mathcal{L}) \preceq_{st} (S', \mathcal{L}')$ and $(S', \mathcal{L}') \preceq_{st} (S, \mathcal{L})$ imply $(S, \mathcal{L}) \equiv_{st} (S', \mathcal{L}')$. □

Theorem 3.2.
For all formulas φ from LTL–X mentioning only process variables, $S \models \varphi$ iff $S' \models \varphi$.

Proof. Straightforward from Lemma 3.11 and Theorem 2.2. □

3.5 Conclusion

The transformation proposed in this chapter alleviates state explosion caused by the traditional SDL interpretation of time and timers. In the transformed model, timeouts are not placed into input queues but modelled by timeout guards, so we will have fewer possible combinations of messages in the input queues. Treating timers as variables is simpler than treating them as signals. There is no global time in the transformed system, thus the factor leading to infinite state space is eliminated.

The transformation preserves both negative and positive results of verification. A branching simulation relation R (cf. Theorem 3.1) guarantees that any LTL–X-formula satisfied by the transformed model is satisfied by the original one. Path equivalence up to stuttering (cf. Lemma 3.11) allows us to find a trace of the original system equivalent to a counterexample trace found in the transformed system. The proof of path equivalence up to stuttering also connects the presented subset of SDL with DTPromela that can be considered as an implementation of the "timers as variables" idea. The semantics of the transformed model will also be used in Chapter 5 to present an approach to automatic closing of SDL specifications.

4

Using Fairness to Make Abstractions Work

The state explosion remains a stumbling block of model checking. Abstraction techniques help to solve this problem replacing one model by an abstract smaller one. Here we propose a timer abstraction and argue its correctness.

Abstractions often introduce infinite traces that have no corresponding traces at the concrete level and can lead to the failure of the verification. We show how one can exclude them by imposing a strong fairness constraint on the abstract model.

By employing the fact that the timer abstraction introduces a self-loop, we render the strong fairness constraint into a weak fairness constraint and embed it into the verification algorithm.

The chapter is based on [26].

4.1 Introduction

Currently, most model checkers provide facilities to (automatically) reduce a state space, like partial-order reduction techniques. These techniques deal mainly with the control flow of a model. However, data (values stored and transmitted in a system), whose domain is often infinite or very large, are not handled by them; it is a task of a user to present data in a verification model in a finite form of reasonable size.

Abstraction techniques are widely used to make the verification of complex/parameterised/infinite systems feasible. Abstraction intuitively means replacing a semantical model by an abstract, in general simpler one. Depending on the property to be verified, the actual values of data may sometimes be ignored or replaced by some abstract values. In an abstract model, the operations on data are mimicked by new ones on the abstract data. The main requirement for an abstraction is that the abstract system behaviour should correctly reflect the behaviour of the original system with respect to a verification task in the sense that (1) an abstraction should capture all essential points in the system behaviour, i.e., be not "too abstract", and (2) an abstraction should be *safe*, which means that every property checked to be true on the abstract model, holds for the concrete one as well. This allows the transfer of positive verification results from the abstract model to the concrete one.

The concept of safe abstraction is well-developed within the *Abstract Interpretation* framework [46, 47, 51]. The relation between the concrete model and its safe abstraction is formalized there as a requirement on the relation between the data operations of the concrete system and their abstract counterparts. Every value of the concrete state space is mapped by the *abstraction function* α into an abstract value that "describes" the concrete value. As an example consider the abstraction of integers into their signs in which e.g. -3 is mapped by α into **neg**. For every operation (function) f on the concrete level, an abstraction f^α needs to be defined which "mimics" f. In general, the abstraction can be *nondeterministic*. For example, addition $(+)$ over the integers is abstracted into an operation $(+^\alpha)$ such that **pos** $+^\alpha$ **neg** may yield **pos** or **neg** nondeterministically. This is formally captured by letting f^α be a function into the powerset over the domain of abstract values.

Working within the Abstract Interpretation framework guarantees the preservation (in the direction from the abstract to the concrete model) of the truth of formulas of temporal logics without existential quantification over paths, e.g. $\Box L_\mu$ (cf. Theorem 2.5). Counterexamples can be spurious. In case a counterexample is found, the abstraction should be refined and the refined model is then model-checked. Such a sequence of refinements can happen to be infinite; in this case one needs different techniques to prove or disprove the property.

The systems we consider are specified as parallel compositions of communicating processes. A process consists of a number of locations, variables and transitions connecting the locations and changing the valuations of variables. Processes can communicate by rendezvous/buffered message passing and through

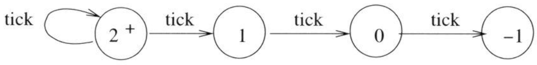

Fig. 5. Abstracted timer

shared memory. There are explicit timing constraints in the specification imposed by timer operations.

We assume that the properties are given in the universal fragment of the μ-calculus, $\Box L_\mu$, consisting of formulas in positive normal form, where the negations are applied only to atomic propositions. Since every L_μ formula has an equivalent in positive normal form (see [112]), this is not a significant loss of generality. The verification methodology we propose works for any formula of the universal fragment without negation $\Box L_\mu^+$ and, under certain conditions that occur relatively often in practice (for instance, if the formula does not refer to abstracted variables), for the whole $\Box L_\mu$.

In this chapter, we consider a simple abstraction for (discrete) timers similar to the one from [49]. This abstraction is often used to prove that a property holds for all instantiations of timer settings that are greater than or equal to some value k. It leaves all values below k unchanged and maps all other values to the abstract value k^+. Being a deterministic operation on the concrete model, the time progress operation *tick* becomes nondeterministic on the abstract one (see Fig. 5). That introduces infinite traces with $k^+ \rightarrow_{tick} k^+$ being chosen whenever *tick* is enabled. As a result the timer never expires, which in general does not correspond to any trace of the concrete model. For instance, properties of the form $\Box(\phi \rightarrow \Diamond\psi)$ get disproved on the abstract model whenever they depend on the fact that the timer in question eventually expires after being set. Refining the model by taking a greater value k_{new}, we still keep the loop at k_{new}^+. So refinement gives no solution to this problem.

To exclude the infinite loop $k^+ \rightarrow_{tick} k^+$ that causes spurious counterexamples, we impose a strong fairness condition Φ^α on the abstract model, which we call *t-fairness*: "For any trace where $k^+ \rightarrow_{tick} (k-1)$ is enabled infinitely often, $k^+ \rightarrow_{tick} (k-1)$ is taken infinitely often or t is set to a new value infinitely often". We show that the concrete property Φ that corresponds to the t-fairness condition Φ^α trivially holds on the concrete model. Therefore, in order to prove a formula ϕ on the concrete system, we check validity of the formula $\Phi^\alpha \rightarrow \phi^\alpha$ on the abstract one, where ϕ^α is the corresponding abstract version of ϕ. If $\Phi^\alpha \rightarrow \phi^\alpha$ holds, we conclude that ϕ holds on the concrete system.

By exploiting some specifics of the class of systems we are working with, we show that the strong fairness criterion can be reformulated into a weak fairness criterion. When one deals with explicit model checking, this is often a significant advantage, because algorithmically it could be easier to deal with the latter.

Moreover, when one stays in the realm of explicit-state model checking, it is much more efficient to build the t-fairness check into the model checking algorithm, than to express it as a formula. In this case, one can check for the validity of ϕ on the abstract model, assuming a built-in t-fairness check. The t-fairness check algorithm we propose here is inspired by Choueka's flag algorithm [36], and it is a version of the algorithm for weak process fairness which is implemented in *Spin*.

We implemented our algorithm in *DTSpin* [24] (a discrete-time version of the Spin model checker [93]) and tested the prototype implementation on some examples from the literature with encouraging results.

Related work. Counter abstractions similar to the timer abstraction we use are quite standard and they can be traced to [123]. Such abstractions are often applied to abstract (discrete) timers for the verification of *safety* properties (see e.g. [49]). We study here the verification of *liveness* properties, which gives rise to the use of fairness requirements on the abstract model.

There are several papers that deal with the problem of eliminating spurious execution sequences caused by abstraction. Closest to our approach is the theory of linear abstraction from [108] (also described in [109]). The general method of data abstraction presented there can also suffer from the problem of spurious execution sequences. To eliminate those, it is suggested to augment the system under consideration by an auxiliary monitoring module (executed synchronously with the system) and then to abstract the system obtained by such a composition. In one of the examples, [108] features a three-valued counter abstraction ($\{0, 1, 2^+\}$, using our notation). Thus, one could apply the idea of a monitoring process to eliminate extra sequences introduced by self-loops to abstract states. However, this would lead to a solution based on strong fairness on the transition level. The monitor labels the "critical" transitions with -1 or $+1$. The (strong) fairness criterion requires that if a -1 transition is executed infinitely often then also a $+1$ transition is executed infinitely often. This ensures leaving the artificial self-loops in the abstract state space introduced by the abstraction.

As it was already emphasized, we show that in the context of timer abstraction, such a straightforward strong fairness can be transformed into a weak one, which is a significant advantage in the context of explicit model checking.

In [138] the authors present a three-valued counter abstraction in the context of the verification of parameterized systems, i.e., networks of N identical concurrent processes, where N is an arbitrary finite number. The counters count the number of processes at a particular control (program) location. The solution to the problem of spurious execution sequences in this case also boils down to strong fairness. To this end two new variables *from* and *to* are introduced. The unwanted self-looping sequences are eliminated by the natural requirement that for each process location l if the processes enter l infinitely many times, then they must also leave it infinitely many times.

The problem of parameterized networks of processes is also treated in [14], with a solution for the spurious sequences which resembles both of the above given approaches. The role of the monitors from [108] is played by "ranking functions", similar to the ones used to ensure termination of sequential programs. The ranking functions count how many processes have executed a particular transition in the concrete system. By abstracting a ranking function value, similarly to [138], one obtains a separation of the "critical" transitions into "negative" and "positive" ones. The "marking algorithm" which solves the problem of spurious sequences is based on strong fairness. The efficiency remarks in favor of our solution in the context of explicit model checking would also apply to [14] and [138].

α-Spin [69] is an extension of *Spin* with abstraction. The abstraction framework of *α-Spin* is based on the Abstract Interpretation theory and in that regard it is similar to our approach. However, to the best of our knowledge, there is no work that deals with spurious executions in the context of *α-Spin*. Another approach to use abstractions in combination with *Spin* can be found in [93].

The chapter is organized as follows: In Section 4.2 we describe the timer abstraction and prove that it is safe. In Section 4.3 we introduce the notion of t-fairness. In Section 4.4 we present the verification algorithm. In Section 4.5 we describe our implementation of t-fairness in *DTSpin*. In Section 4.6 we discuss some experimental results. Finally, we give some conclusions in Section 4.7.

4.2 Timer Abstraction

Abstraction of Temporal Formulas

Given a system specification M whose semantics is given by transition system $T = (S, R)$, a system property is usually formulated as a formula ϕ of temporal logic and the main question of model checking is formulated as $T \models \phi$. Given a description relation $\rho \subseteq S \times {}_{\alpha}S$ that gives for concrete states in S their "descriptions" in ${}_{\alpha}S$, we can derive a transition system ${}_{\alpha}T = ({}_{\alpha}S, {}_{\alpha}R)$ such that it is an *abstraction* of T (Def. 2.36). Data abstraction can influence system variables mentioned in the temporal formula. Further, we consider how an abstract temporal formula ϕ^{α} can be built from a concrete formula ϕ so that ${}_{\alpha}T \models \phi^{\alpha}$ implies $T \models \phi$.

First we define how to construct abstract versions of atomic propositions. Let $\mathcal{P}$ be the set of atomic propositions of ϕ. Since ${}_{\alpha}T$ is an abstraction of T, there exists a Galois connection (α, γ) from 2^{S} to $2^{{}_{\alpha}S}$ (see Def. 2.36 and Lemma 2.3). We define p^{α} to be the proposition that corresponds to the subset $\alpha(\mathcal{I}(p)) - \alpha(\overline{\mathcal{I}(p)})$ of the abstract state space ${}_{\alpha}S$, obtained under the Galois connection (α, γ). We say that p^{α} is the *contracting* abstraction of p under α. The notion of contracting abstraction is similar to one given in [107].

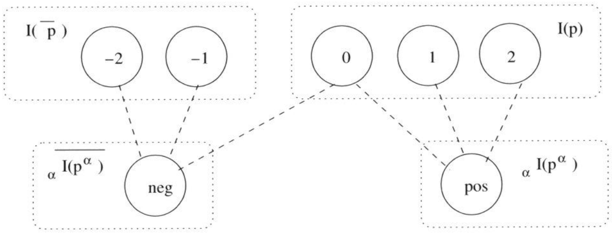

Fig. 6. Contracting abstraction

Definition 4.1. [CONTRACTING ABSTRACTION]
Let p be a proposition from $\mathcal{P}$, (α, γ) be a Galois connection, and $\mathcal{I}\colon \mathcal{P} \to 2^S$ and ${}_\alpha\mathcal{I}\colon \mathcal{P} \to 2^{{}_\alpha S}$ be two interpretation functions for transition systems $T = (S, R)$ and ${}_\alpha T = ({}_\alpha S, {}_\alpha R)$ respectively. p^α is the contracting abstraction *of p under α iff ${}_\alpha\mathcal{I}(p^\alpha) = \alpha(\mathcal{I}(p)) - \alpha(\overline{\mathcal{I}(p)})$.*

The contracting abstraction of a formula ϕ is ϕ^α that is obtained by replacing each atomic proposition p in ϕ with its contracting abstraction p^α.

Fig. 6 gives an example of a contracting abstraction. There concrete states $\{-2, -1, 0\}$ are mapped to abstract state $\{neg\}$, and concrete states $\{0, 1, 2\}$ are mapped to $\{pos\}$. We want to build a contracting abstraction for the atomic proposition $(x \geq 0)$. It is satisfied on concrete states $\{0, 1, 2\}$ and not satisfied on concrete states $\{-2, -1\}$, i.e. $\mathcal{I}(p) = \{0, 1, 2\}$ and $\overline{\mathcal{I}(p)} = \{-2, -1\}$. By abstracting $\{0, 1, 2\}$, we get $\{neg, pos\}$. We remove neg, because it can be mapped back to -2 or -1, where the concrete proposition is not satisfied, i.e. ${}_\alpha\mathcal{I}(p^\alpha) = \{pos\}$. The contracting abstraction of the concrete proposition $(x \geq 0)$ is the proposition corresponding to the set of abstract states ${}_\alpha\mathcal{I}(p^\alpha) = \{pos\}$. Note that for all $s \in S, s^\alpha \in S^\alpha$ such that $s^\alpha \in \alpha(\{s\})$, $s \models p$ if $s^\alpha \models p^\alpha$ when p^α is contracting. The contracting abstraction p^α of $(x \geq 0)$ is not satisfied in abstract state neg. However, abstract state neg can be mapped back to concrete state 0, where the concrete proposition $(x \geq 0)$ is satisfied. $s^\alpha \not\models p^\alpha$ does not imply $s \not\models p$.

To be able to transfer not only positive results but also negative results from the abstract system to the original one, we define the notion of *consistent* abstraction. Abstraction function α is called consistent with interpretation $\mathcal{I}$ (Def. 2.37), if the images by α of the interpretations of p and $\neg p$ are disjoint.

Definition 4.2. [CONSISTENT ABSTRACTION]
Let α be consistent with interpretation $\mathcal{I}$ and p^α be the contracting abstraction of p under α. We call the proposition p^α the consistent abstraction *of p under α.*

The consistent abstraction of a formula ϕ is ϕ^α that is obtained by replacing each atomic proposition p in ϕ with its consistent abstraction p^α.

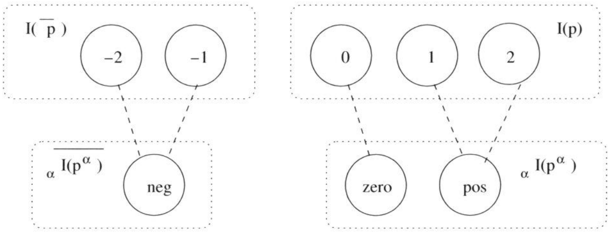

Fig. 7. Consistent abstraction

Fig.7 gives an example of a consistent abstraction. Concrete states $\{-2,-1\}$ are abstracted to $\{neg\}$, Concrete states $\{1,2\}$ are mapped to $\{pos\}$, and $\{0\}$ is mapped to $\{zero\}$. We want to build a consistent abstraction of an atomic proposition $(x \geq 0)$. The concrete proposition is satisfied on $\{0,1,2\}$ and not satisfied on $\{-2,-1\}$. By abstracting concrete states $\{0,1,2\}$, we get $\{zero, pos\}$. The contracting abstraction of the proposition is the proposition corresponding to the set of abstract states ${}_{\alpha}\mathcal{I}(p^{\alpha}) = \{zero, pos\}$. The consistent abstraction p^{α} is not satisfied on neg. Abstract state neg is concretized to $\{-2,-1\}$, where $\neg p$ is satisfied. Note that for all $s \in S, s^{\alpha} \in S^{\alpha}$ such that $s^{\alpha} \in \alpha(\{s\})$, $s \models p$ iff $s^{\alpha} \models p^{\alpha}$, when p^{α} is consistent.

Lemma 4.1.
Let (α,γ) be a Galois connection and α be surjective and consistent with $\mathcal{I}$. Then $\widetilde{\gamma}$ is consistent with ${}_{\alpha}\mathcal{I}$ defined as in Def. 4.1.

Proof. First, we have to show that if α is consistent with $\mathcal{I} \colon \mathcal{P} \to 2^{S}$, then $\forall p \in \mathcal{P},\ \widetilde{\gamma}(\alpha(\mathcal{I}(p))) = \mathcal{I}(p)$.

By the definition of $\widetilde{\gamma}$, $\widetilde{\gamma}(\alpha(\mathcal{I}(p))) = \overline{\gamma(\overline{\alpha(\mathcal{I}(p))})}$. Since α is consistent and surjective, $\overline{\gamma(\overline{\alpha(\mathcal{I}(p))})} = \overline{\gamma(\alpha(\overline{\mathcal{I}(p)}))}$. By Lemma 2.5, $\overline{\gamma(\alpha(\overline{\mathcal{I}(p)}))} = \overline{\overline{\mathcal{I}(p)}} = \mathcal{I}(p)$.

Further, we have to show that if α is consistent with $\mathcal{I} \colon \mathcal{P} \to 2^{S}$, then $\widetilde{\gamma}$ is consistent with ${}_{\alpha}\mathcal{I} \colon {}_{\alpha}\mathcal{P} \to 2^{{}_{\alpha}S}$. According to Def. 4.1, ${}_{\alpha}\mathcal{I}(p^{\alpha}) = \alpha(\mathcal{I}(p)) - \alpha(\overline{\mathcal{I}(p)})$. $\widetilde{\gamma}$ is consistent with ${}_{\alpha}\mathcal{I}$ iff

$$\widetilde{\gamma}({}_{\alpha}\mathcal{I}(p^{\alpha})) \cap \widetilde{\gamma}(\overline{{}_{\alpha}\mathcal{I}(p^{\alpha})}) = \emptyset.$$

By the definition of ${}_{\alpha}\mathcal{I}(p)$, we can rewrite the left part of this equality as

$$\widetilde{\gamma}[\alpha(\mathcal{I}(p)) - \alpha(\overline{\mathcal{I}(p)})] \cap \widetilde{\gamma}[\overline{\alpha(\mathcal{I}(p)) - \alpha(\overline{\mathcal{I}(p)})}].$$

Since α is consistent with $\mathcal{I}$, $\alpha(\mathcal{I}(p)) \cap \alpha(\overline{\mathcal{I}(p)}) = \emptyset$, and hence, $\alpha(\mathcal{I}(p)) - \alpha(\overline{\mathcal{I}(p)}) = \alpha(\mathcal{I}(p))$. Therefore, we rewrite the expression further to

$$\widetilde{\gamma}[\alpha(\mathcal{I}(p))] \cap \widetilde{\gamma}[\overline{\alpha(\mathcal{I}(p))}].$$

Now we only have to show that $\overline{\alpha(\mathcal{I}(p))} = \alpha(\overline{\mathcal{I}(p)})$. Since α is consistent, $\alpha(\mathcal{I}(p)) \cap \alpha(\overline{\mathcal{I}(p)}) = \emptyset$. Since α is surjective, $\alpha(\mathcal{I}(p)) \cup \alpha(\overline{\mathcal{I}(p)}) = {}_{\alpha}S$. Therefore, $\overline{\alpha(\mathcal{I}(p))} = \alpha(\overline{\mathcal{I}(p)})$. By Lemma 2.5, we have

$$\widetilde{\gamma}(\alpha(\mathcal{I}(p))) \cap \widetilde{\gamma}(\alpha(\overline{\mathcal{I}(p)}) = \mathcal{I}(p) \cap \overline{\mathcal{I}(p)} = \emptyset.$$

□

Theorem 4.1.
Let $T = (S, R)$ and ${}_{\alpha}T = ({}_{\alpha}S, {}_{\alpha}R)$ be two transition systems with interpretation functions $\mathcal{I}, {}_{\alpha}\mathcal{I}$ defined as in Def. 4.1. Let $T \sqsubseteq_{(\alpha,\gamma)} {}_{\alpha}T$. Given a $\Box L_{\mu}^{+}$ (respectively $\Box L_{\mu}$) formula ϕ, let ϕ^{α} be a contracting (respectively consistent with $\mathcal{I}$) abstraction of ϕ. Then ${}_{\alpha}T \models \phi^{\alpha}$ implies $T \models \phi$.

Proof. By Lemma 4.1, the consistency of α with $\mathcal{I}$ implies the consistency of $\tilde{\gamma}$ with ${}_{\alpha}\mathcal{I}$. The theorem is a corollary of Theorem 2.5. □

Often it is more convenient to apply abstractions directly on system specification M than on its transition system T. Such an abstraction on the level of M is well-developed within the *Abstract Interpretation* framework [46, 47, 51]. The requirement that Abstract Interpretation imposes on the relation between the concrete model T and its safe abstraction T^{α} can be formalized as a requirement on the relation between the data and the operations of the concrete system and their abstract counterparts as follows: Each value of the concrete domain Σ is mapped by a *description function* ρ_d to a value from the abstract domain ${}_{\alpha}\Sigma$. The abstract value "describes" the concrete value. We assume an ordering $\preceq$ on the abstract domain ${}_{\alpha}\Sigma$ according to the "precision" of abstract values: given a concrete value x and its abstract description $x^{\alpha} = \rho_d(x)$, we say that any $y^{\alpha} \in {}_{\alpha}\Sigma$ such that $x^{\alpha} \preceq y^{\alpha}$ is a *less precise* description of x.

For every operation (function) f on the concrete data domain, an abstract function f^{α} is defined, which "mimics" f. (For simplicity, we assume f to be a unary operation.) In general, the abstraction can be nondeterministic. This is formally captured by letting f^{α} be a function into the *powerset* over the domain of abstract values. The requirement of mimicking is then formally phrased with the following *safety statement*:

$$\forall x \in \Sigma \; \exists y \in f^{\alpha}(\rho_d(x)) : \rho_d(f(x)) \preceq y. \tag{1}$$

Definition 4.3. SAFE ABSTRACTION
Let M^{α} be obtained by replacing each constant c and function f of M with their abstract versions. We say that M^{α} is a safe abstraction *of M iff the safety statement is satisfied for all the abstract versions of the functions.*

A state s can be seen as a valuation vector $\langle v_0, v_1, \ldots, v_{n-1}\rangle$ and, thus, $S = \Sigma_0 \times \ldots \times \Sigma_{n-1}$, with $\Sigma_0, \ldots, \Sigma_{n-1}$ being the corresponding data domains. We relate S and ${}_{\alpha}S$ via the description relation, which in our case is the function $\rho_s : S \rightarrow {}_{\alpha}S$ defined as $\rho_s = \langle \rho_{d0}(v_0), \ldots, \rho_{dn-1}(v_{n-1})\rangle$, where $\rho_{0d}, \ldots, \rho_{dn-1}$

$$\begin{tikzcd}
M \arrow[d] \arrow[dr, dashed] & \\
T \arrow[d, dashed] & M^{\alpha} \arrow[d] \\
{}_{\alpha}T & T^{\alpha}
\end{tikzcd}$$

Fig. 8. Two approaches to abstraction

are description functions for the corresponding variables and the set of abstract states ${}_{\alpha}S = {}_{\alpha}\Sigma_0 \times \ldots \times {}_{\alpha}\Sigma_{n-1}$. We assume a trivial (identity) mapping as description function for unabstracted variables.

Figure 8 illustrates two approaches to abstraction. In the first case, transition system $T = (S, R)$ giving the semantics of specification M is abstracted to transition system ${}_{\alpha}T = ({}_{\alpha}S, {}_{\alpha}R)$ by a description relation on the states of T and ${}_{\alpha}T$. In the second case, M^{α} is obtained by replacing each constant c and function f of M by their abstract versions. Let M^{α} be a safe abstraction of M, and the semantics of M^{α} be given by $T^{\alpha} = (S^{\alpha}, R^{\alpha})$. Obviously $S^{\alpha} = {}_{\alpha}\Sigma_0 \times \ldots \times {}_{\alpha}\Sigma_{n-1} = {}_{\alpha}S$. Moreover, for "usual" modelling languages, like PROMELA, $R^{\alpha} \supseteq {}_{\alpha}R$, which follows from Lemma 4.4.1.1 of [47]. This trivially implies ${}_{\alpha}T \sqsubseteq_{id} T^{\alpha}$, where id is the identity function. Given a $\Box L_{\mu}^{+}$ (respectively $\Box L_{\mu}$) formula ϕ, we can find ϕ^{α} that is a contracting (respectively consistent with $\mathcal{I}$) abstraction of ϕ. By Theorem 4.1, we obtain that $T^{\alpha}, {}_{\alpha}\mathcal{I} \models \varphi^{\alpha} \Rightarrow {}_{\alpha}T, {}_{\alpha}\mathcal{I} \models \varphi^{\alpha} \Rightarrow T, \mathcal{I} \models \varphi$.

Timer Abstraction

The timer abstraction proposed here is similar to abstractions given in [49] and in [25]. The abstraction from [25] is based on a natural idea of allowing timers to expire at an *arbitrary moment* after they are set. This means that the timer can expire immediately after setting. A typical problem arising when one starts to apply this abstraction in practice is the introduction of zero-time cycles (cycles without time progression) which are not present in the concrete model. A usual pattern for SDL-specifications is that a timer schedules some periodical activity; after a timeout signal is consumed by a process, some actions are taken, the timer is set again, and the process returns to the same control state where it was before consumption of the timer signal. Since the abstraction allows a timer to expire at any arbitrary moment after its setting (also immediately after it has been set), undesirable cyclic behaviour can be introduced. The "timeout input – timer setting – timer expiration" chain of transitions can be executed infinitely many times and all the other behavioural branches may be ignored forever. Therefore, the properties which hold for the concrete model and which are expected to hold independently of the timer settings, fail to hold for the abstract model, since "independently" means in this case that the property holds for the concrete model whatever *positive* delay is assigned to a timer. Another problem arises in case a timer serves as a guard to prevent from taking a transition too early. By abstracting time, this timer guard is broken.

We propose an abstraction for timers that keeps this guard delaying the timer expiration. Moreover, we prove that the abstraction is safe. The concept of timers was defined in Section 3.3. The system semantics we use here is similar to one defined in Section 3.3. The only difference is that we do not require *tick* to have the least priority and leave the semantics of time partially open here, since our approach does not depend on it.

For a timer t, the concrete domain of timer values $\Sigma = \mathbb{N} \cup \{-1\}$, where -1 represents a deactivated timer, is replaced with the abstract domain ${}_{\alpha}\Sigma_t = \{-1, 0, \ldots, k_t - 1, k_t^+\}$, where the value k_t is a positive value defined by the user, assuming that the property we want to verify still holds even if we do not distinguish between the values of the timer greater than or equal to k_t. We overload the notation by using c $(-1 \leq c < k_t)$ as an abstract value representing the single concrete value c, while c^+ describes the set of concrete values $\{c, c+1, c+2, \ldots\}$. We do not consider 0^+ abstraction here.

The description function ρ_t is defined as $\rho_t(c) = c$ if $c < k_t$ and $\rho_t(c) = k_t^+$ otherwise. Abstract operations on timers are defined in an intuitive way: setting a timer to value x becomes setting it to value $\rho_t(x)$; the timeout guard g_t^α is *true* iff $[\![t]\!] = 0$; and $tick^\alpha$ is a nondeterministic operation that changes the value of a timer from a to b according to the following rules: (1) if $a = -1$ then $b = -1$, (2) if $0 \leq a < k_t$ then $b = a - 1$ (where "$-$" works on abstract values as on integers), (3) if $a = x^+$ then $b \in \{x^+, x - 1\}$.

Varying k_t, we can change the refinement degree of the abstraction. Taking k equal to 0, we get the most abstract version of it, which is an abstraction from [25] where not just timers but time is abstracted. Taking k equal to the lower boundary of the timer delays in the system, we get the most refined abstraction that can be obtained with this abstraction schema.

Lemma 4.2.
System M^α built from system M according to the rules given above is a safe abstraction of M.

Proof. Let M^α be obtained by replacing each constant c and function f of M with their abstract versions, and $T^\alpha = (S^\alpha, R^\alpha)$ be the transition system that corresponds to M^α. Now we check whether the safety statement holds for all functions on timers. For the timer abstraction, we reformulate the safety statement as: $\forall x \in \Sigma\ \exists y \in f^\alpha(\rho_d(x)) : \rho_d(f(x)) = y$.

Abstract setting of a timer obviously satisfies this safety statement. Since zero value of a timer is always mapped by the description function to zero, g_t^α also satisfies the statement.

For $tick^\alpha$ we consider three cases: Let x be the value of the timer t and (i) $x < k_t$, (ii) $x = k_t$ or (iii) $x > k_t$.

In the first case, *tick* decreases the value of the timer to $(x-1)$. Since $x < k_t$, $(x-1)$ is mapped to $(x-1)$ by ρ_d. If we apply $tick^\alpha$ to the value of the abstract timer, it becomes $(x-1)$ as well, hence the safety statement is satisfied.

In the second case, *tick* decreases the value of the timer to $(x-1)$. Since $x = k_t$, $(x-1)$ is mapped to $(x-1)$ by ρ_d. Abstract function $tick^\alpha$ chooses nondeterministically one of the values from $\{k_t^+, (x-1)\}$. The safety statement is satisfied.

In the last case, $\rho_d(x-1) = k_t^+$. If we apply $tick^\alpha$ to the value of the abstract timer, $tick^\alpha$ chooses nondeterministically one of the values from $\{k_t^+, (k_t-1)\}$. Therefore, the safety statement is satisfied as well. □

4.3 Fair Timer Abstraction

From now on we assume that systems under consideration have neither deadlocks nor infinite zero-time cycles (infinite traces with a finite number of *tick*'s). Note that if system contains a zero-time cycle, there is an infinite trace of the system where time never progresses. Such a trace is absolutely unrealistic for a real system. If M^α is a safe abstraction of M, the absence of zero-time cycles can be checked on the abstract model by verifying the property $\Box\Diamond\mathbf{tick}^\alpha$, which is a consistent abstraction of $\Box\Diamond\mathbf{tick}$. (**tick** is a formula that encodes an occurrence of a *tick*-step in the original system. $\mathbf{tick}^\alpha$ is the consistent abstraction of **tick**.) The absence of deadlocks follows straightforwardly from the fact that time can progress even when no other action is possible in the system, and thus the *tick* action is still possible.

An abstracted system contains more behaviour than the original one. Therefore, positive verification results can be transferred from the abstract to the concrete system, while counterexamples can be spurious. Abstraction refinement is a common technique used in case spurious counterexamples are found (see e.g. [40]), though just a change of the granularity level does not always help—the sequence of refinements can turn out to be infinite.

Suppose we use the timer abstraction described in the previous section to prove that some property holds for all timer settings greater than or equal to some k_t. Due to the nondeterminism introduced with the abstract version of *tick*, it becomes possible that the timer once set will never expire. This means that the states that are always reachable in the concrete system are not reached in the abstract system if the $k_t^+ \rightarrow_{tick} k_t^+$ step is always chosen. Such a trace gives a spurious counterexample: In the concrete system the timer expires after a finite number of time slices. The only possible refinement is taking the same abstraction with a greater value of k. But the same trace where the timer never expires is still possible, so a counterexample would be produced again. Therefore, we need a different technique to cope with this problem.

Imposing a strong fairness condition that requires that for any trace where the transition $k_t^+ \rightarrow_{tick} (k_t-1)$ is infinitely often enabled it is infinitely often taken, gives incorrect results: One can easily build a (concrete) model where a timer t is infinitely often set to a new value (before it expires), so it can be seen every time as a new variable in the one-assignment framework. This observation leads us to the following definition of *t-fairness*:

Definition 4.4.
Given an LTS T of a system with a set of abstract timers $TVar^{\alpha}$. We say that a trace of T is t-fair *iff for any $t \in TVar^{\alpha}$ the following holds: $k_t^+ \rightarrow_{tick} (k_t - 1)$ is infinitely often enabled implies that $k_t^+ \rightarrow_{tick} (k_t - 1)$ is infinitely often executed or $set(t, x)$, $x \in {}_{\alpha}\Sigma_t$, is infinitely often executed.*

This definition has a *strong fairness* pattern. Interestingly, due to the fact that the loop introduced on a timer with the abstraction is a *self-loop*, this requirement can be reformulated as a condition with a *weak fairness* pattern:

Lemma 4.3.
A trace ξ of T is t-fair *iff for any $t \in TVar^{\alpha}$ the following holds: if there exists an infinite suffix σ of ξ such that $[\![t]\!]_{s_j} = k_t^+$ for every state of σ, then $set(t, k_t^+)$ is infinitely often executed along the trace.*

Proof. Let p, q, and r denote the following propositions (from Def. 4.4): "$k_t^+ \rightarrow_{tick} (k_t - 1)$ is enabled", "$k_t^+ \rightarrow_{tick} (k_t - 1)$ is executed", and "$set(t, x)$, $x \in {}_{\alpha}\Sigma_t$, is executed", respectively. Then the t-fairness condition from Def. 4.4 can be expressed as the following *LTL* formula:

$$\Box\Diamond p \rightarrow (\Box\Diamond q \vee \Box\Diamond r). \tag{2}$$

We can split the proposition r into a disjunction of two propositions r_1 and r_2: "$set(t, k_t^+)$ is executed" and "$set(t, x)$, where $x \neq k_t^+$, is executed", respectively. After straightforward transformations, (2) becomes

$$\neg(\Box\Diamond p \wedge \Diamond\Box(\neg q \wedge \neg r_1)) \vee \Box\Diamond r_2. \tag{3}$$

We will show that $\Box\Diamond p \wedge \Diamond\Box(\neg q \wedge \neg r_1)$ (*), is semantically equivalent to $\Diamond\Box p'$, where p' denotes the proposition "the value of t is k_t^+".

The conjunct $\Box\Diamond p$ says that $k_t^+ \rightarrow_{tick} (k_t - 1)$ is infinitely often enabled. Since we assume the absence of zero-time cycles, by the timer abstraction definition, this implies the proposition "timer t has value k_t^+ infinitely often". The conjunct $\Diamond\Box(\neg q \wedge \neg r_1)$ says that after some point in the execution sequence neither $k_t^+ \rightarrow_{tick} (k_t - 1)$ nor $set(t, x)$, with $x \neq k_t^+$, are executed. As these transitions are the only ones that can change the value of t from k_t^+ to a value different than k_t^+, we can conclude that from some point on the value of t will remain k_t^+ forever.

For the other direction, we first observe that if t has value k_t^+ from some point on, then $k_t^+ \rightarrow_{tick} (k_t - 1)$ is enabled infinitely many times. (Again, we use the absence of zero-time cycles, i.e., a *tick* transition is executed infinitely often along any execution sequence.) Also, the other conjunct of (*) follows immediately: As $k_t^+ \rightarrow_{tick} (k_t - 1)$ and $set(t, x)$, where $x \neq k_t^+$, are the only statements which can change the value of the abstract timer t to a value different from k_t^+, they also cannot be executed after some point on.

Thus, we can replace (*) with the equivalent proposition $\Diamond\Box p'$ and rewrite (3) as $\Diamond\Box p' \rightarrow \Box\Diamond r_2$, which is the (weak t-fairness) condition of Lemma 4.3.

□

Thus we can express the t-fairness criterion by the following *LTL* formula $\Phi^\alpha = \bigwedge_{t\in TVar^\alpha}(\Diamond\Box p \rightarrow \Box\Diamond q)$, where p and q are propositions corresponding to the terms "$[\![t]\!]_{s_j} = k_t^+$" and "$set(t, k_t^+)$" from Lemma 4.3, respectively. Though this property is formulated on states *and transitions*, it can be easily encoded as a property defined on the states of the system. (To express the fact that some transition q is taken infinitely often, one can e.g. extend the model with introducing a boolean variable b_q that is negated every time the transition is taken and replace $\Box\Diamond q$ with $\Box\Diamond b_q \wedge \Box\Diamond\neg b_q$.) One can see the analogy between Φ^α and the definition of *weak fairness for processes*, where a timer set to k_t^+ corresponds to an enabled process and an execution of the *set* operation corresponds to an execution of an action by the process.

Further, one can show that the t-fairness criterion Φ^α is a consistent (also contracting) abstraction of the *LTL* formula $\Phi = \bigwedge_{t\in TVar}(\Diamond\Box p' \rightarrow \Box\,\Diamond q')$, where p', q' are defined as "$[\![t]\!]_{s_j} \geq k_t$" and "$set(t, x)$, where $x \geq k_t$", respectively. This can be done by a simple check that p and q are consistent abstractions of p' and q', respectively. Indeed, let $s^\alpha \in \alpha(\{s\})$. Timer t has the value k_t^+ in the abstract state s^α iff t has a value greater than or equal to k_t in s. Similarly, t is set to some x which is greater than or equal to k_t by a transition which has s as the target state iff it is set by a transition in the abstract state which ends up in the state s^α with $[\![t]\!]_{s^\alpha} = k_t^+$.

Suppose we want to verify that $T \models \phi$ for some $\Box L_\mu^+$ (resp. $\Box L_\mu$) formula ϕ and a concrete system T without infinite zero-time traces. The "concrete" version of the abstract t-fairness condition, Φ, holds on any trace of T: If from some point on the value of timer t remains greater than or equal to k_t, then the timer must be infinitely often set to some value greater than k_t. Otherwise, since *tick* happens infinitely often, the value of t will eventually become less than k_t. Thus, $T \models \phi$ iff $T \models (\Phi \rightarrow \phi)$.

By Theorem 4.1 we know that instead of verifying $T \models (\Phi \rightarrow \phi)$ on the concrete system, we can verify its contracting (resp. consistent) abstraction $(\Phi \rightarrow \phi)^\alpha$ on the abstract system. By definition of contracting (consistent) abstraction, the last formula is equivalent to $\Phi^\alpha \rightarrow \phi^\alpha$. In case ϕ does not refer to variables (timers) that are abstracted, the abstraction α is trivially a consistent abstraction for all atomic propositions in ϕ and we have $\phi^\alpha = \phi$. If ϕ does mention abstracted timers, one has to derive the contracting abstraction ϕ^α of ϕ.

Finally, by Theorem 4.1, $T^\alpha \models (\Phi^\alpha \rightarrow \phi^\alpha)$ implies $T \models (\Phi \rightarrow \phi)$ and thus also $T \models \phi$.

So, by imposing the t-fairness condition on the abstract model, we eliminate spurious counterexamples caused by unfair nondeterministic choices made by abstract functions.

4.4 Incorporating t-Fairness into the Verification Algorithm

To express the formula Φ^α as an *LTL* formula defined on the states of the system, one needs to introduce additional variables (see Section 4.3). Therefore, it is computationally expensive to verify the formula $\Phi^\alpha \rightarrow \phi^\alpha$ and it is more convenient to incorporate the t-fairness requirement into the verification algorithm that verifies ϕ^α by considering t-fair traces only. In this section we describe how to embed the t-fairness check into a model-checking algorithm for *LTL*.

Since there is a strong analogy between *t-fairness* and *weak process fairness*, one can easily adapt any algorithm for model checking under weak process fairness. The algorithm we propose here is inspired by the weak process fairness algorithm used in Spin [93, 22], which is a combination of the Nested Depth First Search (*ndfs*) algorithm (see Algorithm 2.3) and Choueka's flag algorithm [36]. In the automata-theoretic approach (see Section 2.4), the negation of the formula is translated into a Büchi automaton and satisfaction of the *LTL* formula is proven by detecting acceptance cycles in the synchronous product of the Büchi automaton for the negation and a Büchi automaton representing the system. A Büchi automaton can also be seen as an LTS with a predefined set of accepting states, so the satisfaction of an *LTL* formula can also be proven by detecting acceptance cycles in an *extended* LTS that is the synchronous product of an L^{2}TS representing the system and the Buchi automaton for the formula (see [95]). Further we assume that we work directly with an extended LTS that is defined as follows:

Definition 4.5. EXTENDED LTS *[95]*
Let $\mathcal{D} = (P, Lab, \rightarrow, p_0, \mathcal{L})$ be an L^{2}TS and $B = (Q, \delta, q_0, F)$ be a Büchi automaton over the alphabet $2^{\mathcal{P}}$.

The extended *LTS $\mathcal{E}$ is given by a tuple $(P \times Q, Lab, \rightarrow', (p_0, q_0), P \times F)$ where labelled transition relation $\rightarrow' \subseteq (P \times Q) \times Lab \times (P \times Q)$ is defined as follows: $((p,q), \alpha, (p', q')) \in \rightarrow'$ iff $(p, \alpha, p') \in \rightarrow$, $(q, a, q') \in \delta$ and $\mathcal{L}(s) = a$.*

Given an extended LTS $\mathcal{E} = (S, Lab, \longrightarrow, s_{init}, F)$, which is the synchronous product of the L^{2}TS of a given abstract system with the Büchi automaton that represents the negation of a property to be verified, our goal is to construct an extension of $\mathcal{E}$ that contains an acceptance cycle iff there exists a *t-fair acceptance cycle* in $\mathcal{E}$. (We say that a cycle $s_0 \xrightarrow{a_0} ...s_n \xrightarrow{a_n} s_0$ is t-fair iff $\forall t \in TVar^\alpha$ there exists i $(0 \leq i \leq n)$ such that $[\![t]\!]_{s_i} \neq k_t^+$ or $a_i = set(t, k_t^+)$.) Therefore, we will define this extension in such a way that any acceptance cycle would be t-fair by construction.

Let the abstract system have N abstract timers. Then we construct the extended LTS $\mathcal{E}' = (S', Lab', \longrightarrow', s'_{init}, F')$ in the following way: The set of states of the extended system is a set of pairs (s, c), where $s \in S$ and $0 \leq c \leq N$. We call (s, c) a *c-replica* of s. (Note that not every replica (s, c) of a reachable state s of $\mathcal{E}$ will be reachable in $\mathcal{E}'$.) 0-replicas are the basic replicas of the states, while replicas $1, \ldots, N$ allow to track the behaviour of abstract timers $t_1, \ldots, t_N$, respectively. The accepting states and the initial state of $\mathcal{E}'$ are 0-replicas of the accepting states and the initial state of $\mathcal{E}$, respectively. All transitions from accepting states of $\mathcal{E}'$ lead to 1-replicas only. Transitions from a c-replica (s, c), related to timer t_c, lead either to c-replicas, or, when they guarantee t-fair behaviour wrt. timer t_c, to the $((c+1)mod(N+1))$-replica. Since all the acceptance states of $\mathcal{E}'$ are 0-replicas, any acceptance cycle contains for every abstract timer at least one transition that either sets timer t to k_t^+ or results in a value of t different from k_t^+. So every acceptance cycle of $\mathcal{E}'$ is t-fair.

The verification algorithm starts the construction of $\mathcal{E}'$ from the initial state $(s_{init}, 0)$ and proceeds by adding the 0-replicas in accordance with the transition function $\longrightarrow$ until an accepting state is met. If an accepting state s is encountered, the algorithm adds a dummy τ-step that connects the 0-replica of s with the 1-replica of the same state. A move from a c-replica with $1 \leq c \leq N$ to the $((c+1) \mod (N+1))$-replica happens when a state is encountered in which t_c has a value different from k_t^+ or a step setting timer t_c is taken, i.e. *when the t-fairness condition for t_c is fulfilled.* (A move from a 0-replica to a 1-replica is possible only by τ-steps connecting the replicas of the same accepting state.) For the rest, the algorithm adds transitions following the transition function $\longrightarrow$.

Theorem 4.2.
Given an extended LTS $\mathcal{E} = (S, Lab, \longrightarrow, s_{init}, F)$ *and abstract timers* $t_1, \ldots, t_N$*, a t-fair extension of* $\mathcal{E}$ *is an extended* LTS $\mathcal{E}' = (S', Lab', \longrightarrow', s'_{init}, F')$ *that satisfies the following conditions:*

1. $Lab' = Lab \cup \{\tau\}$;
2. $s'_{init} = (s_{init}, 0)$;
3. $(s, 0) \xrightarrow{a}{}' (s', 0)$ *if* $(s, 0) \in S'$ *and* $s \xrightarrow{a} s'$ *and* $s \notin F$;
4. $(s, 0) \xrightarrow{\tau}{}' (s, 1)$ *if* $(s, 0) \in S'$ *and* $s \in F$;
5. $(s, c) \xrightarrow{a}{}' (s', c')$ *if* $(s, c) \in S'$ *and* $c > 0$ *and* $s \xrightarrow{a} s'$ *with* $c' = ((c+1)$ mod $(N+1))$ *if* $(\llbracket t_c \rrbracket_s \neq k_{t_c}^+$ *or* $a = set(t_c, k_{t_c}^+))$*, and* $c' = c$ *otherwise;*
6. $F' = \{(s, 0) \mid s \in F\}$.

Then the following statements hold:

1. $(S, Lab, \longrightarrow, s_{init})$ *and* $(S', Lab', \longrightarrow', s'_{init})$ *are branching bisimilar.*
2. $\mathcal{E}$ *contains a reachable* t-fair *acceptance cycle iff* $\mathcal{E}'$ *contains a reachable acceptance cycle.*

Proof. 1. Consider relation $Q \subseteq S \times S'$ where $(s, s') \in Q$ iff $s' = (s, c)$ where $0 \leq c \leq N$. It is straightforward to check by case analysis that Q is a branching bisimulation (Def. 2.20).

2. First we show that all acceptance cycles of the extended state space are t-fair by construction. An acceptance cycle contains at least one accepting state; this state is a 0-replica and has outgoing transitions to 1-replicas only. As transitions from a c-replica lead either to c-replicas, or to $((c+1) \mod (N+1))$-replicas $(0 \leq c \leq N)$, for any c, the cycle includes a c-replica. Every move from a c-replica to its neighbour satisfies the t-fairness condition for timer t_c, so for every abstract timer there is a transition in the cycle satisfying the t-fairness condition and thus the cycle is t-fair.

Due to the branching bisimulation result, any acceptance cycle of $\mathcal{E}'$ (which is always t-fair) has a corresponding t-fair acceptance cycle in $\mathcal{E}$.

In the opposite direction: Assume that there is a trace $s_{init} \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ of $\mathcal{E}$ that contains a fair acceptance cycle. Then there are s_i, s_j such that $s_i = s_j$ with $j > i$. The path π from s_i to s_j contains at most $m = (j - i)$ distinct states. Trace $\sigma = s_{init} \xrightarrow{a} ...s_i....s_i...s_i$ going through the cycle $N + 1$ times is also a trace of $\mathcal{E}$. Due to the branching bisimulation result, there is a trace σ' in $\mathcal{E}'$ that mimics σ. The suffix ξ' of σ' that mimics passing through the cycle $N + 1$ times contains at least $m(N + 1)$ transitions, so it visits at least $m(N + 1) + 1$ states. The states of ξ' are replicas of the states of π, therefore at most $m(N+1)$ of them are distinct. Thus, there is at least one state that is present in ξ' twice, and ξ' is a cycle.

Now we shall show that ξ' is an *acceptance* cycle. We denote the suffix of σ corresponding to ξ' as ξ and pick an arbitrary state s of ξ. Then ξ' contains some state (s, c), $0 \leq c \leq N$. Assume that $c > 0$ (for else we are done). Since ξ is a t-fair cycle, there are some states q_1, q_2 reachable from s such that $q_1 \xrightarrow{a}_{\mathcal{E}} q_2$ and $([\![t_c]\!]_{q_1} \neq k_t^+$ or $a = set(t_c, k_t^+))$. Hence, there exists a transition from the c-replica q_1 to the $((c+1) \mod N)$-replica q_2 in ξ'. Proceeding in the same way, we will obtain transitions leading to some $((c + 2) \mod N)$-replica, etc., and eventually we arrive at a 0-replica. Thus, we conclude that ξ' contains at least one 0-replica of some state. In $\mathcal{E}'$, transitions from 0-replicas of non-accepting states lead to 0-replicas. Since ξ contains an accepting state and ξ' is a cycle, ξ' contains an accepting state as well and thus it is an acceptance cycle of $\mathcal{E}'$. □

We call the extension $\mathcal{E}'$ a *t-fair extension* of $\mathcal{E}$. An algorithm that generates the extended state space is given in Fig. 9. The algorithm is based on the depth first search (*dfs*) algorithm ([45]). It is straightforward to prove the following claim:

Lemma 4.4.
Given an extended LTS $\mathcal{E}$, *let* $\mathcal{E}'$ *be an extended* LTS *produced from* $\mathcal{E}$ *by applying Procedure 4.3. Then* $\mathcal{E}'$ *is a t-fair extension of* $\mathcal{E}$.

To detect acceptance cycles, *dfs* is extended with a cycle-check procedure (Fig. 10). Whenever Procedure 4.4 detects an accepting state, it starts Proce-

Procedure 4.3 (dfs(s, c))

```
add $(s, c)$ to $S'$                                         add a pair to the state space
if $c = 0$  and $s \in F$                                    0-replica and state $s$ is accepting
then  if $(s, 1) \notin S'$ then $dfs(s, 1)$;                $\tau$-step from 0-replica to 1-replica
else
  for all $s \xrightarrow{a} s'$ do                          for all transitions enabled in $s$
    if $c > 0$ and $(a = set(t_c, k^+_{t_c})$ or $[\![t_c]\!]_s \neq k^+_{t_c})$    t-fairness condition
    then $c' = (c + 1)$ mod $N$                              the next replica number
    else $c' = c$;                                           the same replica number
    if $(s', c') \notin S'$ then $dfs(s', c')$;              recursive call
  od;
```

Fig. 9. Generating t-fair extension of S

dure 4.5, which is again a *dfs*, that reports an accepting state if the seed state is matched within the cycle-check. Here we omit a detailed description of the *ndfs* algorithm and refer the interested reader to [45].

The correctness of the algorithm is given by the following claim:

Theorem 4.6.
Given an extended LTS $\mathcal{E}$, *Procedure 4.4 called with* $(s_{init}, 0)$ *reports an acceptance cycle iff there exists a reachable t-fair acceptance cycle in* $\mathcal{E}$.

Proof. Follows from the correctness of the *ndfs* algorithm from Lemma 2.1 by observing that the algorithm is actually *ndfs* from [45] applied on the extended state space $\mathcal{E}'$. □

The last result completes the series of claims that guarantee soundness of the verification approach proposed in this chapter. If no acceptance cycle is detected then the verified property holds for t-fair traces of the abstract system and therefore also for the concrete system.

Time complexity of the *ndfs* Algorithm in Fig. 10 is $O(N \cdot |\mathcal{E}|)$, where N is the number of timers, while $|\mathcal{E}|$ is the size (states and transitions) of the abstract system state space. Memory space needed to save $\mathcal{E}'$ is virtually the same as the one for $\mathcal{E}$. Instead of keeping each of the N replicas (s, i), $(1 \leq i \leq N)$ one can save only the "useful" part s plus additional $2(N + 1)$ bits, like it is done for process fairness in *Spin*([93, 23]). The first $N + 1$ bits correspond to the replicas in the main depth first search of the *ndfs* algorithm, while the second group of $(N + 1)$ bits corresponds to the nested *dfs*. If bit i of the first group is set then this means that the state (s, i) has been visited by the algorithm. Similarly for the second group. As the description of s is usually much greater than $2(N + 1)$ bits, the bookkeeping overhead is negligible [23].

Procedure 4.4 (ndfs$_1$(s, c))

```
add $(s, c, 0)$ to $S'$                                          add a pair to the state space
if $c = 0$ and $s \in F$                                         0-replica, and state s is accepting
then if $(s, 1, 0) \notin S'$ then $ndfs_1(s, 1)$;               τ-step from 0-replica to 1-replica
else
  for all $s \xrightarrow{a}_{\mathcal{E}} s_1$ do                for all transitions enabled in s
    if $c > 0$ and $(a = set(t_c, k^+_{t_c})$ or $[\![t_c]\!]_s \neq k^+_{t_c})$    t-fairness condition
    then $c' = (c + 1)$ mod $N$                                  the next replica number
    else $c' = c$;                                               the same replica number
    if $(s', c', 0) \notin S'$ then $ndfs_1(s', c')$;             recursive call
  od;
if $c = 0$ and $s \in F$ then $seed := (s, 0, 1)$; $ndfs_2(s, 0)$;   set the seed and start ndfs$_2$
```

Procedure 4.5 (ndfs$_2$(s, c))

```
add $(s, c, 1)$ to $S'$                                          add a pair to the state space
if $c = 0$ and $s \in F$                                         0-replica, and state s is accepting
then if $(s, 1, 1) \notin S'$ then $ndfs_2(s, 1)$;               τ-step from 0-replica to 1-replica
else
  for all $s \xrightarrow{a}_{\mathcal{E}} s'$ do                 for all transitions enabled in s
    if $c > 0$ and $(a = set(t_c, k^+_{t_c})$ or $[\![t_c]\!]_s \neq k^+_{t_c})$    t-fairness condition
    then $c' = (c + 1)$ mod $N$                                  the next replica number
    else $c' = c$;                                               the same replica number
    if $seed = (s, c', 1)$ then REPORT CYCLE!                     seed is matched, report the cycle
    else if $(s', c', 1) \notin S'$ then $ndfs_2(s', c')$;        recursive call
  od;
```

Fig. 10. *ndfs* version of Procedure 4.3

4.5 *T*-fairness in *DTSpin*

DTSpin [24] is a discrete-time extension of *Spin* [93] that has all verification features of *Spin*. It was successfully applied for debugging and verification of timed models of industrial size protocols (see e.g. [25, 103]). *DTSpin* is designed for the verification of systems where delays are significantly larger than the duration of the events within the system. Therefore, system transitions are assumed to be instantaneous. *DTSpin* employs the concept of timers to express time aspects of a system. In DTPROMELA, the input language of *DTSpin*, timers are modelled by variables of a predefined type *timer*. The data domain and the operations on timers are defined as in Section 4.2.

Since the system transitions are assumed to be instantaneous, time progress has the least priority in the system and may take place only when the system is *blocked*. A special process *Timer* ticks all the active timers down in case the system is blocked. *DTSpin* employs PROMELA's statement *timeout* to check whether the system is blocked. To ensure that time progression has the least priority, the usage of *timeout* is reserved for the implementation of time progression and forbidden in DTPROMELA specifications. Note that by the definition of *tick*, all DTPROMELA models are *deadlock-free*.

To implement the timer abstraction defined in Section 4.2, we extend DT-PROMELA with a new data type $timer^{\alpha}$ for abstract timers and define the operations on them as macros. The abstract version of *tick*, $tick^{\alpha}$, decreases values of active abstract timers if they are different from k_t^+. If a timer has the k_t^+ value, a nondeterministic choice is made between decreasing the value of the timer to $(k_t - 1)$ and leaving it unmodified. Our fairness algorithm from Section 4.4 is implemented by means of a PAN2TFPAN Java program that transforms the *pan* verifier generated by *Spin* ([93, 154]) for the verification of a property without t-fairness into a new one that checks the property under t-fairness. The transformation is automatic and does not require any interaction with the user. The prototype implementation PAN2TFPAN can be downloaded at `www.cwi.nl/~ustin/tfair.html`.

The user thus applies the following verification scheme : (1) Choose timers of a concrete model that should be abstracted and define a k_t value for each of those timers. (2) Redefine the type of the chosen times to $timer^{\alpha}$ and redefine the *set* operations according to the k_t values. (3) Check whether the abstract system is free from zero-time cycles, i.e. check whether *tick* happens infinitely often. This is done by checking *LTL* formula $\Box\Diamond timeout$. In *DTSpin*, time progresses if the statement *timeout* of PROMELA is *true*. Since this statement is forbidden to use in DTPROMELA specifications, $\Box\Diamond timeout$ expresses the absence of zero-time cycles. (4) Formulate the abstract version of the property to check and generate the *pan* verifier for this property. (5) Transform the *pan* verifier with PAN2TFPAN to the new *pan* verifier, which checks the property under the t-fairness condition. Positive verification results imply that the property holds for the concrete system as well. If the property gets violated on the abstract system, a counterexample is generated, and the user checks whether the counterexample is spurious or not.

4.6 Experimental Results

In this section we describe some experimental results that show the efficiency of our approach. Our test cases are the positive acknowledgment retransmission protocol (PAR) [155] and Fischers mutual exclusion protocol [118]. We compare the results obtained when we specify t-fairness as *LTL* formulas according to strong fairness and weak fairness patterns (we will refer to this as verifying with strong/weak fairness respectively) with the results obtained with our prototype implementation of the algorithm from Section 4.4 in *DTSpin*, which we refer to as built-in t-fairness. Our prime goal here is to compare the performance of the three methods rather than to verify the protocols.

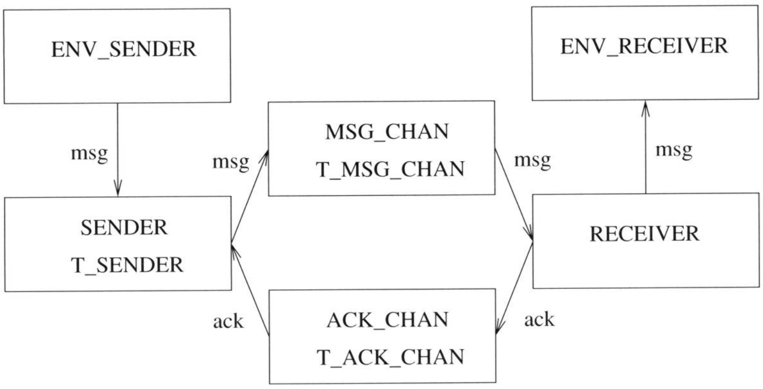

Fig. 11. PAR

The strong fairness pattern for a timer t states that if the transition $k_t^+ \rightarrow_{tick} (k_t - 1)$ is infinitely often enabled, then either this transition is infinitely often taken, or the timer is infinitely often set to a new value. To reformulate this property as a state property, we introduce two boolean variables for each abstract timer. The first variable, x_t, is used to specify the fact that the timer is infinitely often set to a new value, and the second one, y_t, is employed to express enabledness of the $k_t^+ \rightarrow_{tick} (k_t - 1)$ transition. The model of the system is extended so that y_t gets negated every time the *tick*-step is enabled while timer t is in the k^+-state; x_t is negated every time t is set to a new value.

To check whether transition $k_t^+ \rightarrow_{tick} (k_t - 1)$ is taken infinitely often, we also should introduce a new variable. However, we can avoid this since transition $k_t^+ \rightarrow_{tick} (k_t - 1)$ changes the value of timer t to a value different from k^+. Instead of checking whether transition $k_t^+ \rightarrow_{tick} (k_t - 1)$ is taken infinitely often, we check whether timer t takes values different from k^+ infinitely often. So, the strong pattern of t-fairness is expressed by the *LTL* formula:

$$\bigwedge_{t \in TVar} \Big((\Box\Diamond y_t \wedge \Box\Diamond\neg y_t) \rightarrow (\Box\Diamond(t \neq k_t^+) \vee (\Box\Diamond x_t \wedge \Box\Diamond\neg x_t)) \Big)$$

The weak fairness pattern requires that if eventually the value of timer t stays k_t^+, then timer t is infinitely often set to a new value. To specify this pattern as a state formula, we introduce a boolean variable x_t for each abstract timer and extend the model so that the variable is negated each time timer t is set. The formula for the weak fairness pattern looks then as follows:

$$\bigwedge_{t \in TVar} \Big(\Diamond\Box(t = k_t^+) \rightarrow (\Box\Diamond x_t \wedge \Box\Diamond\neg x_t) \Big).$$

Experiments with PAR

PAR [155] is a classical example of a communication protocol where time issues are essential for the correct functionality of the protocol. PAR involves a sender, a receiver, a message channel and an acknowledgment channel (Fig. 11). The sender receives a frame from the upper layer, sends it to the receiver via the message channel and waits for a positive acknowledgment from the receiver via the acknowledgment channel. When the receiver delivers the message to the upper layer, it sends the acknowledgment to the sender. After the positive acknowledgment is received, the sender becomes ready to send the next message. The channels delay the delivery of messages. Moreover, they can lose or corrupt messages. Therefore, the sender handles lost frames by timing out. If the sender times out, it re-sends the message.

The following is an example of a possible erroneous scenario. The sender times out while the acknowledgment is still on the way. The sender sends a duplicate, then receives the acknowledgment and believes that this is the acknowledgment for the duplicate. The sender sends the next frame, which gets lost. The sender, however, receives the acknowledgment for the duplicate, which it believes to be the acknowledgment for the last frame. Thus, the sender does not retransmit the lost message and the protocol fails. To avoid this erroneous behaviour, the timeout interval must be long enough to prevent a premature timeout, which means that the timeout interval should be larger than the sum of delays on the message channel, acknowledgment channel and receiver, i.e. T_SENDER > T_MSG_CHAN + T_ACK_CHAN.

We specified PAR in DTPromela using concrete timers to represent delays on the channels and the sender timeout. Our goal was to check that if the channels do not lose messages continuously, no message reordering occurs and no message gets lost, for any timeout of the sender that is greater than the sum of the (given) delays on the channels. The system is open, i.e. both the sender and the receiver communicate with upper layers, hence we have closed the system by two environment processes: one provides frames for the sender, another receives frames delivered by the receiver.

To prove the property for an arbitrary message sequence we used a well-known canonical abstraction [75, 171]. The data domain is abstracted to $\{a, b, x\}$ where a and b represent two data elements that we differentiate and x represents the rest. An environment for a sender that sends frames with any data

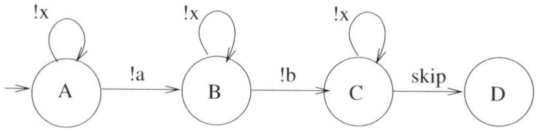

Fig. 12. Environment

Table 7. PAR

pattern	states	transitions	memory(Mb)	time
strong fairness	825761	5.10962e+06	52.286	0:21.00
weak fairness	227569	1.49527e+06	15.320	0:05.98
built-in t-fairness	100275	390012	6.693	0:01.56

chaotically is abstracted into the environment whose behaviour is illustrated by an automaton on Fig. 12. The environment for the receiver behaves analogously but it receives messages instead of sending them. Then we abstracted the sender's timer to check the property for *all* values greater than the sum of the channels' delays.

Without t-fairness, the property gets violated, since there exists a trace where the abstract timer of the sender never expires, staying in the loop $k_t^+ \rightarrow_{tick} k_t^+$ (we obtained a t-unfair trace as counterexample). Under the t-fairness condition, we proved that the property holds. Table 7 contains information on the time and memory consumption for the verification with *DTSpin* of the property formulated with the strong and weak fairness patterns and for the verifier with built-in t-fairness.

Fischer's mutual exclusion protocol

Our second test example is Fischer's mutual exclusion protocol. The protocol uses time constraints and a shared variable to ensure mutual exclusion in a system that consists of N processes running in parallel and competing for a critical section. We assume that each process has a unique *id* from 1 to N. The initial value of the shared variable x is 0. When a process observes that x is 0, it waits for *at most* δ_1 time units and then writes its *id* to x. After that, it waits for *at least* δ_2 time units, and if x still equals the process *id*, the process enters the critical section. The process stays in the critical section for some time and then leaves it.

We have specified Fischer's mutual exclusion protocol in DTPromela using concrete timers to represent delays not larger than δ_1 and abstract timers to represent delays which are at least δ_2. As known, mutual exclusion is ensured provided that $\delta_1 < \delta_2$. We have checked the property that if there comes a request of access to the critical section, one of the processes will get it. Using timer abstraction, we checked that the property is satisfied *for all* delays δ_2 greater than δ_1. The main goal of the experiment was to check whether t-fairness approach works well if we have more than one timer abstracted.

Table 8 contains results for strong, weak and built-in t-fairness for the case of two, three and four processes. Note that the number of abstracted timers in this example is equal to the number of processes. In case of four abstract timers, the *pan* verifier for the property with t-fairness which was expressed as an *LTL* formula according to the strong fairness pattern was not able to generate the state space. The *pan* for the property with t-fairness expressed as an *LTL* formula according to the weak fairness pattern has generated the state space much larger than the state space generated by the *pan* verifier for the same property with built-in t-fairness. The experiments were done on AMD Athlon(TM) XP 2400+ with 1Gb of memory. In all experiments, the verification with built-in t-fairness took significantly less time and memory than the verification with strong and weak fairness patterns expressed as *LTL* formulas. The prototype implementation PAN2TFPAN and the models for PAR and Fischer's mutual exclusion protocol are available at `www.cwi.nl/~ustin/tfair.html`.

4.7 Conclusion

In this chapter, we considered a timer abstraction that introduces a cyclic behaviour on abstract timers that is not present at the concrete level. This could lead to spurious counterexamples for liveness properties. We showed how one can eliminate those by imposing a strong fairness constraint on the traces of the abstract model. Using the fact that the loop on the abstract timer is a self-loop for this abstract timer (though there is possibly no self-loop on the corresponding LTS), we transformed the strong fairness constraint into a constraint which has a weak fairness pattern, and embedded it into the verification algorithm. Our experiments with the prototype implementation of the algorithm were encouraging. We conjecture that the ideas in this chapter can also be used for other data abstractions that introduce self-loops on the abstracted data.

Table 8. Fischer's mutual exclusion

fairness	num. of proc.	states	transitions	memory(Mb)	time
strong	2	41384	171586	4.363	0:00.46
weak	2	4705	13053	2.724	0:00.08
built-in	2	1236	4181	1.573	0:00.01
strong	3	3.28599e+06	2.01406e+07	190.539	1:01.79
weak	3	115874	362068	8.561	0:01.22
built-in	3	21592	110332	2.700	0:00.26
strong	4	out of memory			
weak	4	2.60665e+06	9.2549e+06	151.729	0:38.34
built-in	4	346903	2.45733e+06	20.927	0:05.69

5

Closing and Flow Analysis for Model Checking Reactive Systems

STANDARD MODEL CHECKERS CANNOT HANDLE OPEN SYSTEMS DIRECTLY AND CLOSING IS COMMONLY DONE BY ADDING AN ENVIRONMENT PROCESS, WHICH IN THE SIMPLEST CASE BEHAVES *chaotically*. HOWEVER, FOR MODEL CHECKING, THE WAY OF CLOSING SHOULD BE WELL-CONSIDERED TO ALLEVIATE THE STATE EXPLOSION PROBLEM.

IN THIS CHAPTER WE PROPOSE AN AUTOMATIC TRANSFORMATION YIELDING A CLOSED SYSTEM. BY *embedding* THE OUTSIDE CHAOS INTO THE SYSTEM, WE AVOID THE STATE-SPACE PENALTY CAUSED BY ASYNCHRONOUS COMMUNICATION WITH THE ENVIRONMENT. TO CAPTURE THE CHAOTIC TIMING BEHAVIOUR OF THE ENVIRONMENT, WE INTRODUCE A NON-STANDARD 3-VALUED TIMER ABSTRACTION. THE TRANSFORMATION IS BASED ON *data-flow analysis* THAT DETECTS INSTANCES OF CHAOTIC VARIABLES AND TIMERS.

The chapter is based on [102–104].

5.1 Introduction

Despite all algorithmic advances in model checking techniques and progress in raw computing power, the state explosion problem limits the applicability of model-checking [41, 139, 38] and thus *partial-order reduction* [74, 163], *decomposition* and *abstraction* [122, 41, 50] are indispensable when confronted with checking large designs. Following a compositional approach and after singling out a subcomponent to check in isolation, the next step is often to *close* the subcomponent with an environment since most model checkers (e.g. *Spin* [93]) cannot handle open systems.

Closing is commonly done by adding an environment process that, in order to be able to infer properties for the concrete system, must exhibit at least all the behaviour of the real environment. The simplest safe abstraction of the environment thus behaves *chaotically*, i.e. it sends and receives all possible messages in an arbitrary order. When done manually, this closing, as simple as it is, is tiresome and error-prone for large systems, already due to the sheer amount of signals. Moreover, for model checking, the way of closing should be well-considered to counter the state explosion problem. This is especially true in the context of model checking SDL-specifications (*Specification and Description Language*) [140] with its *asynchronous* message-passing communication model. Sending arbitrary message streams to the unbounded input queues will immediately lead to an infinite state space, unless some assumptions restricting the environment behaviour are incorporated in the closing process. Even so, external chaos results in a combinatorial explosion caused by all combinations of messages in the input queues. This way of closing is even more wasteful, since most of the messages are dropped by the receiver due to the discard-feature of SDL-92.

Another problem the closing must address is that the *data* carried with the messages coming from the environment are usually drawn from some infinite data domain. Since furthermore we are dealing with the discrete-time semantics [94, 25] of SDL, special care must be taken to ensure that the chaos also shows more behaviour wrt. *timing* issues such as timeouts and time progress.

In [151] a program transformation based on static analysis which takes the most abstract, i.e., chaotic environment, and "embeds" it into the component was formalized. Embedding the external chaos eliminates the need to explore the combinatorial state space of the external queues. Part of the approach is the abstraction of environmental *data*, where, assuming a chaotic environment, a single abstract value is used. Interested in a fully-automatic approach, [151] stressed efficiency over precision of abstraction, and used a *static* data-flow analysis to mark all instances of variables *potentially* influenced from outside as chaotic, and to transform the program according to this reckoning. The transformation gets rid of all the data potentially influenced by the environment. A 3-valued timer abstraction is proposed to capture the chaotic timing behaviour.

We improve on this abstraction and generalize the approach in the following way. We combine *may*-analysis (that is reminiscent to the one presented in [103]) marking all the variables potentially influenced by chaotic environment with *must*-analysis that marks data definitely influenced from outside. The combination of *may* and *must* analysis allows to differentiate data definitely influenced from outside, and data definitely *not* influenced from outside, i.e., reliable data; then the rest forms a "don't know" intermediate value for instances at those process locations where both chaotic and non-chaotic values can occur, depending on the system run leading to this instance.

We propose a transformation based on the results of the combined analysis. It gets rid of all the data that are *definitely* influenced by the environment and yields a closed system $S^\sharp$ that treats the remaining data *dynamically*, which gives a more precise approximation and hence less false negatives in the verification. The transformation yields a *closed* system $S^\sharp$ which shows more behaviour in terms of traces than the original one. For formulas of next-free *LTL* [137, 120], we thus get the desired property preservation: if $S^\sharp \models \varphi$ then $S \models \varphi$.

Typical practical applications we are interested in are protocol specifications in SDL [140] and PROMELA [93]. More concretely, the developed methods for closing open asynchronous systems are used to automate the model checking of translations of SDL-specifications into DTPROMELA, the input language of the discrete-time *Spin*, model checker *DTSpin*. The approach is implemented as a tool which automatically closes DTPROMELA translations of SDL-specifications by embedding the timed chaotic environment into the system. To corroborate the usefulness of our approach, we compare the state space of models closed by embedding chaos with the state space of the same models closed with chaos as external environment process on some simple models and on a case study from a wireless ATM medium-access protocol.

Related work

Closing open (sub)systems is common for software *testing*. In this field, a work close to ours in spirit and techniques is the one of [44]. It describes a dataflow algorithm to close program fragments given in the C-language with the most general environment and at the same time eliminating the external interface. The algorithm is incorporated into the *VeriSoft* tool. Similar to the work presented here, they assume an asynchronous communication model, but do not consider *timed* systems and their abstraction. Similarly, [58] consider partial (i.e., open) systems which are transformed into closed ones. To enhance the precision of the abstraction, their approach allows to close the system by an external environment more specific than the most general, chaotic one, where the closing environment can be built to conform to given assumptions, which they call filtering [59]. As in our work, they use *LTL* as temporal logic and Spin as model checker, but the environment is modelled separately and is not embedded into the system.

A more fundamental approach to model checking open systems, also called reactive modules [4], is known as *module* checking [116][115]. Instead of transforming the system into a closed one, the underlying computational model is generalized to distinguish between transitions under control of the module and those driven by the environment. MOCHA [6] is a model checker for reactive modules, which uses alternating-time temporal logic [5] as specification language.

Slicing, a well-known program analysis technique, resembles the analysis described in this paper, in that it is a data-flow analysis computing — in forward or backward direction — parts of the program that may depend on certain points of interest (cf. for a survey [159]). The analysis of Section 5.3 computes in a forward manner the cone of influence of all points of the system influenced from the outside. The usefulness of slicing for model checking is explored in [128], where slicing is used to speed up model checking and simulation for programs in Promela, Spin's input language. However, the program transformation in [128] is not intended to preserve program properties in general. Likewise in the context of *LTL* model checking, [57] use slicing to cut away irrelevant program fragments but the transformation yields a safe, property-preserving abstraction and potentially a smaller state space.

The chapter is organized as follows: Section 5.2 gives a semantics that is a simplification of the semantics given in Section 3.3 and argues the correctness of the simplification. Section 5.3 describes *data-flow analysis*. In Section 5.4, we describe our approach to closing and present some preservation results. Section 5.5 describes an implementation of the approach, provides some examples motivating the necessity of embedding and presents a case study from a wireless ATM medium-access protocol.

5.2 Semantics

The transformation described in Section 3.3 substitutes the SDL concept of timeouts as a special kind of signals that are kept in input queues of the processes by the concept of timeouts as guards. After the transformation, no timeout signal is placed into the input queue of the process, so the input queue becomes just an extra buffer between a channel and a process. In this chapter, we use the semantics that is a simplification of one in Section 3.3. Here, processes take messages directly from input channels *without putting* them first *into* an *input queue*. In this section, we show that the systems with input queues considered in Section 3.3 and the systems without input queues used in this chapter are path-equivalent up to stuttering (see Def. 2.28).

The semantics of a process specification is the LTS defined by the rules of Table 9. The semantics coincides with the semantics given by the rules of Tables 5 and 6 except for rules RECEIVE, INPUT and DISCARD of Table 5 and the definition of a process state. Here, a state of a process is given by a location and a valuation of process and timer variables. A valuation is denoted as η.

Definition 5.1. STATE OF A PROCESS
A state σ of a process P is a pair (l, η), where l is a location and η is a valuation of variables. Σ denotes the set of states of the process.

Definition 5.2. PROCESS P
A process P is an LTS $S = (\Sigma, Lab_P, \rightarrow_\lambda, \sigma_0, In, Out)$ *where $\sigma_0 = (l_0, \eta_0)$ is an initial state, In is a set of input channel names, Out is a set of output channels names and $\rightarrow \subseteq \Sigma \times Lab \times \Sigma$ is a labelled transition relation derived by applying the rules of Table 9 to some process specification $Spec_P$.*

Rules OUTPUT, ASSIGN, SET, RESET, TICK$_P$, TIMEOUT, TDISCARD of Table 9 coincide with the same rules of Table 5. Rules RECEIVE, INPUT and DISCARD of Table 5 are substituted by rules INPUT and DISCARD of Table 9. The semantics for channels and n-ary composition that allows to put n entities into communication is given by the rules IN and OUT of Table 2 and the rules of Table 6.

Further, we consider a system specification *Spec* that consists of n components (channels and processes), LTS S' obtained from *Spec* by applying the rules of Tables 5 and 6, and LTS S obtained from *Spec* by applying rules of Tables 9 and 6. Suppose $\gamma = (\sigma_1, \ldots, \sigma_i, \ldots, \sigma_n)$ is a configuration of S consisting of n entities (processes and channels), and let $\gamma' = (\sigma'_1 \ldots, \sigma'_i, \ldots, \sigma'_n)$ be a configuration of S'. Moreover, σ_i denotes a state of the ith-entity in the original system and σ'_i denotes a state of this entity in the transformed system. In S', each process has an input queue, and in S there are no input queues.

The following lemma expresses that the blocked predicate is compositional in the sense that the n-ary composition of entities (processes and channels) is blocked iff each entity is blocked (rule TICK of Table 6).

Lemma 5.1.
For a state $\gamma = (\sigma_1, \ldots, \sigma_n)$ of a system S, blocked(γ) iff blocked(σ_i) for all σ_i.

Proof. If γ is not blocked, it can perform a τ-step or an output-step. The output-step must originate from a process, which thus is not blocked. The τ-step is either caused by a single process or by a synchronizing action of a sender and a receiver; in both cases at least one process is not blocked.

For the reverse direction, a τ-step of a single process being thus not blocked, entails that γ is not blocked. An output-step of a single process causes γ either to do the same output-step or, in case of internal communication, to do a τ-step. In both cases, γ is not blocked. □

Lemma 5.2.
Let S be a system and $\gamma \in \Gamma$ one of its states.

1. *If $\gamma \rightarrow_{tick} \hat{\gamma}$, then $[\![t]\!]_\gamma \neq on(0)$, for all timers t.*
2. *If $\gamma \rightarrow_{tick} \hat{\gamma}$, then for all channel states (c, q), $q = \epsilon$.*

$$\frac{l \longrightarrow_{c?s(x)} \hat{l} \in \mathit{Edg}}{(l,\eta) \to_{c_i?s(pid,v)} (\hat{l}, \eta_{[x \mapsto v]})}\ \textsc{Input}$$

$$\frac{s' \notin \{s \mid l \longrightarrow_{c?s(x)} \hat{l} \in \mathit{Edg}\} \qquad l \in \mathit{Loc}_i}{(l,\eta) \to_{c_i?s'(pid,v)} (l, \eta)}\ \textsc{Discard}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, c!(s,e)} \hat{l} \in \mathit{Edg} \qquad \llbracket g \rrbracket_\eta = \mathit{true} \qquad \llbracket e \rrbracket_\eta = v}{(l,\eta) \to_{c_o!s(pid,v)} (\hat{l}, \eta)}\ \textsc{Output}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in \mathit{Edg} \qquad \llbracket g \rrbracket_\eta = \mathit{true} \qquad \llbracket e \rrbracket_\eta = v}{(l,\eta) \to_{\tau} (\hat{l}, \eta_{[x \mapsto v]})}\ \textsc{Assign}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, set\ t:=e} \hat{l} \in \mathit{Edg} \qquad \llbracket g \rrbracket_\eta = \mathit{true} \qquad \llbracket e \rrbracket_\eta = v}{(l,\eta) \to_{\tau} (\hat{l}, \eta_{[t \mapsto on(v)]})}\ \textsc{Set}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, reset\ t} \hat{l} \in \mathit{Edg} \qquad \llbracket g \rrbracket_\eta = \mathit{true}}{(l,\eta) \to_{\tau} (\hat{l}, \eta_{[t \mapsto \mathit{off}]})}\ \textsc{Reset} \qquad \frac{\mathit{blocked}(l,\eta)}{(l,\eta) \to_{tick} (l, \eta_{dec})}\ \textsc{Tick}_P$$

$$\frac{l \longrightarrow_{g_t \,\triangleright\, reset\ t} \hat{l} \in \mathit{Edg} \qquad \llbracket t \rrbracket_\eta = on(0)}{(l,\eta) \to_{\tau} (\hat{l}, \eta_{[t \mapsto \mathit{off}]})}\ \textsc{Timeout}$$

$$\frac{t' \notin \{t \mid l \to_{g_t \,\triangleright\, reset\ t} \hat{l} \in \mathit{Edg}'\} \qquad \llbracket t' \rrbracket_\eta = on(0) \qquad l \in \mathit{Loc}_i}{(l,\eta) \to_{\tau} (l, \eta_{[t' \mapsto \mathit{off}]})}\ \textsc{TDiscard}$$

Table 9. Step semantics for process specification Spec_P

Proof. For part (1), if $[\![t]\!]_\eta = on(0)$ for a timer t in a process P, then a τ-step is allowed due to either rule TIMEOUT or rule TDISCARD of Table 9. Hence, the system is not *blocked* and therefore cannot do a *tick*-step.

Part (2) follows from the fact that a channel can perform a *tick*-step only when it is empty (rule TICK$_c$ of Table 6). □

We want to be sure that the absence of the input queues does not influence verification results, i.e. $S' \models \phi$ iff $S \models \phi$ for all formulas ϕ of *LTL–X*. Namely, we show that there is a branching simulation relation (see Def. 2.19) relating system S' to system S. In the other direction, there is a weak trace inclusion relation (see Def. 2.17) between S and S'. This implies that the system with input queues and the system without them are weakly trace equivalent (see Def. 2.18). We also show that they are path-equivalent up to stuttering (see Def. 2.28).

Let P' be the LTS obtained by applying the rules of Table 5 to some process specification $Spec_P$, and let P be the LTS obtained by applying the rules of Table 9 to $Spec_P$. The absence/presence of input queues influences only communication between a process and its input channels. Therefore, we first consider LTSs E and E' that are the compositions of P and P', respectively, with the LTSs of its input channels according to the rules of Table 6.

For E' and E, we define a relation $\mathcal{Q} \subseteq \Gamma' \times \Gamma$ on configurations of E' and E respectively. Here the typical element of Γ' is a configuration $\gamma' = ((l, \eta, q), (c_1, q_1), \ldots, (c_n, q_n))$ and the typical element of Γ is $\gamma = ((l, \eta), (c_1, \tilde{q}_1), \ldots, (c_n, \tilde{q}_n))$, where $c_1, \ldots, c_n$ are input channel names of P and P', q_i and $\tilde{q}_i$ are queues modelling the channels and q is the input queue of process P'. We assume that all the messages kept in the input queue and in the queues modelling channels are different. The first requirement imposed on configurations is that the valuations of variables should be equal. The second one requires that the contents of the channel queues of E should allow process P to consume messages in the same order in which they can be consumed by P' from its input queue. $\pi(q)_{Msg-Msg_{\tilde{q}_i}}$ denotes the projection of q on all the messages that are not an element of $\tilde{q}_i$.

Definition 5.3. INPUT QUEUE APPROXIMATION RELATION
We call $\mathcal{Q} \subseteq \Gamma' \times \Gamma$ an input queue approximation *relation iff for all $\gamma' \mathcal{Q} \gamma$: $\gamma' = ((l, \eta, q), (c_1, q_1), \ldots, (c_n, q_n))$ and $\gamma = ((l, \eta), (c_1, \tilde{q}_1), \ldots, (c_n, \tilde{q}_n))$, and the following conditions hold:*

- *for all $i = 1..n$, $\pi(q)_{Msg-Msg_{\tilde{q}_i}} :: q_i = \tilde{q}_i$;*
- *for all messages msg in q, msg is an element of $\tilde{q}_i$ for some $i = 1..n$.*

Further, we prove that there is a $\mathcal{Q}$ that is an *input queue approximation* relation and a branching simulation on E' and E (Def. 2.19).

Lemma 5.3.
Let $Spec_P$ be a process specification and P' be the LTS obtained by applying the rules of Table 5 to $Spec_P$. Let P be the LTS obtained by applying the rules of

Table 9 to the same process specification. Let E be the composition of P with the LTSs of its input channels, and let E' be the composition of P' with the LTSs of its input channels according to the rules of Table 6. Then there exists $\mathcal{Q} \subseteq \Gamma' \times \Gamma$ that is an input queue approximation relation and a branching simulation between E' and E.

Proof. Let γ_0' be the initial configuration of E' and γ_0 be the initial configuration of E. Since all queues (modelling channels and the input queue of P') are initially empty, $\gamma_0' \mathcal{Q} \gamma_0$. Assume that $\gamma' \mathcal{Q} \gamma$ holds for some states γ' and γ of E' and E respectively. To show that $E' \preceq_{br} E$, we proceed with a case analysis on the rules in Table 5 and rule COMM of Table 6. Note that here we consider rule COMM only for the case of receiving a message by a process that involves the use of input queues.

Case: COMM
Assume that $\gamma' = ((l, \eta, q), \ldots, (c_i, msg :: q_i), \ldots)$ and that E' makes step $((l, \eta, q), \ldots, (c_i, msg :: q_i), \ldots) \to_\tau ((l, \eta, q :: msg), \ldots, (c_i, q_i), \ldots)$. Since $\gamma' \mathcal{Q} \gamma$, $\gamma = ((l, \eta), \ldots, (c_i, msg :: \tilde{q}_i), \ldots)$.

After the τ-step of E', where $\hat{\gamma}' = ((l,\ \eta,\ q :: msg),\ \ldots,\ (c_i, q_i),\ \ldots)$. Moreover, $\hat{\gamma}' \mathcal{Q}\ \gamma$ still holds. Condition 2 of Def. 2.19 is satisfied.

Case: INPUT
Assume that $\gamma' = ((l, \eta, s(pid, v) :: q), \ldots, (c_n, q_n))$ and that E' makes step $((l, \eta, s(pid, v) :: q), \ldots, (c_n, q_n)) \to_\tau ((\hat{l}, \eta[x \mapsto v], q), \ldots, (c_n, q_n))$.

By rule INPUT of Table 5, $l \longrightarrow_{c_j ? s(x)} \hat{l} \in Edg$. Since $\gamma' \mathcal{Q} \gamma$, the mimicking step $((l, \eta),\ \ldots,\ (c_j,\ s(pid, v) :: \tilde{q}_j),\ \ldots) \to_\tau ((\hat{l},\ \eta[x \mapsto v]),\ \ldots,\ (c_j, \tilde{q}_j),\ \ldots)$ is possible in E by rule INPUT of Table 9 and rule COMM of Table 6. For $\hat{\gamma}' = ((\hat{l},\ \eta[x \mapsto v], q),\ \ldots,\ (c_n, q_n))$ and $\hat{\gamma} = ((\hat{l},\ \eta[x \mapsto v]),\ \ldots,\ (c_j,\ \tilde{q}_j),\ \ldots)$, $\hat{\gamma}' \mathcal{Q}\ \hat{\gamma}$. Moreover, condition 1 of Def. 2.19 is satisfied.

Case: DISCARD
Analogous to the case INPUT.

Rules OUTPUT, ASSIGN, SET, RESET, TICK$_P$, TIMEOUT, TDISCARD of Table 9 coincide with the rules of Table 5. The channels have the same semantics in both cases. It is straightforward to show that condition 1 of Def. 2.19 is satisfied for OUTPUT, ASSIGN, SET, RESET, TICK$_P$, TIMEOUT and TDISCARD cases, and so $E' \preceq_{br} E$. □

Proposition 5.1. BRANCHING SIMULATION
Let Spec be a system specification, S' be the LTS obtained by applying the rules of Tables 5 and 6 to Spec, and S be the LTS obtained by applying the rules of Tables 9 and 6 to Spec. Then there exists $\mathcal{Q} \subseteq \Gamma' \times \Gamma$ that is an input queue approximation relation and a branching simulation between S' and S.

Proof. For each channel, there is only one process reading from the channel (see Def. 3.2). Relation $\mathcal{Q}$ as given in Def. 5.3 can be easily lifted to the definition of $\mathcal{Q}$ on configurations of S' and S. S' and S are the LTSs obtained using the same rules for n-ary composition that allows to put n processes into communication.

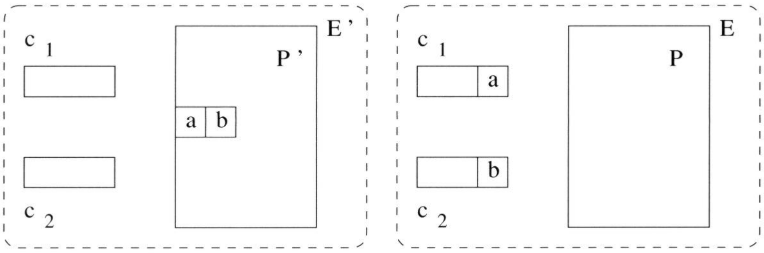

Fig. 13. Input queue

By Lemma 5.3 and by a case analysis on rules of Table 6, it is straightforward to show that $S' \preceq_{br} S$. □

Intuitively, we would like $\mathcal{Q}^{-1}$ to be a branching simulation. An attempt to establish a simulation in the reverse direction fails. In Fig. 13, the configurations $((l, \eta, a :: b), (c_1, \epsilon), (c_2, \epsilon))$ and $((l, \eta), (c_1, a), (c_2, b))$ of system E' and E are related by $\mathcal{Q}$. In E', first message a can be consumed and then message b. E still has a choice, i.e. the messages can be consumed in any order, hence, $\mathcal{Q}$ is *not* a branching bisimulation (Def. 2.20). However, it is possible to show that for each trace of S there exists a trace of S' having the same stuttering-free projection (cf. Def. 2.26). Further, we show that S' and S are path equivalent up to stuttering (Def. 2.28).

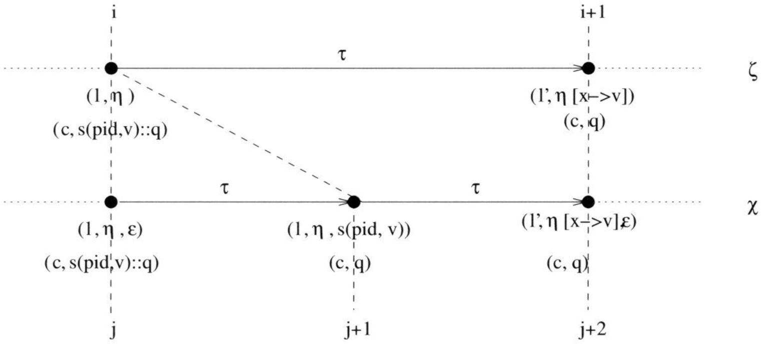

Fig. 14. Mimicking a reception τ-step

Definition 5.4.
Let ζ and χ be traces of S and S', respectively. We write $\zeta \equiv_{\mathcal{V}} \chi$ iff there exists $\mathcal{V} \subseteq \mathbb{N} \times \mathbb{N}$ such that

1. $(i, j) \in \mathcal{V}$ *implies* $\chi_\gamma(j)\mathcal{Q}\zeta_\gamma(i)$.
2. ζ *and* χ *can be partitioned as* $\zeta^{I_1}\zeta^{I_2}\ldots$ *and* $\chi^{J_1}\chi^{J_2}\ldots$, *respectively, so that for all* $k > 0$, $I_k = \{i, i+1\}$, $J_k = \{j, \ldots, j+m\}$, $1 \leq m \leq 2$ *and the following conditions are satisfied:*
   - $(i, j), (i+1, j+m) \in \mathcal{V}$, $\zeta_\lambda(i+1) = \chi_\lambda(j+m)$, *and all input queues are empty at* $\chi_\gamma(j)$ *and at* $\chi_\gamma(j+m)$;
   - $\chi_\lambda(j+1) = \tau$ *and* $(i, j+1) \in \mathcal{V}$ *if* $m = 2$.

We write $S \preceq_\mathcal{V} S'$ *iff for every trace* ζ *of* S *there exists a trace* χ *in* S' *such that* $\zeta \equiv_\mathcal{V} \chi$ *for some* $\mathcal{V} \subseteq \mathbb{N} \times \mathbb{N}$.

Further, we show that for each trace ζ of E there is trace χ of E' such that χ has the same stuttering-free projection as ζ. Roughly speaking, each step of ζ is mimicked by the same step of χ except τ-steps that take a message from a queue modelling an input channel and consume it. For an example on Fig. 14, assume that $\zeta^{(i)} \equiv_\mathcal{V} \chi^{(j)}$ for some $\mathcal{V} \subseteq \mathbb{N} \times \mathbb{N}$ and that the $(i+1)$th-step of ζ is the τ-step that takes message $s(pid, v)$ from the queue modelling channel c and consumes the message. To be able to mimic such a step in E', we postpone moving messages from queues modelling channels into the input queue of the process as long as possible.

Since $\zeta^{(i)} \equiv_\mathcal{V} \chi^{(j)}$, $\chi_\gamma(j)\mathcal{Q}\zeta_\gamma(i)$ by condition 1 of Def. 5.4. Therefore, we take a τ-step that removes signal $s(pid, v)$ from the queue modelling channel c and adds it to the empty input queue of process P' (cf. rule COMM of Table 6). For the configuration $\chi_\gamma(j+1)$ reached by the τ-step, $\chi_\gamma(j+1)\mathcal{Q}\zeta_\gamma(i)$ holds, and we add $(i, j+1)$ to $\mathcal{V}$.

According to the rule INPUT of Table 5, we can make a τ-step that consumes the message $s(pid, v)$ from the input queue of process P' and leads to configuration $\chi_\gamma(j+2)$ such that $\chi_\gamma(j+2)\mathcal{Q}\zeta_\gamma(i+1)$. We add $(i+1, j+2)$ to $\mathcal{V}$, and thus we obtain $\zeta^{(i+1)} \equiv_\mathcal{V} \chi^{(j+2)}$ for the extended $\mathcal{V}$.

Lemma 5.4.
Let ζ *and* χ *be traces of* S *and* S' *respectively. Let* $\zeta \equiv_\mathcal{V} \chi$ *for some* $\mathcal{V} \subseteq \mathbb{N} \times \mathbb{N}$. *Then* $\zeta \equiv_{wtr} \chi$.

Proof. Traces χ and ζ that satisfy condition 2 of Def. 5.4 also satisfy conditions of Def. 2.16. □

Let π_ζ and π_χ be paths corresponding to traces ζ and χ respectively, i.e. $\pi_\zeta = \zeta_\gamma(0)\zeta_\gamma(1)\ldots$ and $\pi_\chi = \chi_\gamma(0)\chi_\gamma(1)\ldots$.

Lemma 5.5.
Let ζ *and* χ *be traces of* S *and* S' *respectively. Let* $\zeta \equiv_\mathcal{V} \chi$ *for some* $\mathcal{V} \subseteq \mathbb{N} \times \mathbb{N}$. *Let* $\mathcal{L}\colon \Gamma \to 2^\mathcal{P}$ *and* $L'\colon \Gamma' \to 2^\mathcal{P}$ *be interpretation functions,* $\mathcal{P}$ *be the set of atomic propositions, and* Γ *and* Γ' *be sets of configurations of* S *and* S' *respectively. Then* $\pi_\zeta \equiv_{st} \pi_\chi$.

Proof. Both S and S' are obtained from some specification *Spec*, so they have the same set of variables. Since $\zeta \equiv_{\mathcal{V}} \chi$ for some $\mathcal{V} \subseteq \mathbb{N} \times \mathbb{N}$, $\mathcal{L}(\zeta_\gamma(i)) = \mathcal{L}'(\chi_\gamma(j))$ for all $(i, j) \in \mathcal{V}$ by condition 1 of Def. 5.4.

By condition 2 of Def. 3.14, each step of trace ζ is mimicked either by the same step of trace χ or by by the same step of χ preceded by one τ-step that does not change the valuation of variables. Further, it is straightforward to show that $\mathcal{L}(Pr(\pi_\zeta)(k)) = \mathcal{L}'(Pr(\pi_\chi)(k))$ for all $k \geq 0$ (cf. Def. 2.27). □

Lemma 5.6.
Let $Spec_P$ be a process specification, P' be the LTS obtained by applying the rules of Table 5 to $Spec_P$, and P be the LTS obtained by applying the rules of Table 9 to $Spec_P$. Let E be the composition of P with the LTSs modelling its input channels and let E' be the composition of P' with the LTSs modelling its input channels according to the rules of Table 6. For each trace ζ of E there is a trace χ of E' such that $\zeta \equiv_{\mathcal{V}} \chi$ for some $\mathcal{V} \subseteq \mathbb{N} \times \mathbb{N}$.

Proof. In Lemma 5.3, we have already demonstrated that $\gamma'_0 \mathcal{Q} \gamma_0$ is valid for the initial configurations of E' and E; thus, $\mathcal{V} = \{(0, 0)\}$ and $\zeta_\gamma(0) = \gamma_0$ initially. Further, we construct a trace χ and a relation $\mathcal{V}$ by induction on the length of the trace ζ. Each step of ζ is mimicked by a step of E', and configurations reached by a step of ζ and the mimicking step are related by $\mathcal{Q}$. Now we proceed with a case analysis on the rules of Table 9 and rule COMM of Table 6. Note that here we consider rule COMM only for the case when a process receives a message from a channel. Assume that $(i, j) \in \mathcal{V}$ and that all input queues are empty in $\chi_\gamma(j)$. By Def. 5.4, traces $\zeta^{(i)}$ and $\chi^{(j)}$ can be partitioned as ${\zeta^{(i)}}^{I_1} \ldots {\zeta^{(i)}}^{I_i} \ldots$ and ${\chi^{(j)}}^{J_1} \ldots {\chi^{(j)}}^{J_i}$.

Case: COMM
Let the $(i + 1)$th-step of ζ be

$$((l, \eta), \ldots, (c_k, s(pid, v) :: \tilde{q}_k), \ldots) \rightarrow_\tau ((\hat{l}, \eta[x \mapsto v]), \ldots, (c_j, \tilde{q}_j), \ldots)$$

By rule COMM of Table 6 and rule INPUT of Table 9, $l \longrightarrow_{c_j?s(x)} \hat{l} \in Edg$.

By Def. 5.4, $(i, j) \in \mathcal{V}$ implies that $\chi_\gamma(j) \mathcal{Q} \zeta_\gamma(i)$ and the input queue of P' is empty at $\chi_\gamma(j)$. To mimic the τ-step of ζ, we extend $\chi^{(j)}$ by two steps. By rule COMM of Table 6, we first add the τ-step:

$$((l, \eta, \epsilon), \ldots, (c_j, s(pid, v) :: q_j), \ldots) \rightarrow_\tau ((l, \eta, s(pid, v)), \ldots, (c_j, q_j), \ldots)$$

that removes message $s(pid, v)$ from the head of the queue modelling channel c_j and adds it to the input queue of process P'. We add $(i, j + 1)$ to $\mathcal{V}$ and define $\chi_\gamma(j+1) = ((l, \eta, s(pid, v)), \ldots, (c_i, q_i), \ldots)$, $\chi_\lambda(j+1) = \tau$. For this step, $\chi_\gamma(j + 1) \mathcal{Q} \zeta_\gamma(i)$.

By rule INPUT of Table 5, we add the τ-step:

$$((l, \eta, s(pid, v)), \ldots, (c_n, q_n)) \rightarrow_\tau ((\hat{l}, \eta[x \mapsto v], \epsilon), \ldots, (c_n, q_n))$$

consuming $s(pid, v)$ from the input queue of P'. We add $(i+1, j+2)$ to $\mathcal{V}$ and define $\chi_\gamma(j+2) = ((\hat{l}, \eta[x \mapsto v], \epsilon), \ldots, (c_n, q_n))$, $\chi_\lambda(j+2) = \tau$. Moreover, $\chi_\gamma(j+2)\mathcal{Q}\zeta_\gamma(i+1)$ and the input queue of P' is empty at $\chi_\gamma(j+2)$.

The $(i+1)$th-step of $\zeta^{(i+1)}$ forms partition I_{i+1}. The $(j+1)$th-step together with $(j+2)$th-step forms partition J_{i+1} of $\chi^{(j+2)}$. All conditions of Def. 5.4 are satisfied, and $\zeta^{(i+1)} \equiv_{\mathcal{V}} \chi^{(j+2)}$.

It is straightforward to show that we can mimic the $(i+1)$th-step of ζ by the $(j+1)$th-step of E' so that $\zeta^{(i+1)} \equiv_{\mathcal{V}} \chi^{(j+1)}$ for cases DISCARD, OUTPUT, ASSIGN, SET, RESET, TICK$_P$, TIMEOUT and TDISCARD of Table 9.

Here we showed that for each finite prefix $\zeta^{(i+1)}$ of trace ζ of E, we can construct a finite prefix $\chi^{(j+m)}$ of trace χ of E' such that $\zeta^{(i+1)} \equiv_{\mathcal{V}} \chi^{(j+m)}$. It means that for each trace ζ of E there is a trace χ of E' such that $\zeta \equiv_{\mathcal{V}} \chi$ for some $\mathcal{V}$. □

Proposition 5.2.
Let Spec be a system specification, S' be the LTS obtained by applying the rules of Tables 5, rules IN *and* OUT *of Table 2 and the rules of Table 6 to Spec, and S be the LTS obtained by applying the rules of Tables 9, rules* IN *and* OUT *of Table 2 and the rules of Table 6 to Spec. For each trace ζ of S there is a trace χ of S' such that $\zeta \equiv_{\mathcal{V}} \chi$ for some $\mathcal{V} \subseteq \mathbb{N} \times \mathbb{N}$.*

Proof. For each channel c there is only one process that takes messages from the channel (see Def. 3.2). Relation $\mathcal{Q}$ as given by Def. 5.3 can be easily lifted to a definition of $\mathcal{Q}$ on configurations of S' and S. S and S' are the LTSs obtained using the same rules of n-ary composition. By Lemma 5.6 and by a case analysis on the rules of Table 6, it is straightforward to show that for each trace ζ of S there is a trace χ of S' such that $\zeta \equiv_{\mathcal{V}} \chi$ for some $\mathcal{V}$. □

Proposition 5.3. WEAK TRACE EQUIVALENCE
Let Spec be a system specification, S' be the LTS obtained by applying the rules of Tables 5, rules IN *and* OUT *of Table 2 and the rules of Table 6 to Spec, and S be the LTS obtained by applying the rules of Tables 9, rules* IN *and* OUT *of Table 2 and the rules of Table 6 to Spec. Then $S \equiv_{wtr} S'$.*

Proof. Case: $S' \preceq_{wtr} S$
Follows from Proposition 5.1.

Case: $S \preceq_{wrt} S'$
Follows from Proposition 5.2 and Lemma 5.4. □

Proposition 5.4. PATH INCLUSION UP TO STUTTERING
Let Spec be a system specification, S' be the LTS obtained by applying the rules of Tables 5, rules IN *and* OUT *of Table 2 and the rules of Table 6 to Spec, and S be the LTS obtained by applying the rules of Tables 9, rules* IN *and* OUT *of Table 2 and the rules of Table 6 to Spec. Further, let $\mathcal{L}$ and $\mathcal{L}'$ be interpretation functions for the set of atomic propositions $\mathcal{P}$. Then $(S, \mathcal{L}) \equiv_{st} (S', \mathcal{L}')$.*

Proof. Both $(S, \mathcal{L})$ and $(S', \mathcal{L}')$ are L^{2}TSs (Def. 2.25). Here we assume that neither system S nor system S' contains deadlocks, because even if the system cannot proceed, time can progress. So all traces of S and S' are infinite.

Case: $(S', \mathcal{L}') \preceq_{st} (S, \mathcal{L})$
Follows from Def. 5.3 and Proposition 5.1.

Case: $(S, \mathcal{L}) \preceq_{st} (S', \mathcal{L}')$
Follows from Proposition 5.2 and Lemma 5.5.

$(S, \mathcal{L}) \preceq_{st} (S', \mathcal{L}')$ and $(S', \mathcal{L}') \preceq_{st} (S, \mathcal{L})$ imply $(S, \mathcal{L}) \equiv_{st} (S', \mathcal{L}')$. □

Theorem 5.1.
For all formulas φ of LTL–X mentioning process variables and timer variables of Spec, $S \models \varphi$ iff $S' \models \varphi$.

Proof. Straightforward from Proposition 5.4 and Theorem 2.2. □

5.3 Marking Chaotically-influenced Variables

Originating from an unknown or underspecified environment, signals from outside can carry *any* value, which often leads to infinite state space. Assuming nothing about the data means one can conceptually abstract values from outside into one abstract *"chaotic"* value, which basically means to ignore these data and focus on the control structure. Data not coming from outside is left untouched, though chaotic data from the environment influence internal data of the system. In this section, we present a straightforward data-flow analysis marking variable and timer instances that may be influenced by the environment, namely we establish for each process- and timer-variable in each location whether

1. the variable is guaranteed to be *not* influenced by the outside, or
2. the variable is guaranteed to be influenced by the outside, or
3. whether its status depends on the actual run.

The analysis is a combination of the ones from [151] and [103].

5.3.1 Data-Flow Analysis

The analysis works on a simple *flow graph* representation of the system; each process is represented by a single flow graph, whose nodes $n \in nodes$ are associated with the process' actions and the flow relation captures the intra-process data dependencies. Since the structure of the language we consider is rather simple, the flow graph can be easily obtained by standard techniques.

We use an abstract representation of the data values, where $\top$ is interpreted as a value chaotically influenced by the environment and $\bot$ stands for a non-chaotic value. We write $\eta^\alpha, \eta^\alpha_1, \ldots$ for abstract valuations, i.e., for typical elements from $Val^\alpha = Var \to \{\top, \bot\}$. The abstract values are ordered $\bot \leq \top$, and the order is lifted pointwise to valuations. With this ordering, the set of valuations forms a complete lattice, where we write $\eta_\bot$ for the least element, given as $\eta_\bot(x) = \bot$ for all $x \in Var$, and we denote the least upper bound of $\eta_1^\alpha, \ldots, \eta_n^\alpha$ by $\bigvee_{i=1}^n \eta_i^\alpha$ (or by $\eta_1^\alpha \vee \eta_2^\alpha$ in the binary case). By slight abuse of notation, we will use the same symbol η^α for the valuation per node, i.e., for functions of type $node \rightarrow Val^\alpha$.

Depending on whether we are interested in an answer to point (1) or point (2) from above, $\top$ is interpreted as a variable *influenced* from outside, and, dually for the second case, $\top$ stands for variables *guaranteed* to be influenced from outside. We present *may* and *must* analysis for the first and the second case respectively.

May Analysis

First we consider *may* analysis that marks variables *potentially* influenced by data from outside. Each node n of the flow graph has associated with it an abstract transfer function $f_n : Val^\alpha \rightarrow Val^\alpha$, describing the change of the abstract valuations depending on the kind of action at the node. The functions are given in Table 10. The equations are mostly straightforward; the only case deserving mention is the one for $c?s(x)$, whose equation captures the inter-process data-flow from a sending to a receiving action. If s is an external signal then variable x is *potentially* influenced from outside. A process within the system can also send a message parameterized by data influenced from outside. That is captured by $\bigvee\{[\![e]\!]_{\eta^\alpha} \mid n' = g \rhd c!s(e)\}$. It allows to mark a variable by $\top$ if at least one process sends signal s with data influenced from outside. Sending a signal $c!s(e)$ does not change the valuation of variables, hence it does not influence abstract valuation as well.

Assignment $g \rhd x := e$ changes the valuation of x depending on the valuation of expression e. The abstract valuation $[\![e]\!]_{\eta^\alpha}$ for an expression e equals $\bot$ iff all variables in e are evaluated to $\bot$, $[\![e]\!]_{\eta^\alpha}$ is $\top$ iff the abstract valuation of at least one of the variables in e is $\top$. Setting a timer is similar to an assignment. Reset and timeout set a timer to the reliable value *off*. It is easy to see that the transfer functions are *monotonic.*

Upon the start of the analysis, the variables' values at each node are assumed to be defined; the initial valuation is the least one: $\eta_{init}^\alpha(n) = \eta_\bot$. This choice rests on the assumption that all local variables of each process are properly initialized. We are interested in the least solution to the data-flow problem given by the following constraint set:

$$\begin{aligned} \eta_{post}^\alpha(n) &\geq f_n(\eta_{pre}^\alpha(n)) \\ \eta_{pre}^\alpha(n) &\geq \bigvee\{\eta_{post}^\alpha(n') \mid (n', n) \text{ in flow relation}\} \end{aligned} \tag{4}$$

For each node n of the flow graph, the data-flow problem is specified by two inequations or constraints. The first one relates the abstract valuation η_{pre}^α before entering the node with the valuation η_{post}^α afterwards via the abstract

$$
\begin{aligned}
f(c?s(x))\eta^\alpha &= \begin{cases} \eta^\alpha{}_{[x \mapsto \top]} & s \in Sig_{ext} \\ \eta^\alpha{}_{[x \mapsto \bigvee\{[\![e]\!]_{\eta^\alpha} \mid n' = g \rhd c!s(e)\}]} & s \notin Sig_{ext} \end{cases} \\
f(g \rhd c!s(e))\eta^\alpha &= \eta^\alpha \\
f(g \rhd x := e)\eta^\alpha &= \eta^\alpha{}_{[x \mapsto [\![e]\!]_{\eta^\alpha}]} \\
f(g \rhd set\ t := e)\eta^\alpha &= \eta^\alpha{}_{[t \mapsto on([\![e]\!]_{\eta^\alpha})]} \\
f(g \rhd reset\ t)\eta^\alpha &= \eta^\alpha{}_{[t \mapsto off]} \\
f(g_t \rhd reset\ t)\eta^\alpha &= \eta^\alpha{}_{[t \mapsto off]}
\end{aligned}
$$

Table 10. May analysis: transfer functions/abstract effect for process P

```
input : the flow graph of the program
output: $\eta^\alpha_{pre}, \eta^\alpha_{post}$;

$\eta^\alpha(n) = \eta^\alpha_{init}(n)$;
$WL = \{n \mid \alpha_n = ?s(x), s \in Sig_{ext}\}$;

repeat
    pick $n \in WL$;
    if   $n = g \rhd c!s(e)$  then
        let  $S' = \{n' \mid n' = c?s(x)$ and $[\![e]\!]_{\eta^\alpha(n)} \not\leq [\![x]\!]_{\eta^\alpha(n')}\}$
        in
            for all $n' \in S'$: $\eta^\alpha(n') := f_{n'}(\eta^\alpha(n'))$;
    let $S = \{n'' \in succ(n) \mid f_n(\eta^\alpha(n)) \not\leq \eta^\alpha(n'')\}$
    in
        for all $n'' \in S$: $\eta^\alpha(n'') := f_n(\eta^\alpha(n))$;
    $WL := WL \backslash \{n\} \cup S \cup S'$;
until $WL = \emptyset$;

$\eta^\alpha_{pre}(n) = \eta^\alpha(n)$;
$\eta^\alpha_{post}(n) = f_n(\eta^\alpha(n))$
```

Fig. 15. *May* analysis: worklist algorithm

effects of Table 10. The *least* fixpoint of the constraint set can be found iteratively in a fairly standard way by a *worklist algorithm* (see e.g., [111, 85, 131]), where the worklist steers the iterative loop until the least fixpoint is reached (cf. Figure 15).

The worklist data-structure *WL* used in the algorithm is a set of elements, more specifically a set of nodes from the flow graph, where we denote by $succ(n)$ the set of successor nodes of n in the flow graph in forward direction. It supports as operation to randomly pick one element from the set (without removing it), and we write $WL\backslash\{n\}$ for the worklist without the node n and $\cup$ for set-union on the elements of the worklist. The algorithm starts with the least valuation on all nodes and an initial worklist containing nodes with input from the environment. It enlarges the valuation within the given lattice step by step until it stabilizes, i.e., until the worklist is empty. If adding the abstract effect of one node to the current state enlarges the valuation, i.e., the set S is non-empty, those successor nodes from S are (re-)entered into the list of unfinished ones. The special case for output nodes $c!s(e)$ captures interprocess communication. Since the set of variables in the system is finite, and thus the lattice of abstract valuations, termination of the algorithm is immediate.

With the worklist as a set-like data-structure, the algorithm is free to work off the list in any order. In practice, more deterministic data-structures and traversal strategies are appropriate, for instance traversing the graph in a breadth-first manner (see [131] for a broader discussion or various traversal strategies).

After termination the algorithm yields two mappings $\eta^\alpha_{pre} : Node \rightarrow Val^\alpha$ and $\eta^\alpha_{post} : Node \rightarrow Val^\alpha$. On a location l, the result of the analysis is given by $\eta^\alpha(l) = \bigvee\{\eta^\alpha_{post}(\tilde{n}) \mid \tilde{n} = \tilde{l} \longrightarrow_a l\}$ (where a is an action), also written as η^α_l.

Lemma 5.7. CORRECTNESS (*may*)
The algorithm in Fig. 15 terminates. Upon termination, the algorithm gives back the least solution to the constraint set as given by the equations (4), where the transfer function is defined by Table 10.

Proof. The set of variables is finite and the lattice of abstract valuations is complete (Def. 2.2). Transfer functions are monotonic. Using Theorem 2.1, it is straightforward to show that an abstract valuation reached upon the termination of the algorithm on Fig. 15 is the least solution to the constraint set as given by the equations (4). □

Must Analysis

The *must* analysis is almost dual to the *may* analysis. The *must* analysis marks the variables that are *guaranteed* to be influenced by data from outside. The transfer function that describes the change of the abstract valuation depending on the action at the node is defined in Table 11. For inputs $c?s(x)$, the transfer function assigns $\bot$ to x if the signal is sent to P with reliable data only. This

$$
\begin{aligned}
f(c?s(x))\eta^{\alpha} &= \begin{cases} \eta^{\alpha}{}_{[x \mapsto \top]} & s \notin Sig_{int} \\ \eta^{\alpha}{}_{[x \mapsto \bigwedge\{[\![e]\!]_{\eta^{\alpha}} \mid n'=g \rhd c!s(e)\}]} & s \in Sig_{int} \end{cases} \\
f(g \rhd c!s(e))\eta^{\alpha} &= \eta^{\alpha} \\
f(g \rhd x := e)\eta^{\alpha} &= \eta^{\alpha}{}_{[x \mapsto [\![e]\!]_{\eta^{\alpha}}]} \\
f(g \rhd \mathit{set}\ t := e)\eta^{\alpha} &= \eta^{\alpha}{}_{[t \mapsto \mathit{on}([\![e]\!]_{\eta^{\alpha}})]} \\
f(g \rhd \mathit{reset}\ t)\eta^{\alpha} &= \eta^{\alpha}{}_{[t \mapsto \mathit{off}]} \\
f(g_t \rhd \mathit{reset}\ t)\eta^{\alpha} &= \eta^{\alpha}{}_{[t \mapsto \mathit{off}]}
\end{aligned}
$$

Table 11. Must analysis: transfer functions/abstract effect for process P

```
input: the flow graph of the program
output: $\eta^{\alpha}_{pre}, \eta^{\alpha}_{post}$;

$\eta^{\alpha}(n) = \eta^{\alpha}_{init}(n)$;
$WL = \{n \mid \alpha_n = g \rhd x := e\}$;

repeat
    pick $n \in WL$;
    if $n = g \rhd c!s(e)$ then
        let $S' = \{n' \mid n' = c?s(x)$ and $[\![e]\!]_{\eta^{\alpha}(n)} \not\geq [\![x]\!]_{\eta^{\alpha}(n')}\}$
        in
            for all $n' \in S'$: $\eta^{\alpha}(n') := f_{n'}(\eta^{\alpha}(n'))$;
    let $S = \{n'' \in succ(n) \mid f_n(\eta^{\alpha}(n)) \not\geq \eta^{\alpha}(n'')\}$
    in
        for all $n'' \in S$: $\eta^{\alpha}(n'') := f_n(\eta^{\alpha}(n))$;
    $WL := WL \backslash \{n\} \cup S \cup S'$;
until $WL = \emptyset$;

$\eta^{\alpha}_{pre}(n) = \eta^{\alpha}(n)$;
$\eta^{\alpha}_{post}(n) = f_n(\eta^{\alpha}(n))$
```

Fig. 16. *Must* analysis: worklist algorithm

means the values after reception correspond to the greatest lower bound over all expressions which can occur in a matching send-action.

Similar to the may analysis, the data-flow problem is specified for each node n of the flow graph by two inequations (5) (see Table 11). Analogously, the *greatest* fixpoint of the constraint set can be found iteratively by a worklist algorithm (cf. Figure 16). Upon the start of the analysis, at each node the variables' values are assumed to be defined, i.e., the initial valuation is the greatest one: $\eta^{\alpha}_{init}(n) = \eta_{\top}$.

$$\begin{aligned} \eta^{\alpha}_{post}(n) &\leq f_n(\eta^{\alpha}_{pre}(n)) \\ \eta^{\alpha}_{pre}(n) &\leq \bigwedge\{\eta^{\alpha}_{post}(n') \mid (n', n) \text{ in flow relation}\} \end{aligned} \tag{5}$$

As in the case of may analysis, termination of the algorithm follows from the finiteness of the set of variables. After termination the algorithm yields two mappings $\eta^{\alpha}_{pre}, \eta^{\alpha}_{post} : \mathit{Node} \to \mathit{Val}^{\alpha}$. On a location l, the result of the analysis is given by $\eta^{\alpha}(l) = \bigwedge\{\eta^{\alpha}_{post}(\tilde{n}) \mid \tilde{n} = \tilde{l} \longrightarrow_a l\}$, also written as η^{α}_{l}.

Lemma 5.8. CORRECTNESS (*must*)
The algorithm in Fig. 16 terminates. Upon termination, the algorithm gives back the greatest solution to the constraint set as given by equations (5), where the transfer function is defined by Table 11.

5.4 Program Transformation

For model checking, we cannot live with the infinity of data injected from outside by the chaotic environment. Therefore, we abstract this infinity into one single abstract value $\mathbb{T}$. For chaotically influenced timers, we should differentiate among deactivated timers, timers that are ready to *expire immediately* and timers that are active but *not ready* to expire in the *current* time slice. The timer abstraction introduced in Sec. 4.2 cannot be used for this purpose, because k^+ abstraction is suitable only to represent an active timer that cannot expire immediately. Therefore, we will need a more refined abstraction. Since the abstract system is still open, we close it by embedding the chaotic environment into the system. Special care is taken to properly embed chaotic behaviour wrt. timed behaviour.

Based on the result of the analysis, we transform a given specification *Spec* into a closed one denoted by $\mathit{Spec}^{\sharp}$, where communication with the environment is embedded into the system, all the data exchanged with it is abstracted.

The intention is to use the information collected in the analysis about the influence of the environment to reduce the state space. Depending on whether one relies on the may-analysis alone (which variable occurrences may be influenced from the outside) or takes into account the results of both analyses (additional information which variable occurrences are definitely chaotic) the precision of the abstraction varies. Using only the may-information overapproximates the system (further) but in general leads to a smaller state space. The second option, on the other hand, gives a more precise abstraction and thus less false negatives.

Here we describe only the transformation based on a combination of *may* and *must* analyses, since the alternative transformation using the results of the may analysis is simpler. The transformation not only closes the system but also optimises it by removing unnecessary instances of variables or expressions which are guaranteed to be $\top$. The transformation is defined by the rules in Table 12.

Overloading the symbols $\top$ and $\bot$ we mean for the rest of the paper: the value of $\top$ for a variable at a location refers to the result of the must analysis, i.e., the definite knowledge that the data is chaotic for all runs. Dually, $\bot$ stands for the definite knowledge of the may analysis, i.e., for data which is never influenced from outside. Additionally, we write $\top\!\!\!\!\bot$ in case neither analysis gives a definite answer. The set of variables whose value is $\top\!\!\!\!\bot$ in at least one location is denoted further as $Var_{\top\!\!\!\!\bot}$. The strict lifting of a valuation η^{α} to expressions is denoted by $[\![.]\!]_{\eta^{\alpha}}$.

We extend each data domain by the additional value $\mathbb{T}$, representing unknown, chaotic, data, i.e., we assume now domains such as $\mathbb{N}^{\mathbb{T}} = \mathbb{N} \dot{\cup} \{\mathbb{T}\}$, $Bool^{\mathbb{T}} = Bool \dot{\cup} \{\mathbb{T}\}, \ldots, D^{\mathbb{T}} = D \dot{\cup} \{\mathbb{T}\}$ where we do not distinguish notationally the various types of chaotic values. These values $\mathbb{T}$ are considered as the largest values, i.e., we introduce $\leq$ as the smallest reflexive relation with $v \leq \mathbb{T}$ for all elements v (separately for each domain). The strict lifting of a valuation $\eta^{\sharp}$ ($\eta^{\sharp}\colon Var \to D^{\mathbb{T}}$) to expressions is denoted by $[\![.]\!]_{\eta^{\sharp}}$.

The transformation of untimed guards is straightforward: guards influenced by the environment are taken non-deterministically, i.e., a guard g at a location l is replaced by *true*, if $[\![g]\!]_{\eta^{\alpha}_{l}} = \top$. A guard g whose value at a location l is $\top\!\!\!\!\bot$ is treated dynamically on the extended data domain, i.e. it is replaced by $((g = \mathit{true}) \vee (g = \mathbb{T}))$. The guards whose value at a location is $\bot$ are left unmodified. Further, the transformed guards are denoted as $g^{\sharp}$.

For assignments, we distinguish between the variables that carry the value $\top\!\!\!\!\bot$ in at least one location and the rest. Assignments of $\mathbb{T}$ to variables that take $\top\!\!\!\!\bot$ at no location are omitted (rule T-ASSIGN$_1$ of Table 12). Assignments of concrete values are left untouched and assignments to variables that are marked by $\top\!\!\!\!\bot$ in at least one location are performed on the extended data domain. If an assigned expression e is guaranteed to be influenced from the outside, i.e., $[\![e]\!]_{\eta^{\alpha}_{l}} = \top$, we get rid of the expression and assign $\mathbb{T}$ directly (rule T-ASSIGN$_2$ of Table 12).

The interpretation of *timer variables* on the extended domain requires special attention. Chaos can influence timers only via the *set*-operation by setting it to a chaotic value in the *on*-state. Therefore, the domain of timer values contains the additional chaotic value $on(\mathbb{T})$. Since we need the transformed system to show at least the behaviour of the original one, we must provide proper treatment of the rules involving $on(\mathbb{T})$, i.e., the TIMEOUT-, the TDISCARD-, and the TICK-rule of Table 9. As $on(\mathbb{T})$ stands for any value of active timers, it must cover the cases where timeouts and timer discards are enabled (because

$$\frac{l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in Edg \qquad x \notin Var_{\mathbb{T}} \qquad [\![e]\!]_{\eta_l^\alpha} = \top}{l \longrightarrow_{g^\sharp \,\triangleright\, skip} \hat{l} \in Edg^\sharp}\ \text{T-ASSIGN}_1$$

$$\frac{l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in Edg \qquad x \in Var_{\mathbb{T}} \qquad [\![e]\!]_{\eta_l^\alpha} = \top}{l \longrightarrow_{g^\sharp \,\triangleright\, x:=\mathbb{T}} \hat{l} \in Edg^\sharp}\ \text{T-ASSIGN}_2$$

$$\frac{l \in Loc_i}{l \longrightarrow_{t=on(\mathbb{T}) \,\triangleright\, set\ t:=\mathbb{T}+} l \in Edg^\sharp}\ \text{T-NOTIMEOUT}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, set\ t:=e} \hat{l} \in Edg \qquad [\![e]\!]_{\eta_l^\alpha} = \top}{l \longrightarrow_{g^\sharp \,\triangleright\, set\ t:=\mathbb{T}} \hat{l} \in Edg^\sharp}\ \text{T-SET}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, c!(s,e)} \hat{l} \in Edg \qquad c \in In_{env}}{l \rightarrow_{g^\sharp \,\triangleright\, skip} \hat{l} \in Edg^\sharp}\ \text{T-OUTPUT}_{ext}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, c!(s,e)} \hat{l} \in Edg \qquad c \notin in_{env} \qquad [\![e]\!]_{\eta_l^\alpha} = \top}{l \longrightarrow_{g^\sharp \,\triangleright\, c!(s,\mathbb{T})} \hat{l} \in Edg^\sharp}\ \text{T-OUTPUT}_{int}$$

$$\frac{l \longrightarrow_{c?s(x)} \hat{l} \in Edg \qquad s \in Sig_{ext} \qquad x \in Var_{\mathbb{T}}}{l \longrightarrow_{g_{t_P} \,\triangleright\, x:=\mathbb{T}} \hat{l} \in Edg^\sharp}\ \text{T-INPUT}^1_{ext}$$

$$\frac{l \longrightarrow_{c?s(x)} \hat{l} \in Edg \qquad s \in Sig_{ext} \qquad x \notin Var_{\mathbb{T}}}{l \longrightarrow_{g_{t_P} \,\triangleright\, skip} \hat{l} \in Edg^\sharp}\ \text{T-INPUT}^2_{ext}$$

$$\frac{l \in Loc_i}{l \longrightarrow_{g_{t_P} \,\triangleright\, set\ t_P:=1} l \in Edg^\sharp}\ \text{T-NOINPUT}$$

$$\frac{l \longrightarrow_{c?s(x)} \hat{l} \in Edg \qquad s \in Sig_{int} \qquad x \notin Var_{\mathbb{T}} \qquad [\![x]\!]_{\eta_l^\alpha} = \top}{l \longrightarrow_{c?s(\_)} \hat{l} \in Edg^\sharp}\ \text{T-INPUT}_{int}$$

Table 12. Transformation

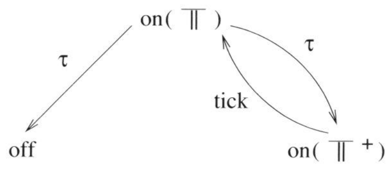

Fig. 17. Timer abstraction

of the concrete value $on(0)$) as well as *disabled* (because of $on(n)$ with $n \geq 1$). The second one is necessary, since the enabledness of the tick steps depends on the disabledness of timeouts and timer discards via the blocked-condition.

To distinguish the two cases, we introduce a refined abstract value $on(\mathbb{T}^+)$ for chaotic timers, representing all *on*-settings larger than or equal to 1 (see Fig. 17). The order on the domain of timer values is given as the smallest reflexive order relation such that $on(0) \leq on(\mathbb{T})$ and $on(n) \leq on(\mathbb{T}^+) \leq on(\mathbb{T})$, for all $n \geq 1$. To treat the case where the abstract timer value $on(\mathbb{T})$ denotes *absence* of immediate timeout, we add edges $l \longrightarrow_{t=on(\mathbb{T}) \,\triangleright\, set\ t:=\mathbb{T}^+} l \in Edg^\sharp$ which set back the timer value to $\mathbb{T}^+$ representing a non-zero delay (rule T-NoTimeout of Table 12). Rule T-Set of Table 12 transforms setting a timer to a value given by expression e into setting the timer to $\mathbb{T}$ if e is always influenced by the environment. The decreasing operation needed in the Tick$_P$-rule of Table 9 is defined in extension to the definition on values from $on(\mathbb{N})$ on $\mathbb{T}^+$ by $on(\mathbb{T}^+) - 1 = on(\mathbb{T})$. Note that the operation is left undefined on $\mathbb{T}$. Timeout guards g_t are transformed into $((t = on(0)) \vee (t = on(\mathbb{T})))$ denoted further $g_t^\sharp$.

Lemma 5.9.
Let $S^\sharp$ be LTSs obtained by applying the rules of Table 9 and Table 6 to $Spec^\sharp$. Let $(l, \eta^\sharp)$ be a configuration of $S^\sharp$. If $(l, \eta^\sharp) \rightarrow_{tick}$, then $[\![t]\!]_{\eta^\sharp} \not\in \{on(0), on(\mathbb{T})\}$, for all timers.

Proof. If at least one timer has a value from $\{on(0), on(\mathbb{T})\}$ then either timeout (rule Timeout Table 9), or discard of timeout (rule TDiscard Table 9), or setting the timer to $on(\mathbb{T}^+)$ possible by the rule T-NoTimeout is enabled. Since there is an enabled step, the system is not blocked and no *tick*-step is possible. □

We have abstracted from data coming from outside, but so far, the system is still *open*. The rules T-Input$^1_{ext}$, T-Input$^2_{ext}$, T-NoInput, T-Input$_{int}$ T-Output$_{ext}$ of Table 12 *embed* the chaotic environment's behaviour into the system. Embedding concerns only communication statements. For communication statements, we distinguish between signals going to or coming from the environment and those exchanged within the system. Outputs to the outside are skipped (rule T-Output$_{ext}$). Outputs within the system are basically left unmodified. If an expression e is guaranteed to be influenced from the outside, i.e., $[\![e]\!]_{\eta_l^\alpha} = \top$, we get rid of the expression and send $\top$ directly (rule T-OUTPUT$_{int}$ of Table 12).

Inputs from the outside are treated similarly. However we cannot just replace an input from the environment by an unconditionally enabled assignment of $\top$ to the variable influenced by the input. It would render potential *tick*-steps impossible by ignoring the situation when the chaotic environment does *not* send any message. The core of the problem is that with the timed semantics, the chaotic environment not just sends streams of messages but "chaotically timed" message streams, i.e. with *tick*'s interspersed at arbitrary points.

We embed the chaotic nature of the environment by adding to each process specification $Spec_P$ a new timer variable t_P, used to guard the input from outside. These timers behave in the same manner as the "chaotic" timers, except that we do not allow the new t_P timers to become deactivated. The expiration of timer t_P is expressed by the time guard $(t_P = on(0))$ denoted by g_{t_P}. When guard g_{t_P} is *true*, a non-deterministic choice is made between the assignment of an abstract value $\top$ to variable x (rule T-INPUT$^1_{ext}$ of Table 12) and the setting of timer t_P that postpones inputs from the environment till the next time slice (rule T-NOINPUT of Table 12). The transformation gets rid of all expressions where at least one variable is guaranteed to be influenced from the outside. Therefore, we skip the assignment of $\top$ to x, if the variable x is not a $\bot$ variable (rule T-INPUT$^2_{ext}$ of Table 12).

Since communication statements using the external signals and environment input channels are replaced by skip in case of output and by assignment or skip in case of input, the embedding yields a *closed* system specification which we denote by $Spec^\sharp$.

5.4.1 Preservation Result

Further, let *Spec* be a specification of the original system and $Spec^\sharp$ be the specification obtained as the result of the transformation of *Spec* according to the rules of Table 12. Let S and $S^\sharp$ be LTSs obtained by applying the rules of Table 9 and Table 6 to *Spec* and $Spec^\sharp$, respectively. Note that the rules of Table 9 are lifted to data domains with $\top$ values. The relationship between the original and the closed systems will be based on *path inclusion up to stuttering* (Def. 2.28), i.e. $S \models \phi$ if $S^\sharp \models \phi$ for any next-free *LTL* formula ϕ mentioning only variables never influenced by the environment. It will take the rest of this section to establish this claim.

The set of variables $Var^\sharp$ for $S^\sharp$ equals the original *Var*, except that for each process P of the system, a fresh timer-variable t_P is added to its local variables, i.e., $Var^\sharp = Var \mathbin{\dot\cup} \{t_{P_1}, \ldots, t_{P_n}\}$. Based on the data-flow analysis, the transformation considers certain variable instances as chaotic and unreliable. Hence to compare the configurations of S and $S^\sharp$, we have to take η^α into account. Variable instances that are not influenced by the environment should have the same values in S and $S^\sharp$. Variable instances whose value depends on the system run should have the same value when they are reliable and $\mathbb{T}$ when they are unreliable. By the transformation, we get rid of variable instances that are guaranteed to be influenced from outside. In this case, we cannot relate a value of the variable in S to its value in $S^\sharp$. Therefore, we require $\llbracket x \rrbracket_{\eta^\alpha} = \top$ for such variable instances. Relative to a given analysis η^α, we define the relation $\leq$ on valuations as follows.

Definition 5.5. RELATION $\leq$ ON VALUATIONS
Given η^α, $\eta \leq \eta^\sharp$ iff the following conditions hold:

- *for all process variables $x \in Var$: either $\llbracket x \rrbracket_\eta = \llbracket x \rrbracket_{\eta^\sharp}$, or $\llbracket x \rrbracket_{\eta^\sharp} = \mathbb{T}$, or $\llbracket x \rrbracket_{\eta^\alpha} = \top$;*
- *for all timer variables $t \in Var$: $\llbracket t \rrbracket_\eta \leq \llbracket t \rrbracket_{\eta^\sharp}$.*

Introducing the additional $\mathbb{T}$-value renders a system less deterministic. Before proving the corresponding branching simulation lemmas, the next lemmas state monotonicity of the semantics of expressions, monotonicity of updating a valuation, and preservation of the $\leq$-relation by the count-down operation on timers.

Lemma 5.10.
Let e be an arbitrary expression and η and $\eta^\sharp$ two valuations $Var \to D$ and $Var \to D^{\mathbb{T}}$. Then $\eta \leq \eta^\sharp$ implies $\llbracket e \rrbracket_\eta \leq \llbracket e \rrbracket_{\eta^\sharp}$.

Proof. Straightforward. □

Lemma 5.11.
If $\eta \leq \eta^\sharp$ and $v \leq v^{\mathbb{T}}$, then $\eta_{[x \mapsto v]} \leq \eta^\sharp{}_{[x \mapsto v^{\mathbb{T}}]}$.

Proof. Straightforward. □

Lemma 5.12.
Assume $\eta \leq \eta^\sharp$ with $\llbracket t \rrbracket_\eta \neq on(0)$ and $\llbracket t \rrbracket_{\eta^\sharp} \notin \{on(0), on(\mathbb{T})\}$ for all timers $t \in Var$. Then $\eta_{[t \mapsto (t-1)]} \leq \eta^\sharp{}_{[t \mapsto (t-1)]}$.

Proof. The $\leq$-relation on valuations is defined by pointwise lifting of the corresponding relation on the values. The preservation results for single timer variables follow directly from the definition of $\leq$ on the domain $\{off, on(n) \mid n \in \mathbb{N}^{\mathbb{T}} \dot{\cup} \{\mathbb{T}^+\}\}$ and the definition of the decreasing operation "$-$" on this domain. □

Before relating traces of the original system to traces of the closed one, we define order relations on configurations. To relate states from $Loc \times Val$ with those from $Loc^\sharp \times Val^\sharp$, we define the relation $\leq$ on states as the smallest relation such that $(l, \eta) \leq (l, \eta^\sharp)$ if $\eta \leq \eta^\sharp$.

Definition 5.6. RELATION $\leq$ ON PROCESS STATES
Let $Spec_P$ be a process specification, and $Spec_{P^\sharp}$ be the process specification obtained by transforming $Spec_P$ according to the rules of Table 12. Let P and $P^\sharp$ be LTSs built by applying the rules of Table 9 to $Spec_P$ and $Spec_{P^\sharp}$, respectively. Let $\sigma = (l, \eta)$ and $\sigma^\sharp = (l, \eta^\sharp)$ be states of P and $P^\sharp$ respectively. We write $\sigma \leq \sigma^\sharp$ iff $\eta \leq \eta^\sharp$.

Definition 5.7. RELATION $\leq$ ON MESSAGES
Messages from $M = Sig \times Id \times D^\top$ are related by $\leq$ as follows:
$s(pid, v) \leq s(pid, v^\top)$ if $v \leq v^\top$.

Comparing queues modelling channels, external messages are ignored. The internal messages must coincide wrt. signals. The values parameterizing the internal messages must be related by the $\leq$-relation, i.e. queues q and $q^\sharp$ are related by $\leq$ iff $q^\sharp$ is q with all messages from the environment projected out.

Definition 5.8. RELATION $\leq$ ON QUEUES
We define $\leq$ on queues inductively as follows:

- $\epsilon \leq \epsilon$,
- $s(env, v) :: q \leq q^\sharp$ *iff* $q \leq q^\sharp$,
- $s(pid, v) :: q \leq s(pid, v^\top) :: q^\sharp$ *iff* $v \leq v^\top$, $q \leq q^\sharp$ *and* $pid \neq env$.

Since the transformation skips all the outputs to the environment, queues modelling input channels of the environment are always empty in the closed system. We could remove those queues from the closed system, as it is done in the implementation (see Sec. 5.5) of our approach. For the sake of readability, we keep them here.

Definition 5.9. RELATION $\leq$ ON CHANNEL STATES
Relation $\leq$ on channel states is defined as follows:

- $(c, q) \leq (c, q^\sharp)$ *if* $q \leq q^\sharp$ *and* $c \notin In_{env}$;
- $(c, q) \leq (c, \epsilon)$ *if* $c \in In_{env}$.

The definitions $\leq$ are extended to configurations in the obvious manner.

Definition 5.10. RELATION $\leq$ ON CONFIGURATIONS
Let Spec be a specification, and $Spec^\sharp$ be the specification obtained by transforming Spec according to the rules of Table 12. Let S and $S^\sharp$ be LTSs built by applying the rules of Table 9, rules IN *and* OUT *of Table 2 and the rules of Table 6 to Spec and $Spec^\sharp$ respectively. We write $\gamma \leq \gamma^\sharp$ for configurations $\gamma = (\gamma_1, \ldots, \gamma_n)$ and $\gamma^\sharp = (\gamma_1^\sharp, \ldots, \gamma_n^\sharp)$ of S and $S^\sharp$, respectively, iff $\gamma_i \leq \gamma_i^\sharp$ for all $i = 1..n$.*

Lemma 5.13.
Let Spec be a specification, and $Spec^\sharp$ be the process specification obtained by transforming Spec according to the rules of Table 12. Let $\eta \leq \eta^\sharp$ be two evaluations.

1. *Let g be a guard of an edge in Spec originating in location l and $g^\sharp$ its analogue in Spec$^\sharp$. If $[\![g]\!]_\eta = true$, then $[\![g^\sharp]\!]_{\eta^\sharp} = true$.*
2. *Let t be a timer in Spec. If $[\![t]\!]_\eta = on(0)$, then $[\![t]\!]_{\eta^\sharp} \in \{on(0), on(\mathbb{T})\}$.*

Proof. Follows directly from Def. 5.10 and the transformation of guards.

Note that we are interested in preservation of properties that can be expressed by *LTL–X* formulas. Formulas of *LTL–X* are interpreted over Kripke structures (see Sec. 2.3), thus the $\leq$-relation on configurations could be enough to establish our claim about preservation. To keep the proofs of this section more intuitive, we also define the observable effect and the $\leq$-relation on labels. The observable effect renames to τ the labels concerning communication with the environment.

Definition 5.11. OBSERVABLE EFFECT
The observable effect $\ulcorner\cdot\urcorner$: Lab $\rightarrow$ Lab on labels of system S is given as the following equations:

$$\ulcorner c?s(pid, v)\urcorner = \begin{cases} \tau & \text{if } pid = env \text{ or } c \in in_{env} \\ c?s(pid, v) & \text{otherwise} \end{cases}$$

$$\ulcorner c!s(pid, v)\urcorner = \begin{cases} \tau & \text{if } pid = env \text{ or } c \in in_{env} \\ c!s(pid, v) & \text{otherwise} \end{cases}$$

$$\ulcorner \tau \urcorner = \tau$$

$$\ulcorner tick \urcorner = tick$$

The observable effect $\ulcorner\cdot\urcorner$: Lab $\rightarrow$ Lab on labels of system $S^\sharp$ is given by an identity function.

Definition 5.12. RELATION $\leq$ ON LABELS
Relation $\leq$ on labels is the smallest relation $\leq \subseteq$ Lab $\times$ Lab such that

- $\tau \leq \tau$,
- *tick $\leq$ tick, and*
- $c?s(pid, v) \leq c?s(pid, v^\mathbb{T})$ *as well as* $c!s(pid, v) \leq c!s(pid, v^\mathbb{T})$ *iff* $v \leq v^\mathbb{T}$.

Further, we show that for each trace ζ of S there is a trace χ of $S^\sharp$ having the same stuttering-free projection as ζ. It guarantees that we may transfer positive verification results from the closed system to the original one for properties that can be expressed by *LTL–X*-formulas mentioning only variables not influenced by the environment. First, we define a trace equivalence relating traces of S to traces of $S^\sharp$.

Definition 5.13.
Let ζ be a trace of S and χ be a trace of $S^\sharp$. We write $\zeta \equiv_\mathcal{R} \chi$ iff there is $\mathcal{R} \subseteq \mathbb{N} \times \mathbb{N}$ such that $(0,0) \in \mathcal{R}$ and

- $(i,j) \in \mathcal{R}$ *implies that* $\zeta_\gamma(i) \leq \chi_\gamma(j)$ *and that one of the following conditions holds:*
  1. $\zeta_\lambda(i+1) \in \{\ c_i!s(pid,v), c_o?s(env,v) \mid c_i \in in_{env}\}$, $\zeta_\gamma(i+1) \leq \chi_\gamma(j)$ *and* $(i+1,j) \in \mathcal{R}$;
  2. $\zeta_\lambda(i+1) \notin \{\ c_i!s(pid,v), c_o?s(env,v) \mid c_i \in in_{env}\}$, $\ulcorner\zeta_\lambda(i+1)\urcorner \leq \ulcorner\chi_\lambda(j+1)\urcorner$, $\zeta_\gamma(i+1) \leq \chi_\gamma(j+1)$ *and* $(i+1,j+1) \in \mathcal{R}$;
  3. $\zeta_\lambda(i+1) = tick$, $\chi_\lambda(j+m) = tick$, $\zeta_\gamma(i+1) \leq \chi_\gamma(j+m)$, $(i+1, j+m) \in \mathcal{R}$, *and for all* $1 \leq k < m$: $\chi_\lambda(j+k) = \tau$, $\zeta_\gamma(i) \leq \chi_\gamma(j+k)$ *and* $(i, j+k) \in \mathcal{R}$.

We write $S \preceq_{\mathcal{R}} S^\sharp$ *iff for each trace* ζ *of* S *there is a trace* χ *of* $S^\sharp$ *such that* $\zeta \equiv_{\mathcal{R}} \chi$ *for some* $\mathcal{R} \subseteq \mathbb{N} \times \mathbb{N}$.

We illustrate conditions 1, 2 and 3 of Def. 5.13 by Fig. 18, 19 and 20 respectively. For all three examples, we assume that $\zeta^{(i)} \equiv_{\mathcal{R}} \chi^{(j)}$ for some $\mathcal{R} \subseteq \mathbb{N}\times\mathbb{N}$. By Def. 5.13, $(i,j) \in \mathcal{R}$, and thus $\zeta_\gamma(i) \leq \chi_\gamma(j)$.

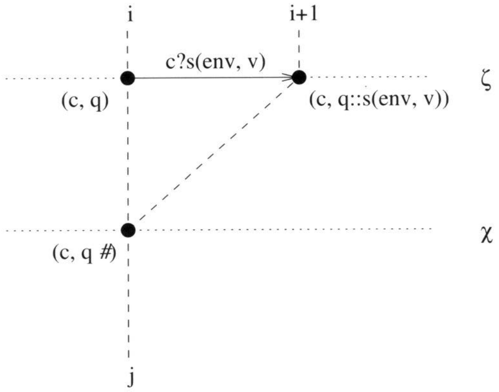

Fig. 18. Skipping communication with the environment

In Fig. 18, we assume that the $(i+1)$th-step of ζ is step $c_o?s(env,v)$ receiving message $s(env,v)$ from the environment. Closing embeds effects of consuming messages from the environment into the system and skips external communication. Condition 1 of Def. 5.13 reflects it. The $c_o?s(env,v)$-step does not change the values of the variables and so the configuration $\zeta_\gamma(i+1)$ reached by the step is in the $\leq$-relation with $\chi_\gamma(j)$. We add $(i+1,j)$ to $\mathcal{R}$, and thus obtain $\zeta^{(i+1)} \equiv_{\mathcal{R}} \chi^{(j)}$ for the extended $\mathcal{R}$.

Each step of S that is not a *tick*-step and not communication with the environment is mimicked by the same step of $S^\sharp$ having the effect related by the $\leq$-relation. In Fig. 19, we assume that the $(i+1)$th-step of ζ is step τ consuming message $s(env,v)$ from the queue modelling channel c, and that value v is influenced by environment, i.e. it is abstracted to $\mathbb{T}$ in $S^\sharp$.

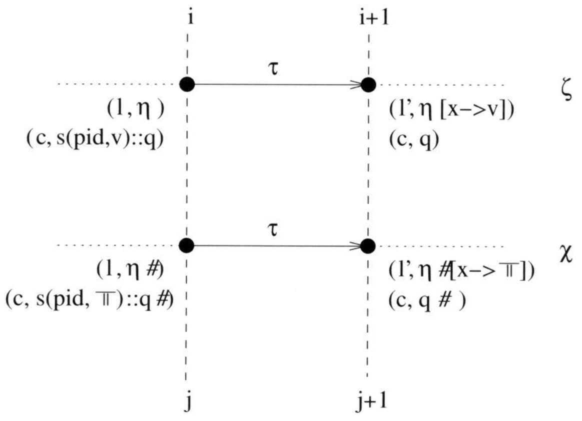

Fig. 19. Mimicking τ steps

Since $\zeta_\gamma(i) \leq \chi_\gamma(j)$, consuming message $s(pid, v)$ in S can be mimicked by consuming message $s(pid, \mathbb{T})$ in $S^\sharp$. The $(i+1)$th-step of ζ leads to a change of location and valuation of the variable x to v. The mimicking $(j+1)$th-step of χ changes location in the same way and modifies the valuation of x to $\mathbb{T}$, and so $\zeta_\gamma(i+1) \leq \chi_\gamma(j+1)$. We add $(i+1, j+1)$ to $\mathcal{R}$, and thus get $\zeta^{(i+1)} \equiv_\mathcal{R} \chi^{(j+1)}$ the extended $\mathcal{R}$.

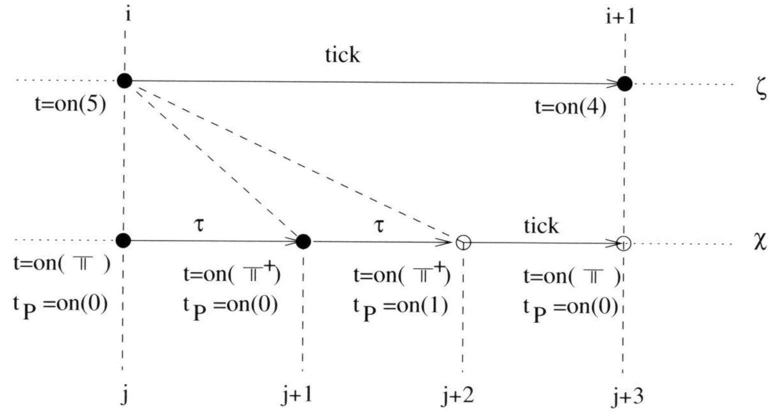

Fig. 20. Mimicking a *tick*-step

In Fig. 20, we assume that $\zeta^{(i)} \equiv_{\mathcal{R}} \chi^{(j)}$ for some $\mathcal{R} \subseteq \mathbb{N} \times \mathbb{N}$, and that the $(i+1)$th-step of ζ is step *tick*. We also assume that timer t is influenced by the environment.

System S is blocked in $\zeta_\gamma(i)$, however, $\mathit{blocked}(\chi_\gamma(j))$ is not necessarily *true*. In the closed system, timer t can be in state $on(\mathbb{T})$, and special timer t_P added by the transformation is $on(0)$ in $\chi_\gamma(j)$. To be able to mimic the *tick*-step, we should reach a configuration where $S^\sharp$ is blocked as well. Timers having value $on(\mathbb{T})$ should be set to $on(\mathbb{T}^+)$ (see rule T-NoTimeout in Table 13). Special timers t_P added by the transformation should be set to $on(1)$ (see rule T-NoInput in Table 13).

First we take a τ-step suspending a timeout of timer t, i.e. setting t to $on(\mathbb{T}^+)$. The configuration $\chi_\gamma(j+1)$ reached by this step is in relation $\leq$ with $\zeta_\gamma(i)$. We add $(i, j+1)$ to $\mathcal{R}$.

In system $S^\sharp$, we do not take step setting special process timers t_P to $on(1)$ until we have to mimic a *tick*-step of S. By a τ-step, we set timer t_P to $on(1)$. The configuration $\chi_\gamma(j+2)$ reached by this step is in relation $\leq$ with $\zeta_\gamma(i)$. We add $(i, j+2)$ to $\mathcal{R}$.

Now system $S^\sharp$ is blocked, and we may take a *tick*-step in $S^\sharp$. The configuration $\chi_\gamma(j+3)$ reached by this step is in relation $\leq$ with $\zeta_\gamma(i+1)$. We add $(i+1, j+3)$ to $\mathcal{R}$, and thus obtain $\zeta^{(i+1)} \equiv_{\mathcal{R}} \chi^{(j+3)}$ for some $\mathcal{R} \subseteq \mathbb{N} \times \mathbb{N}$.

Let π_ζ and π_χ be paths corresponding to traces ζ and χ respectively, i.e. $\pi_\zeta = \zeta_\gamma(0)\zeta_\gamma(1)\ldots$ and $\pi_\chi = \chi_\gamma(0)\chi_\gamma(1)\ldots$. Next, we show that $\zeta \equiv_{\mathcal{R}} \chi$ for some $\mathcal{R} \subseteq \mathbb{N} \times \mathbb{N}$ implies $\pi_\zeta \equiv_{st} \pi_\chi$.

Lemma 5.14.
Let ζ and χ be traces of S and $S^\sharp$ respectively. Let $\zeta \equiv_{\mathcal{R}} \chi$ for some relation $\mathcal{R} \subseteq \mathbb{N} \times \mathbb{N}$. Let $\mathcal{L}\colon \Gamma \to 2^{\mathcal{P}}$ and $\mathcal{L}^\sharp\colon \Gamma^\sharp \to 2^{\mathcal{P}}$ be interpretation functions, $\mathcal{P}$ be the set of atomic propositions that mention only variables never influenced by the environment, Γ and $\Gamma^\sharp$ be sets of configurations of S and $S^\sharp$ respectively. Then $\pi_\zeta \equiv_{st} \pi_\chi$.

Proof. Since $\zeta \equiv_{\mathcal{R}} \chi$ for some $\mathcal{R} \subseteq \mathbb{N}\times\mathbb{N}$, $(0,0) \in \mathcal{R}$ by Def. 5.13. By Def. 5.13, $\mathcal{L}(Pr(\pi_\zeta)(0)) = \mathcal{L}^\sharp(Pr(\pi_\chi)(0))$.

Assume that $\zeta^{(i)} \equiv_{\mathcal{R}} \chi^{(j)}$, and prefixes $\zeta^{(i)}$ and $\chi^{(j)}$ have the same stuttering-free projection. That means $|Pr(\pi_{\zeta^{(i)}})| = |Pr(\pi_{\chi^{(j)}})|$ and that for all $0 \leq k \leq |Pr(\pi_{\zeta^{(i)}})|$, $\mathcal{L}(Pr(\pi_{\zeta^{(i)}})(k)) = \mathcal{L}^\sharp(Pr(\pi_{\chi^{(j)}})(k))$ (cf. Def. 2.27). Since $\zeta \equiv_{\mathcal{R}} \chi$, $\zeta^{(i+1)} \equiv_{\mathcal{R}} \chi^{(j+m)}$ for some $m \geq 0$ (cf. Def. 5.13).

Further, we show that for each prefix $\zeta^{(i+1)}$ there is a prefix $\chi^{(j+m)}$ for some m such that $\zeta^{(i+1)} \equiv_{\mathcal{R}} \chi^{(j+m)}$, and $\zeta^{(i+1)}$ and $\chi^{(j+m)}$ have the same stuttering-free projection. We proceed by a case analysis on conditions 1, 2 and 3 of Def. 5.13.

Case: condition 1
Assume that the $(i+1)$th-step of ζ is a communication with the environment, i.e. $\zeta_\lambda(i+1) \in \{\, c_i!s(pid, v), c_o?s(env, v) \mid c_i \in in_{env}\}$. The $(i+1)$th-step of ζ

does not change the valuation of the variables not influenced by the environment. It only adds (removes) messages from queues modelling channels. Therefore, we may conclude that $|Pr(\pi_{\zeta^{(i+1)}})| = |Pr(\pi_{\chi^{(j)}})|$ and $\mathcal{L}(Pr(\pi_{\zeta^{(i+1)}})(k)) = \mathcal{L}^\sharp(Pr(\pi_{\chi^{(j)}})(k))$ for all $0 \leq k \leq |Pr(\pi_{\zeta^{(i+1)}})|$ (cf. Def. 2.26 and Def. 2.27).

Case: condition 2
Assume that the $(i+1)$th-step of ζ is neither a communication with the environment nor a *tick*-step, i.e. $\zeta_\lambda(i+1) \notin \{\ \mathit{tick}, c_i!s(\mathit{pid}, v), c_o?s(\mathit{env}, v) \mid c_i \in \mathit{in}_{\mathit{env}}\}$. By Def. 5.13, both $\zeta_\gamma(i) \leq \chi_\gamma(j)$ and $\zeta_\gamma(i+1) \leq \chi_\gamma(j+1)$. Thus $\mathcal{L}(\zeta_\gamma(i)) = \mathcal{L}^\sharp(\chi_\gamma(j))$ and $\mathcal{L}(\zeta_\gamma(i+1)) = \mathcal{L}^\sharp(\chi_\gamma(j+1))$. Therefore, $|Pr(\pi_{\zeta^{(i+1)}})| = |Pr(\pi_{\chi^{(j+1)}})|$ and $\mathcal{L}(Pr(\pi_{\zeta^{(i+1)}})(k)) = \mathcal{L}^\sharp(Pr(\pi_{\chi^{(j+1)}})(k))$ for all $0 \leq k \leq |Pr(\pi_{\zeta^{(i+1)}})|$ (cf. Def. 2.26 and Def. 2.27).

Case: condition 3
Assume that the $(i+1)$th-step of ζ is a *tick*-step. By Def. 5.13, for all $0 \leq n < m$, $\zeta_\gamma(i) \leq \chi_\gamma(j+n)$ and $\zeta_\gamma(i+1) \leq \chi_\gamma(j+m)$. Thus, for all $0 \leq n < m$, $\mathcal{L}(\zeta_\gamma(i)) = \mathcal{L}^\sharp(\chi_\gamma(j+n))$ and $\mathcal{L}(\zeta_\gamma(i+1)) = \mathcal{L}^\sharp(\chi_\gamma(j+m))$. Therefore, we may conclude that $|Pr(\pi_{\zeta^{(i+1)}})| = |Pr(\pi_{\chi^{(j+m)}})|$ and $\mathcal{L}(Pr(\pi_{\zeta^{(i+1)}})(k)) = \mathcal{L}^\sharp(Pr(\pi_{\chi^{(j+m)}})(k))$ for all $0 \leq k \leq |Pr(\pi_{\zeta^{(i+1)}})|$ (cf. Def. 2.27).

We demonstrated that for each prefix $\zeta^{(i+1)}$ there is a prefix $\chi^{(j+m)}$ for some m such that $\zeta^{(i+1)} \equiv_{\mathcal{R}} \chi^{(j+m)}$, and $\zeta^{(i+1)}$ and $\chi^{(j+m)}$ have the same stuttering-free projection. ζ and χ are infinite, and thus we may conclude that they have the same stuttering-free projections. □

Lemma 5.15.
Let $Spec_P$ be a process specification, and $Spec_{P^\sharp}$ be the process specification obtained by transforming $Spec_P$ according to the rules of Table 12. Let P and $P^\sharp$ be LTSs built by applying the rules of Table 9 to $Spec_P$ and $Spec_{P^\sharp}$, respectively. Let $\sigma = (l, \eta)$ and $\sigma^\sharp = (l, \eta^\sharp)$ be configurations of P and $P^\sharp$, respectively, such that $\sigma \leq \sigma^\sharp$. If $\mathit{blocked}(\sigma)$, then there exists $\sigma^\sharp = \sigma^\sharp_0 \rightarrow_\tau \sigma^\sharp_1 \rightarrow_\tau \ldots \rightarrow_\tau \sigma^\sharp_n$ for some configurations $\sigma^\sharp_i$ and some $n \geq 0$ such that $\sigma \leq \sigma^\sharp_i$ for all i, and $\mathit{blocked}(\sigma^\sharp_n)$.

Proof. Let $\sigma = (l, \eta)$ and $\mathit{blocked}(\sigma)$. By Lemma 5.2, none of the timers in P has the value $on(0)$. Since $\sigma \leq \sigma^\sharp$, it can be the case that some timers of $P^\sharp$ have value $on(\top)$ in $\sigma^\sharp$ (cf. Def. 5.5). Moreover, timer t_P can have the value $on(0)$ in $\sigma^\sharp$. It means that $P^\sharp$ is not blocked in $\sigma^\sharp$.

We consider only well-formed specifications, hence, S can be blocked only in input locations. Since l is an input location, nothing except an input step or a τ-step mimicking an input from the environment or a timeout may take place in $\sigma^\sharp$. According to rule T-NoTimeout of Table 12, there is an edge $l \longrightarrow_{t=on(\top) \,\triangleright\, \mathit{set}\ t:=\top^+} l \in Edg^\sharp$ for each timer t in each location. By rule Set of Table 9, we take a step $(l, \eta^\sharp) \rightarrow_\tau (l, \eta^\sharp[t \mapsto on(\top^+)])$ for each timer t having value $on(\top)$ in $\sigma^\sharp$. The states $\gamma^\sharp_i$ reachable by these steps are still in relation $\leq$ with σ (cf. Def. 5.5).

After all timers that were in state $on(\top)$ are set to $on(\top^+)$, we also need to set t_P to $on(1)$. According to the rule T-NoInput of Table 12, there is an edge $l \longrightarrow_{t=on(\mathbb{T}) \rhd\, set\ t:=\mathbb{T}^+} l \in Edg^\sharp$. By rule SET of Table 9, we set timer t_P to $on(1)$ by the τ-step $(l, \eta^\sharp) \rightarrow_\tau (l, \eta^\sharp[t_P \mapsto on(1)])$. State $\sigma_n^\sharp$ of $P^\sharp$ is still in relation $\leq$ with σ. All the timers of $P^\sharp$ have either a nonzero value or value $on(\mathbb{T}^+)$ and there are no other enabled transitions, and so $blocked(\sigma_n^\sharp)$. □

Lemma 5.16.
Let $Spec_P$ be a process specification, and $Spec_{P^\sharp}$ be the process specification obtained by transforming $Spec_P$ according to the rules of Table 12. Let P and $P^\sharp$ be LTSs built by applying the rules of Table 9 to $Spec_P$ and $Spec_{P^\sharp}$, respectively. Then for each trace ζ of P there is a trace χ of $P^\sharp$ such that $\zeta \equiv_{\mathcal{R}} \chi$ for some $\mathcal{R} \subseteq \mathbb{N} \times \mathbb{N}$.

Proof. Here, we show that for any trace ζ of P, we can build a trace χ of $P^\sharp$ such that $\zeta \equiv_{\mathcal{R}} \chi$ for some $\mathcal{R} \subseteq \mathbb{N} \times \mathbb{N}$. Initially, all variables have the same initial values and all the timers are deactivated both in P and in $P^\sharp$, hence $\sigma_0 \leq \sigma_0^\sharp$ for initial configurations σ_0 and $\sigma_0^\sharp$ of P and $P^\sharp$, respectively. Therefore, $\mathcal{R}$ is initialized as $\{(0,0)\}$. Further, we proceed by induction on the length of ζ. Each step of P is mimicked by a step of $P^\sharp$ so that conditions of Def. 5.13 are satisfied, and configurations reached by the original step and the mimicking step are related by $\leq$.

Assume that $(i, j) \in \mathcal{R}$. Now we proceed with a case analysis on the rules of Table 9.

Case: INPUT
Here we have to consider two cases: (i) process P receives a message from another process within the system; (ii) process P receives a message from the environment.

Subcase: $s(pid, v)$, $pid \neq env$
Let the $(i+1)$th-step of ζ be $(l, \eta) \rightarrow_{c_i?s(pid,v)} (\hat{l}, \eta[x \mapsto v])$. Let $\zeta_\gamma(i) = (l, \eta)$, $\zeta_\lambda(i+1) = c_i?s(pid, v)$, and $\zeta_\gamma(i+1) = (\hat{l}, \eta[x \mapsto v])$. By rule INPUT of Table 9, we have $l \longrightarrow_{c?s(x)} \hat{l} \in Edg$.

If $x \not\in Var_{\mathbb{T}}$ and $[\![x]\!]_{\eta_l^\alpha} = \top$, we get $l \longrightarrow_{c?s(\_)} \hat{l} \in Edg^\sharp$ by rule T-INPUT$_{int}$ of Table 12. By rule INPUT of Table 9, we obtain the following mimicking step $(l, \eta^\sharp) \rightarrow_{c_i?s(pid,v^\top)} (\hat{l}, \eta^\sharp)$ and define $\chi_\gamma(j+1) = (\hat{l}, \eta^\sharp)$ and $\chi_\lambda(j+1) = c_i?s(pid, v^\top)$, where $v \leq v^\top$, and add $(i+1, j+1)$ to $\mathcal{R}$. $[\![x]\!]_{\eta_l^\alpha} = \top$ and $\zeta_\gamma(i) \leq \chi_\gamma(j)$, hence $\zeta_\gamma(i+1) \leq \chi_\gamma(j+1)$ and the conditions of Def. 5.13 are satisfied.

Otherwise, the input edge is left unmodified and it is straightforward to show that we can mimic the $(i+1)$th-step of ζ by the $(j+1)$th-step of χ, so that the conditions of Def. 5.13 are satisfied.

Subcase: $s(env, v)$
Assume that the $(i+1)$th-step of ζ is $(l, \eta) \rightarrow_{c_i?s(env,v)} (\hat{l}, \eta[x \mapsto v])$, i.e. $\zeta_\gamma(i) = (l, \eta)$, $\zeta_\lambda(i+1) = c_i?s(env, v)$, and $\zeta_\gamma(i+1) = (\hat{l}, \eta[x \mapsto v])$. By rule INPUT of Table 9, we have $l \longrightarrow_{c?s(x)} \hat{l} \in Edg$.

If $x \in \mathit{Var}_{\mathbb{I}}$, we get $l \longrightarrow_{g_{t_P} \,\triangleright\, x:=\mathbb{T}} \hat{l} \in \mathit{Edg}^\sharp$ by rule T-INPUT$^1_{ext}$ of Table 12. By rule INPUT of Table 9, we obtain the mimicking step $(l, \eta^\sharp) \rightarrow_\tau (\hat{l}, \eta^\sharp{[x \mapsto \mathbb{T}]})$. We define $\chi_\lambda(j{+}1) = \tau$, $\chi_\gamma(j{+}1) = (\hat{l}, \eta^\sharp{[x \mapsto \mathbb{T}]})$. Here $\ulcorner\zeta_\lambda(i+1)\urcorner \leq \ulcorner\chi_\lambda(j+1)\urcorner$. Since $\zeta_\gamma(i) \leq \chi_\gamma(j)$ and the mimicking step assigns $\mathbb{T}$ to x, $\zeta_\gamma(i{+}1) \leq \chi_\gamma(j{+}1)$. We add $(i+1, j+1)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

If $x \notin \mathit{Var}_{\mathbb{I}}$, we get $l \longrightarrow_{g_{t_P} \,\triangleright\, skip} \hat{l} \in \mathit{Edg}^\sharp$ by rule T-INPUT$^2_{ext}$ of Table 12. Since *skip* changes the location only, we obtain the mimicking step $(l, \eta^\sharp) \rightarrow_\tau (\hat{l}, \eta^\sharp)$. We define $\chi_\lambda(j+1) = \tau$, $\chi_\gamma(j+1) = (\hat{l}, \eta^\sharp)$, so $\ulcorner\zeta_\lambda(i+1)\urcorner \leq \ulcorner\chi_\lambda(j+1)\urcorner$. Moreover, $[\![x]\!]_{\eta^\alpha_{\hat{l}}} = \top$, because $x \notin \mathit{Var}_{\mathbb{I}}$ and $s \in \mathit{Sig}_{\,ext}$ (see the *must*-analysis in Section 5.3). Therefore, $\zeta_\gamma(i+1) \leq \chi_\gamma(j+1)$. We also add $(i+1, j+1)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

Case: DISCARD
Analogous to the case INPUT above.

Case: OUTPUT
Assume that the $(i+1)$th-step of ζ is $(l, \eta) \rightarrow_{c_o!s(pid,v)} (\hat{l}, \eta)$, i.e. $\zeta_\gamma(i) = (l, \eta)$, $\zeta_\lambda(i+1) = c_o!s(pid, v)$, and $\zeta_\gamma(i+1) = (\hat{l}, \eta)$. By rule OUTPUT of Table 9, we have $l \longrightarrow_{g \,\triangleright\, c!(s,e)} \hat{l} \in \mathit{Edg}$. Here we have to consider two sub-cases: (i) process P sends a message to some process within the system; (ii) process P sends a message to the environment.

Subcase: $c \notin \mathit{In}_{\,env}$
Process P sends a message to some process within the system.

If $[\![e]\!]_{\eta^\alpha_l} = \top$ and $s \notin \mathit{Sig}_{\,ext}$, we obtain $l \longrightarrow_{g^\sharp \,\triangleright\, c!(s,\mathbb{T})} \hat{l} \in \mathit{Edg}^\sharp$ by rule T-OUTPUT$_{int}$ of Table 12. Since $\zeta_\gamma(i) \leq \chi_\gamma(j)$, $[\![g^\sharp]\!]_{\eta^\sharp} = \mathit{true}$ by Lemma 5.13. By rule OUTPUT of Table 9, we get $(l, \eta^\sharp) \rightarrow_{c_o!s(pid,\mathbb{T})} (\hat{l}, \eta^\sharp)$, i.e. the output in P is mimicked by the output in $P^\sharp$. We define $\chi_\lambda(j+1) = c_o!s(pid, \mathbb{T})$ and $\chi_\gamma(j{+}1) = (\hat{l}, \eta^\sharp)$. Since $v \leq \mathbb{T}$, $\ulcorner\zeta_\lambda(i+1)\urcorner \leq \ulcorner\chi_\lambda(j+1)\urcorner$ holds for labels. Since $\zeta_\gamma(i) \leq \chi_\gamma(j)$ and the output step of the original system and the mimicking step of the closed one change only the location, $\zeta_\gamma(i+1) \leq \chi_\gamma(j+1)$. We add the pair $(i+1, j+1)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

If $[\![e]\!]_{\eta^\alpha_l} \neq \top$ then only the guard of the output edge is transformed and $l \longrightarrow_{g^\sharp \,\triangleright\, c!(s,e)} \hat{l} \in \mathit{Edg}^\sharp$. Since $\zeta_\gamma(i) \leq \chi_\gamma(j)$, $[\![g^\sharp]\!]_{\eta^\sharp}$ is *true* by Lemma 5.13 and $[\![e]\!]_\eta \leq [\![e]\!]_{\eta^\sharp}$ by Lemma 5.11. By rule OUTPUT of Table 9, we get transition $(l, \eta^\sharp) \rightarrow_{c_o!s(pid,v^\top)} (\hat{l}, \eta^\sharp)$, where $v^\top = [\![e]\!]_{\eta^\sharp}$. The output in P is mimicked by the output in $P^\sharp$. We define $\chi_\lambda(j+1) = c_o!s(pid, v^\top)$ and $\chi_\gamma(j+1) = (\hat{l}, \eta^\sharp)$. Since $[\![e]\!]_\eta \leq [\![e]\!]_{\eta^\sharp}$, $\ulcorner\zeta_\lambda(i+1)\urcorner \leq \ulcorner\chi_\lambda(j+1)\urcorner$ holds for labels. Since the output step of the original system and the mimicking step of the closed one change only the location and $\zeta_\gamma(i) \leq \chi_\gamma(j)$, $\zeta_\gamma(i+1) \leq \chi_\gamma(j+1)$. We add the pair $(i+1, j+1)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

Subcase: $c \in \mathit{In}_{\,env}$
In this case, process P sends a message to the environment.

The transformation changes all outputs to the environment to *skip*-actions. By rule T-OUTPUT$_{ext}$ of Table 12, we obtain $l \rightarrow_{g^\sharp \,\triangleright\, skip} \hat{l} \in \mathit{Edg}^\sharp$. Since

$\zeta_\gamma(i) \leq \chi_\gamma(j)$, $[\![g^\sharp]\!]_{\eta^\sharp}$ is *true* by Lemma 5.13. The output to the environment in P is mimicked by $(l, \eta^\sharp) \rightarrow_\tau (\hat{l}, \eta^\sharp)$ in $P^\sharp$. We define $\chi_\lambda(j+1) = \tau$ and $\chi_\gamma(j+1) = (\hat{l}, \eta^\sharp)$. $\ulcorner\zeta_\lambda(i+1)\urcorner \leq \ulcorner\chi_\lambda(j+1)\urcorner$ holds for labels. Since $\zeta_\gamma(i) \leq \chi_\gamma(j)$ and the output step of the original system and the mimicking step of the closed one change only the location, $\zeta_\gamma(i+1) \leq \chi_\gamma(j+1)$. We add the pair $(i+1, j+1)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

Case: ASSIGN
Assume that the $(i+1)$th-step of ζ is $(l, \eta) \rightarrow_\tau (\hat{l}, \eta[x \mapsto v])$, i.e. $\zeta_\gamma(i) = (l, \eta)$, $\zeta_\lambda(i+1) = \tau$, and $\zeta_\gamma(i+1) = (\hat{l}, \eta[x \mapsto v])$. By rule ASSIGN of Table 9, we get $l \longrightarrow_{g \,\triangleright\, x:=e} \hat{l} \in Edg$. Here we should consider three cases: (i) expression e is guaranteed to be influenced by the environment and the value of x is not used until it gets a reliable value, i.e. $x \notin Var_{\mathtt{I}}$; (ii) expression e is guaranteed to be influenced by the environment and variable x can be present in expressions or guards that should be treated dynamically, i.e. $x \in Var_{\mathtt{I}}$; (iii) otherwise.

Subcase: $[\![e]\!]_{\eta_l^\alpha} = \top$ and $x \notin Var_{\mathtt{I}}$
By rule T-ASSIGN$_1$ of Table 12, we obtain $l \longrightarrow_{g^\sharp \,\triangleright\, skip} \hat{l} \in Edg^\sharp$. $[\![g^\sharp]\!]_{\eta^\sharp}$ is *true* by Lemma 5.13, because $\zeta_\gamma(i) \leq \chi_\gamma(j)$. Therefore the assignment in P can be mimicked by $(l, \eta^\sharp) \rightarrow_\tau (\hat{l}, \eta^\sharp)$ in $P^\sharp$. We define $\chi_\lambda(j+1) = \tau$ and $\chi_\gamma(j+1) = (\hat{l}, \eta^\sharp)$. $\ulcorner\zeta_\lambda(i+1)\urcorner \leq \ulcorner\chi_\lambda(j+1)\urcorner$ holds for labels. Since $x \notin Var_{\mathtt{I}}$, $[\![e]\!]_{\eta_{\hat{l}}^\alpha} = \top$ and $\zeta_\gamma(i) \leq \chi_\gamma(j)$, we get $\zeta_\gamma(i+1) \leq \chi_\gamma(j+1)$. We add the pair $(i+1, j+1)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

Subcase: $[\![e]\!]_{\eta_l^\alpha} = \top$ and $x \in Var_{\mathtt{I}}$
By rule T-ASSIGN$_{12}$ of Table 12, we obtain $l \longrightarrow_{g^\sharp \,\triangleright\, x:=\mathbb{T}} \hat{l} \in Edg^\sharp$. Since $\zeta_\gamma(i) \leq \chi_\gamma(j)$, $[\![g^\sharp]\!]_{\eta^\sharp}$ is *true* by Lemma 5.13. By rule ASSIGN of Table 9, the assignment in P can be mimicked by the assignment $(l, \eta^\sharp) \rightarrow_\tau (\hat{l}, \eta^\sharp[x \mapsto \mathbb{T}])$ in $P^\sharp$. We define $\chi_\lambda(j+1) = \tau$ and $\chi_\gamma(j+1) = (\hat{l}, \eta^\sharp[x \mapsto \mathbb{T}])$. Moreover, $\ulcorner\zeta_\lambda(i+1)\urcorner \leq \ulcorner\chi_\lambda(j+1)\urcorner$ holds for labels.

The assignment step of the original system changes the value of x to v and the mimicking step of the closed system changes the value of x to $\mathbb{T}$. Since $\zeta_\gamma(i) \leq \chi_\gamma(j)$ and $v \leq \mathbb{T}$, $\zeta_\gamma(i+1) \leq \chi_\gamma(j+1)$. We add the pair $(i+1, j+1)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

Subcase: $[\![e]\!]_{\eta_l^\alpha} \neq \top$
If $[\![e]\!]_{\eta_l^\alpha} \neq \top$, only the guard of the assignment is modified by the transformation, i.e. $l \longrightarrow_{g^\sharp \,\triangleright\, x:=e} \hat{l} \in Edg^\sharp$. Since $\zeta_\gamma(i) \leq \chi_\gamma(j)$, $[\![g^\sharp]\!]_{\eta^\sharp}$ is *true* by Lemma 5.13. By Lemma 5.11, $[\![e]\!]_\eta \leq [\![e]\!]_{\eta^\sharp}$ and $v^{\mathbb{T}} = [\![e]\!]_{\eta^\sharp}$. By rule ASSIGN of Table 9, the assignment in P can be mimicked by $(l, \eta^\sharp) \rightarrow_\tau (\hat{l}, \eta^\sharp[x \mapsto v^{\top}])$ in $P^\sharp$. We define $\chi_\lambda(j+1) = \tau$ and $\chi_\gamma(j+1) = (\hat{l}, \eta^\sharp[x \mapsto v^{\top}])$. Since $\zeta_\gamma(i) \leq \chi_\gamma(j)$ and $[\![e]\!]_\eta \leq [\![e]\!]_{\eta^\sharp}$, $\zeta_\gamma(i+1) \leq \chi_\gamma(j+1)$. We add the pair $(i+1, j+1)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

Case: SET
Assume that the $(i+1)$th-step of ζ is $(l, \eta) \rightarrow_\tau (\hat{l}, \eta[t \mapsto on(v)])$, i.e. $\zeta_\gamma(i) = (l, \eta)$, $\zeta_\lambda(i+1) = \tau$, and $\zeta_\gamma(i+1) = (\hat{l}, \eta[t \mapsto on(v)])$. By rule SET of Table 9, we

have $l \longrightarrow_{g \,\triangleright\, set\ t:=e} \hat{l} \in \mathit{Edg}$. Here we consider two sub-cases: (i) expression e is guaranteed to be influenced by chaos; and (ii) it is not influenced by the environment or it depends on a system run.

Subcase: $[\![e]\!]_{\eta_l^\alpha} = \top$

By rule T-SET of Table 12, $l \longrightarrow_{g^\sharp \,\triangleright\, set\ t:=\mathbb{T}} \hat{l} \in \mathit{Edg}^\sharp$. Since $\zeta_\gamma(i) \le \chi_\gamma(j)$, $[\![g^\sharp]\!]_{\eta^\sharp}$ is *true* by Lemma 5.13. By rule SET of Table 9, setting timer t in P can be mimicked by setting $(l, \eta^\sharp) \rightarrow_\tau (\hat{l}, \eta[t \mapsto on(\mathbb{T})])$ timer t in $P^\sharp$. We define $\chi_\lambda(j+1) = \tau$ and $\chi_\gamma(j+1) = (\hat{l}, \eta^\sharp[t \mapsto on(\mathbb{T})])$. $\ulcorner\zeta_\lambda(i+1)\urcorner \le \ulcorner\chi_\lambda(j+1)\urcorner$ holds for labels. Since $\zeta_\gamma(i) \le \chi_\gamma(j)$ and $on(v) \le on(\mathbb{T})$, $\zeta_\gamma(i+1) \le \chi_\gamma(j+1)$. We add the pair $(i+1, j+1)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

Subcase: $[\![e]\!]_{\eta_l^\alpha} \neq \top$

The edge of the original specification remains unchanged, only the guard is transformed, i.e. $l \longrightarrow_{g^\sharp \,\triangleright\, set\ t:=e} \hat{l} \in \mathit{Edg}^\sharp$. Since $\zeta_\gamma(i) \le \chi_\gamma(j)$, $[\![g^\sharp]\!]_{\eta^\sharp}$ is *true* by Lemma 5.13. By Lemma 5.11, $[\![e]\!]_\eta \le [\![e]\!]_{\eta^\sharp}$ and $v^\top = [\![e]\!]_{\eta^\sharp}$. By rule SET of Table 12, setting timer t in P can be mimicked by $(l, \eta^\sharp) \rightarrow_\tau (\hat{l}, \eta[t \mapsto on(v^\top)])$ setting timer t in $P^\sharp$. We define $\chi_\lambda(j+1) = \tau$ and $\chi_\gamma(j+1) = (\hat{l}, \eta^\sharp[t \mapsto on(v^\top)])$. $\ulcorner\zeta_\lambda(i+1)\urcorner \le \ulcorner\chi_\lambda(j+1)\urcorner$ holds for the labels. $\zeta_\gamma(i+1) \le \chi_\gamma(j+1)$, because $\zeta_\gamma(i) \le \chi_\gamma(j)$ and $on(v) \le on(v^\top)$. $\zeta_\gamma(i+1) \le \chi_\gamma(j+1)$. We add the pair $(i+1, j+1)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

Case: RESET

The transformation does not modify reset edges. The proof that a reset of timer t in P can be mimicked by a reset of t in $P^\sharp$ is straightforward.

Case: TIMEOUT

Assume that the $(i+1)$th-step of ζ is the timeout step $(l, \eta) \rightarrow_\tau (\hat{l}, \eta[t \mapsto \mathit{off}])$, i.e. $\zeta_\gamma(i) = (l, \eta)$, $\zeta_\lambda(i+1) = \tau$, and $\zeta_\gamma(i+1) = (\hat{l}, \eta[t \mapsto \mathit{off}])$. By rule TIMEOUT of Table 9, we have $l \longrightarrow_{g_t \,\triangleright\, reset\ t} \hat{l} \in \mathit{Edg}$.

The timeout guard $t = on(0)$ is modified by the transformation into the guard $((t = on(0)) \vee (t = on(\mathbb{T})))$. Since $\zeta_\gamma(i) \le \chi_\gamma(j)$ and $t = on(0)$ is *true* in $\zeta_\gamma(i)$, $((t = on(0)) \vee (t = on(\mathbb{T})))$ is *true* in $\chi_\gamma(j)$ by Lemma 5.13. By rule TIMEOUT of Table 9, the timeout in P can be mimicked by the timeout $(l, \eta^\sharp) \rightarrow_\tau (\hat{l}, \eta^\sharp[t \mapsto \mathit{off}])$ in $P^\sharp$. We define $\chi_\lambda(j+1) = \tau$ and $\chi_\gamma(j+1) = (\hat{l}, \eta^\sharp[t \mapsto \mathit{off}])$. $\ulcorner\zeta_\lambda(i+1)\urcorner \le \ulcorner\chi_\lambda(j+1)\urcorner$ holds for labels. Both the timeout τ-step of the original system and the mimicking step of the closed one set the timer t to *off*. Since $\zeta_\gamma(i) \le \chi_\gamma(j)$, $\zeta_\gamma(i+1) \le \chi_\gamma(j+1)$. We add the pair $(i+1, j+1)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

Case: TDISCARD

Analogous to the TIMEOUT case above.

Case: TICK$_P$

Assume that the $(i+1)$th-step of ζ is the *tick* step, $(l, \eta) \rightarrow_{tick} (l, \eta[t \mapsto (t-1)])$, i.e. $\zeta_\gamma(i) = (l, \eta)$, $\zeta_\lambda(i+1) = \tau$, and $\zeta_\gamma(i+1) = (l, \eta[t \mapsto (t-1)])$. By rule TICK$_P$ of Table 9, we have $\mathit{blocked}(l, \eta)$.

We postpone setting timer t_P to $on(1)$ in $P^\sharp$ until we meet a *tick*-step along ζ, hence, $P^\sharp$ is not blocked at $\chi_\gamma(j)$. By Lemma 5.15, we add a sequence

$\chi_\gamma(j) \rightarrow_{\chi_\lambda(j+1)} \cdots \rightarrow_{\chi_\lambda(j+m)} \chi_\gamma(j+m)$ such that $\zeta_\gamma(i) \leq \chi_\gamma(j+k)$ and $\chi_\lambda(j+k) = \tau$ for all $k = 1..m$. We also add $(i, j+k)$ to $\mathcal{R}$ for all $k = 1..m$. At $\chi_\gamma(j+m)$, $P^\sharp$ is blocked, hence it can take the mimicking *tick*-step $\chi_\gamma(j+m) \rightarrow_{tick} \chi_\gamma(j+m)_{[t \mapsto (t-1)]}$. We define $\chi_\lambda(j+m+1) = tick$ and $\chi_\gamma(j+m+1) = \chi_\gamma(j+m)_{[t \mapsto (t-1)]}$. By Lemma 5.12, $\zeta_\gamma(i+1) \leq \chi_\gamma(j+m+1)$. We add $(i+1, j+m+1)$ to $\mathcal{R}$. The *tick*-step of P is mimicked by the sequence of τ-steps followed by the *tick*-step in $P^\sharp$. The conditions of Def. 5.13 are satisfied.
□

Lemma 5.17.
Let Spec be a specification, and Spec$^\sharp$ be the specification obtained by transforming Spec according to the rules of Table 12. Let S and $S^\sharp$ be LTSs built by applying the rules of Table 9, rules In *and* Out *of Table 2 and the rules of Table 6 to Spec and Spec$^\sharp$, respectively. Let γ and $\gamma^\sharp$ be configurations of S and $S^\sharp$, such that $\gamma \leq \gamma^\sharp$. If blocked($\gamma$), then $\gamma^\sharp = \gamma_0^\sharp \rightarrow_\tau \gamma_1^\sharp \rightarrow_\tau \cdots \rightarrow_\tau \gamma_n^\sharp$ for some configurations $\gamma_i^\sharp$ and some $n \geq 0$ such that $\gamma \leq \gamma_i^\sharp$ for all i, and blocked($\gamma_n^\sharp$).*

Proof. Straightforward from Lemma 5.15 and Def. 5.10. □

Lemma 5.18.
Let Spec be a specification, and Spec$^\sharp$ be the process specification obtained by transforming Spec according to the rules of Table 12. Let S and $S^\sharp$ be LTSs built by applying the rules of Table 9, rules In *and* Out *of Table 2 and the rules of Table 6 to Spec and Spec$^\sharp$, respectively. Then for each trace ζ of S there is a trace χ of $S^\sharp$ such that $\zeta \equiv_\mathcal{R} \chi$ for some $\mathcal{R} \subseteq \mathbb{N} \times \mathbb{N}$.*

Proof. Here, we show that for any trace ζ of S we can build a trace χ of $S^\sharp$ such that $\zeta \equiv_\mathcal{R} \chi$ for some $\mathcal{R}$. Initially, all queues modelling channels in S and in $S^\sharp$ are empty, all variables have the same initial values and all the timers are deactivated, hence $\gamma_0 \leq \gamma_0^\sharp$ for initial configurations γ_0 and $\gamma_0^\sharp$ of S and $S^\sharp$, respectively. $\mathcal{R}$ is initialized as $\{(0,0)\}$. Further, we proceed by induction on the length of ζ. Each step of S is mimicked by a step of $S^\sharp$ so that conditions of Def. 5.13 are satisfied, and configurations reached by the original step and the mimicking step are related by $\leq$.

Assume that $(i,j) \in \mathcal{R}$. Before we proceed with a case analysis on rules In, Out of Table 2 and the rules of Table 6, we consider channels of the open system and channels of the closed one. The queues modelling channels in closed system $S^\sharp$ do not contain messages sent to or received from the environment.

Case: In
Assume that a channel c has state (c,q) in S and state $(c,q^\sharp)$ in $S^\sharp$, and that $(c,q) \leq (c,q^\sharp)$. Let c in S takes $(c,q) \rightarrow_{c_o?s(pid,v)} (c, q :: s(pid,v))$ (rule In of Table 2), i.e. some process within the system sends a message $s(pid,v)$ via c to another process. Channel c in $S^\sharp$ can mimic step $c_o?s(pid,v)$ by a step $(c,q) \rightarrow_{c_o?s(pid,v^\top)} (c, q :: s(pid,v^\top))$, where $v \leq v^\top$. The channel state reached by the original and the channel state reached by the mimicking step

are in relation $\leq$, i.e. $(c, q :: s(pid, v)) \leq (c, q :: s(pid, v^{\top}))$ (cf. Def. 5.8) and $\ulcorner c_o?s(pid, v)\urcorner \leq \ulcorner c_o?s(pid, v^{\top})\urcorner$ and condition 2 of Def. 5.13 is satisfied.

Assume that c in S makes a step $(c, q) \rightarrow_{c_o?s(pid,v)} (c, q :: s(pid, v))$, i.e. some process within the system sends a message $s(pid, v)$ via c to the environment. In this case, $q^\sharp = \epsilon$ (cf. Def. 5.9). Since $(c, q) \leq (c, \epsilon)$ and $\ulcorner c_o?s(env, v)\urcorner = \tau$, $(c, q :: s(pid, v)) \leq (c, \epsilon)$ and condition 1 of Def. 5.13 is satisfied.

Assume that c in S makes a step $(c, q) \rightarrow_{c_o?s(env,v)} (c, q :: s(pid, v))$, i.e. the environment sends a message $s(env, v)$ to a process within the system. Since $(c, q) \leq (c, q^\sharp)$ and $\ulcorner c_o?s(env, v)\urcorner = \tau$, $(c, q :: s(pid, v)) \leq (c, q^\sharp)$ and condition 1 of Def. 5.13 is satisfied.

Case: Out
Analogous to the In case.

Case: Comm
Let $(\ldots, \sigma_k, \ldots, \sigma_l, \ldots) \rightarrow_\tau (\ldots, \hat{\sigma}_k, \ldots, \hat{\sigma}_l, \ldots)$ be the $(i + 1)$th-step of ζ. It means that $\zeta_\gamma(i) = (\ldots, \sigma_k, \ldots, \sigma_l, \ldots)$, $\zeta_\lambda(i + 1) = \tau$ and $\zeta_\gamma(i + 1) = (\ldots, \hat{\sigma}_k, \ldots, \hat{\sigma}_l, \ldots)$. Further we consider four subcases: (i) receiving a message sent by a process of the system; (ii) receiving a message sent by the environment; (iii) sending a message to a process within the system; (iv) sending a message to the environment.

Subcase: (i)
By rule Comm of Table 6, we have $\sigma_k \rightarrow_{c_i?s(pid,v)} \hat{\sigma}_k$ for some process P_k and $\sigma_l \rightarrow_{c_i!s(pid,v)} \hat{\sigma}_l$ for some channel c_l.

Since $\zeta_\gamma(i) \leq \chi_\gamma(j)$, process $P_k^\sharp$ can take $\sigma_k^\sharp \rightarrow_{c_i?s(pid,v^\top)} \hat{\sigma}_k^\sharp$ mimicking input (see case Input of Lemma 5.16). Moreover, channel c_l in $S^\sharp$ can mimic the $c_i!s(pid, v)$-step by $\sigma_l^\sharp \rightarrow_{c_i!s(pid,v^\top)} \hat{\sigma}_l^\sharp$. By rule Comm of Table 6, the τ-step of S can be mimicked by $(\ldots, \sigma_k^\sharp, \ldots, \sigma_l^\sharp, \ldots) \rightarrow_\tau (\ldots, \hat{\sigma}_k^\sharp, \ldots, \hat{\sigma}_l^\sharp, \ldots)$ in $S^\sharp$.

In this case, we define $\chi_\lambda(j + 1) = \tau$ and $\chi_\gamma(j + 1) = (\ldots, \hat{\sigma}_k^\sharp, \ldots, \hat{\sigma}_l^\sharp, \ldots)$. According to Lemma 5.16, $\zeta_\gamma(i + 1) \leq \chi_\gamma(j + 1)$. We add $(i + 1, j + 1)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

Subcase: (ii)
By rule Comm of Table 6, we have $\sigma_k \rightarrow_{c_i?s(env,v)} \hat{\sigma}_k$ for some process P_k and $\sigma_l \rightarrow_{c_i!s(env,v)} \hat{\sigma}_l$ for some channel c_l.

Since $\zeta_\gamma(i) \leq \chi_\gamma(j)$, process $P_k^\sharp$ can do a mimicking input $\sigma_k^\sharp \rightarrow_\tau \hat{\sigma}_k^\sharp$ (cf. the case Input of Lemma 5.16). By rule Interleave$_\tau$ of Table 6, the τ-step of S can be mimicked by $(\ldots, \sigma_k^\sharp, \ldots, \sigma_l^\sharp, \ldots) \rightarrow_\tau (\ldots, \hat{\sigma}_k^\sharp, \ldots, \sigma_l^\sharp, \ldots)$ in $S^\sharp$. In this case, we define $\chi_\lambda(j + 1) = \tau$ and $\chi_\gamma(j + 1) = (\ldots, \hat{\sigma}_k^\sharp, \ldots, \sigma_l^\sharp, \ldots)$. According to Lemma 5.16, $\zeta_\gamma(i+1) \leq \chi_\gamma(j+1)$. We add $(i+1, j+1)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

Subcase: (iii)
By rule Comm of Table 6, we have $\sigma_k \rightarrow_{c_o?s(pid,v)} \hat{\sigma}_k$ for some channel c_k and $\sigma_l \rightarrow_{c_o!s(pid,v)} \hat{\sigma}_l$ for some process P_l. Moreover, the message is sent to another process within the system, i.e. $c \not\in in_{env}$.

Since $\zeta_\gamma(i) \leq \chi_\gamma(j)$, process $P_l^\sharp$ can take $\sigma_l^\sharp \to_{c_o!s(pid,v^\top)} \hat{\sigma}_l^\sharp$ (cf. the case Output of Lemma 5.16). Channel c_k in $S^\sharp$ can mimic the $c_o?s(pid,v)$-step by $\sigma_k^\sharp \to_{c_o?s(pid,v^\top)} \hat{\sigma}_k^\sharp$. By rule Comm of Table 6, the τ-step of S can be mimicked by $(\ldots,\, \sigma_k^\sharp,\, \ldots,\, \sigma_l^\sharp,\, \ldots) \to_\tau (\ldots,\, \hat{\sigma}_k^\sharp,\, \ldots,\, \hat{\sigma}_l^\sharp,\, \ldots)$ in $S^\sharp$.

We define $\chi_\lambda(j+1) = \tau$ and $\chi_\gamma(j+1) = (\ldots, \hat{\sigma}_k^\sharp, \ldots, \hat{\sigma}_l^\sharp, \ldots)$. According to Lemma 5.16, $\zeta_\gamma(i+1) \leq \chi_\gamma(j+1)$. We add $(i+1, j+1)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

Subcase: (iv)
By rule Comm of Table 6, we have $\sigma_k \to_{c_o?s(pid,v)} \hat{\sigma}_k$ for some channel c_k and $\sigma_l \to_{c_o!s(pid,v)} \hat{\sigma}_l$ for some process P_l. Moreover, the message is sent to the environment, i.e. $c \in in_{env}$.

Since $\zeta_\gamma(i) \leq \chi_\gamma(j)$, process $P_l^\sharp$ can do a mimicking τ-step $\sigma_l^\sharp \to_\tau \hat{\sigma}_l^\sharp$ (see case Output of Lemma 5.16).

By rule Interleave$_\tau$ of Table 6, the τ-step of S can be mimicked by transition $(\ldots,\, \sigma_k^\sharp,\, \ldots,\, \sigma_l^\sharp,\, \ldots) \to_\tau (\ldots,\, \sigma_k^\sharp,\, \ldots,\, \hat{\sigma}_l^\sharp,\, \ldots)$ in $S^\sharp$.

In this case, we define $\chi_\lambda(j+1) = \tau$ and $\chi_\gamma(j+1) = (\ldots, \sigma_k^\sharp, \ldots, \hat{\sigma}_l^\sharp, \ldots)$. According to Lemma 5.16, $\zeta_\gamma(i+1) \leq \chi_\gamma(j+1)$. We add $(i+1, j+1)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

Case: Interleave$_{in}$
Assume that $(\ldots, \sigma_k, \ldots) \to_{c_o?s(env,v)} (\ldots, \hat{\sigma}_k, \ldots)$ is the $(i+1)$th-step of ζ, i.e. $\zeta_\gamma(i) = (\ldots, \sigma_k, \ldots)$, $\zeta_\lambda(i+1) = c_o?s(env,v)$ and $\zeta_\gamma(i+1) = (\ldots, \hat{\sigma}_k, \ldots)$.

This case corresponds to condition 1 of Def. 5.13, i.e. there is no step in $S^\sharp$ mimicking the $(i+1)$th-step of ζ. But the configuration $\zeta_\gamma(i+1)$ reached by the step is in relation $\leq$ with $\chi_\gamma(j)$ and $\ulcorner c_o?s(env,v)\urcorner$ is τ (see case In above). We add $(i+1, j)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

Case: Interleave$_{out}$
Assume that $(\ldots, \sigma_k, \ldots) \to_{c_i!s(pid,v)} (\ldots, \hat{\sigma}_k, \ldots)$ is the $(i+1)$th-step of ζ, i.e. $\zeta_\gamma(i) = (\ldots, \sigma_k, \ldots)$, $\zeta_\lambda(i+1) = c_i!s(pid,v)$ and $\zeta_\gamma(i+1) = (\ldots, \hat{\sigma}_k, \ldots)$. Moreover $c \in in_{env}$.

This case corresponds to condition 1 of Def. 5.13, i.e. there is no step in $S^\sharp$ mimicking the $(i+1)$th-step of ζ. But the configuration $\zeta_\gamma(i+1)$ reached by the step is in relation $\leq$ with $\chi_\gamma(j)$ and $\ulcorner c_i!s(pid,v)\urcorner$ is τ (see case Out above). We add $(i+1, j)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

Case: Interleave$_\tau$
Assume that $(\ldots, \sigma_k, \ldots) \to_\tau (\ldots, \hat{\sigma}_k, \ldots)$ is the $(i+1)$th-step of ζ, i.e. $\zeta_\gamma(i) = (\ldots, \sigma_k, \ldots)$, $\zeta_\lambda(i+1) = \tau$ and $\zeta_\gamma(i+1) = (\ldots, \hat{\sigma}_k, \ldots)$. Moreover, we assume that τ is not the result of synchronizing communication steps, i.e. it corresponds to the Assign, Set, Timeout, TDiscard, or Reset case considered in Lemma 5.16.

By Lemma 5.16 and rule Interleave$_\tau$ of Table 6, the τ-step of S can be mimicked by τ-step $(\ldots,\, \sigma_k^\sharp,\, \ldots) \to_\tau (\ldots,\, \hat{\sigma}_k^\sharp,\, \ldots)$ of $S^\sharp$. We define $\chi_\gamma(j+1) = (\ldots, \hat{\sigma}_k^\sharp, \ldots)$, $\chi_\lambda(j+1) = \tau$ and add $(i+1, j+1)$ to $\mathcal{R}$. The conditions of Def. 5.13 are satisfied.

Case: Tick
Assume that $(\sigma_1, \ldots, \sigma_n) \rightarrow_{tick} (\hat{\sigma}_1, \ldots, \hat{\sigma}_n)$ is the $(i+1)$th-step of ζ, i.e. $\zeta_\gamma(i) = (\sigma_1, \ldots, \sigma_n)$, $\zeta_\lambda(i+1) = \mathit{tick}$ and $\zeta_\gamma(i+1) = (\hat{\sigma}_1, \ldots, \hat{\sigma}_n)$. By rule Tick of Table 6, *blocked*$(\zeta_\gamma(i))$.

By Lemma 5.17, there is $\chi_\gamma(j) = \gamma_0^\sharp \rightarrow_\tau \gamma_1^\sharp \rightarrow_\tau \ldots \rightarrow_\tau \gamma_m^\sharp$ for some configurations $\gamma_i^\sharp$ of $S^\sharp$ and some $n \geq 0$ such that $\gamma \leq \gamma_i^\sharp$ for all i, and *blocked*$(\gamma_n^\sharp)$. Therefore, we extend χ by a sequence $\chi_\gamma(j) \rightarrow_{\chi_\lambda(j+1)} \cdots \rightarrow_{\chi_\lambda(j+m)} \chi_\gamma(j+m)$ such that $\zeta_\gamma(i) \leq \chi_\gamma(j+k)$, $\chi_\lambda(j+k) = \tau$ for all $k = 1..m$. We add $(i, j+k)$ to $\mathcal{R}$ for all $k = 1..m$. At $\chi_\gamma(j+m)$, $S^\sharp$ is blocked, hence it may take the step $\chi_\gamma(j+m) \rightarrow_{tick} \chi_\gamma(j+m)_{[t \mapsto (t-1)]}$. We define $\chi_\lambda(j+m+1) = \mathit{tick}$ and $\chi_\gamma(j+m+1) = \chi_\gamma(j+m)_{[t \mapsto (t-1)]}$ for all $t \in \mathit{Var}$. According to Lemma 5.12, $\zeta_\gamma(i+1) \leq \chi_\gamma(j+m+1)$. We add $(i+1, j+m+1)$ to $\mathcal{R}$. The *tick*-step of S is mimicked by the sequence of τ-steps followed by *tick*-step in $S^\sharp$. The conditions of Def. 5.13 are satisfied.

Here we showed that for each finite prefix $\zeta^{(i+1)}$ of trace ζ of S, we can construct a finite prefix $\chi^{(j+m)}$ of trace χ of $S^\sharp$ such that $\zeta^{(i+1)} \equiv_\mathcal{R} \chi^{(j+m)}$ for some $\mathcal{R}$. It means that for each trace ζ of S there is a trace χ of $S^\sharp$ such that $\zeta \equiv_\mathcal{R} \chi$ for some $\mathcal{R}$, i.e. $S \preceq_\mathcal{R} S^\sharp$. □

Lemma 5.19. Path inclusion up to stuttering
Let Spec be a specification, and Spec$^\sharp$ be the specification obtained by transforming Spec according to the rules of Table 12. Let S and $S^\sharp$ be LTSs built by applying the rules of Table 9, rules In *and* Out *of Table 2 and the rules of Table 6 to Spec and Spec$^\sharp$, respectively. Let $\mathcal{L}\colon \Gamma \to 2^\mathcal{P}$ and $\mathcal{L}^\sharp\colon \Gamma^\sharp \to 2^\mathcal{P}$ be interpretation functions, $\mathcal{P}$ be the set of atomic propositions that mention only variables x (process and timer) such that $[\![x]\!]_{\eta_l^\alpha} = \bot$ for all $l \in Loc$, and Γ and $\Gamma^\sharp$ be sets of configurations of S and $S^\sharp$ respectively. Then $(S, \mathcal{L}) \preceq_{st} (S^\sharp, \mathcal{L}^\sharp)$.*

Proof. Follows directly by Lemma 5.18 and Lemma 5.14. □

Theorem 5.2.
For all formulas φ from next-free LTL mentioning only variables x (process and timer) such that $[\![x]\!]_{\eta_l^\alpha} = \bot$ for all $l \in Loc$, $S \models \varphi$ if $S^\sharp \models \varphi$.

Proof. Straightforward from Lemma 5.19 and Theorem 2.2. □

5.5 Implementation

5.5.1 Extending the Vires Toolset

The Vires toolset (see [167]) was introduced for the verification of industrial-size communication protocols. Its architecture is targeted towards the verification of SDL specifications and it provides an automatic translation of SDL-code into the input language of a discrete-time extension of the *Spin* model-checker. Design, analysis, verification, and validation of SDL specifications is supported by OBJECTGEODE, one of the most advanced integrated SDL-environments. OBJECTGEODE also provides code generation and testing of real-time and distributed applications.

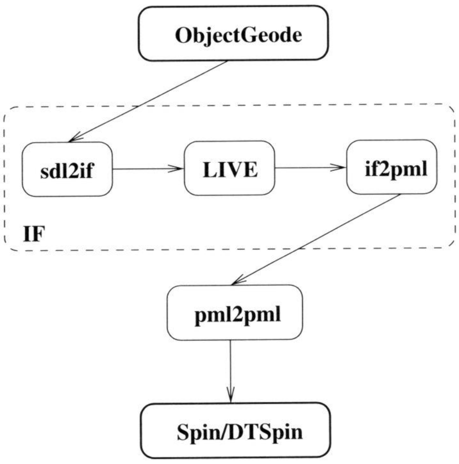

Fig. 21. Toolset components

IF [28] bridges the gap between OBJECTGEODE and *Spin/DTSpin* (cf. Sec. 2.4). It contains a translator, SDL2IF of SDL specifications into the intermediate representation IF. A static analyzer Live [27] performs an optimization of IF-representation to reduce the state space of the model. IF-specifications can be translated to DTPROMELA models with the help of IF2PML-translator [25] and verified by *DTSpin*.

We have developed the PML2PML-translator that takes care of the automatic closing of a subcomponent and implements the theory presented before. The tool post-processes the output from the translation of an SDL-specification to PROMELA, where the implementation covers the subset of SDL described abstractly in Section 5.2. The translator works fully automatic and does not require any user interaction, except that the user is required to indicate the list of external signals. The extension is implemented in Java and requires JDK-1.2 or later. The package can be downloaded using the following URL: `http://www.cwi.nl/~ustin/EH.html`.

5.5.2 Implementation of the Program Transformation

To keep the implementation in *Spin*'s input-language PROMELA simple, the abstraction introduced in Sec. 5.3 is realized as a straightforward source code transformation. Instead of extending the data domains by one single additional

$$\frac{l \longrightarrow_{c?s(x)} \hat{l} \in Edg \qquad s \in Sig_{ext} \qquad x \in Var_{\mathtt{I}}}{l \longrightarrow_{g_{t_P} \rhd\, set\ t_P:=0} \longrightarrow_{(true) \rhd\, b_x:=false}\ \hat{l} \in Edg^{i}}\ \textsc{T-Input}^{1}_{ext}$$

$$\frac{l \longrightarrow_{c?s(x)} \hat{l} \in Edg \qquad s \in Sig_{ext} \qquad x \notin Var_{\mathtt{I}}}{l \longrightarrow_{g_{t_P} \rhd\, set\ t_P:=0} \hat{l} \in Edg^{i}}\ \textsc{T-Input}^{2}_{ext}$$

$$\frac{l \in Loc_i}{l \longrightarrow_{g_{t_P} \rhd\, set\ t_P:=1} l \in Edg^{i}}\ \textsc{NoInput}$$

$$\frac{l \longrightarrow_{c?s(x)} \hat{l} \in Edg \qquad s \in Sig_{int} \qquad x \in Var_{\mathtt{I}}}{l \longrightarrow_{c?s(x,b_x)} \hat{l} \in Edg^{i}}\ \textsc{T-Input}^{1}_{int}$$

$$\frac{l \longrightarrow_{c?s(x)} \hat{l} \in Edg \qquad s \in Sig_{int} \qquad x \notin Var_{\mathtt{I}}}{l \longrightarrow_{c?s(x,\_)} \hat{l} \in Edg^{i}}\ \textsc{T-Input}^{2}_{int}$$

$$\frac{l \longrightarrow_{g \rhd\, c!(s,e)} \hat{l} \in Edg \qquad s \in Sig_{ext}}{l \longrightarrow_{g^{i} \rhd\, skip} \hat{l} \in Edg^{i}}\ \textsc{T-Output}_{ext}$$

$$\frac{l \longrightarrow_{g \rhd\, c!(s,e)} \hat{l} \in Edg \qquad c \notin In_{env}}{l \longrightarrow_{(g^{i} \bigwedge b(e)) \rhd\, c!(s,e,true)} \hat{l} \in Edg^{i}}\ \textsc{T-Output}^{1}_{int}$$

$$\frac{l \longrightarrow_{g \rhd\, c!(s,e)} \hat{l} \in Edg \qquad c \notin In_{env}}{l \longrightarrow_{(g^{i} \bigwedge \neg b(e)) \rhd\, c!(s,\_,false)} \hat{l} \in Edg^{i}}\ \textsc{T-Output}^{2}_{int}$$

$$\frac{l \longrightarrow_{g \rhd\, x:=e} \hat{l} \in Edg \qquad x \in Var_{\mathtt{I}}}{l \longrightarrow_{(g^{i} \bigwedge b(e)) \rhd\, x:=e} \longrightarrow_{(true) \rhd\, b_x:=true}\ \hat{l} \in Edg^{i}}\ \textsc{T-Assign}_{11}$$

$$\frac{l \longrightarrow_{g \rhd\, x:=e} \hat{l} \in Edg \qquad x \in Var_{\mathtt{I}}}{l \longrightarrow_{(g^{i} \bigwedge \neg b(e)) \rhd\, b_x:=false} \hat{l} \in Edg^{i}}\ \textsc{T-Assign}_{12}$$

$$\frac{l \longrightarrow_{g \rhd\, x:=e} \hat{l} \in Edg \qquad x \notin Var_{\mathtt{I}}}{l \longrightarrow_{(g^{i} \bigwedge b(e)) \rhd\, x:=e} \hat{l} \in Edg^{i}}\ \textsc{T-Assign}_{21}$$

$$\frac{l \longrightarrow_{g \rhd\, x:=e} \hat{l} \in Edg \qquad x \notin Var_{\mathtt{I}}}{l \longrightarrow_{(g^{i} \bigwedge \neg b(e)) \rhd\, skip} \hat{l} \in Edg^{i}}\ \textsc{T-Assign}_{22}$$

Table 13. Implementation of the transformation for untimed edges

abstract value for external data, each variable x has associated to it a boolean flag b_x to remember whether its current value is from the outside or not: The flag's value is *false* when x contains data from outside, and *true* otherwise. For model checking, memory and time consumption are crucial. The introduction of a boolean flag for each variable is not optimal in that regard. For instance, a system can contain variables never influenced by the environment, hence we need no boolean flags for these variables.

Clearly, the flags are needed only for variables and timers that carry the value ⌶ (or $on(\text{⌶})$ for timers) in at least one location; for other variables the values found with the static analysis can be used. So let $Var_{\text{⌶}}$ be the set of variables and timers that carry the value ⌶, respectively $on(\text{⌶})$, at least once.

The rules of Table 13 define the transformation of untimed edges with respect to the results of the combined *may*/*must* analysis. Boolean flags are introduced only for variables that are in $Var_{\text{⌶}}$. Expressions are interpreted strictly with respect to chaotic data and we write $b(e)$, where $b(e)$ is *true* iff all of the variables from $Var_{\text{⌶}}$ occurring in e have their flags set to *true* and all of the variables not belonging to $Var_{\text{⌶}}$ are valuated to $\bot$ wrt. the analysis, i.e. $b(e) = (\wedge_{i=1}^{n} b_{x_i}) \bigwedge (\wedge_{j=1}^{m} (\eta_l^{\alpha}(y_j) = \bot))$ where $\forall\ i = 1..n$: $x_i \in Var_{\text{⌶}}$ and $\forall\ j = 1..m$: $y_j \notin Var_{\text{⌶}}$. The transformation of the guards is optimized so that guards which contain at least one variable marked $\top$ are transformed to *true*. If all the variables of a guard are marked as $\bot$, the guard is left unchanged.

As the abstract system must show at least all behavior of the original system, actions with guards whose result depends on values coming from outside, i.e. guards g with $b(g) =$ *false*, must be enabled. Therefore we replace each untimed guard by a transformed guard g^i given by the disjunction $\neg b(g) \vee (g \wedge b(g))$. To propagate the information through the system, the parameter lists of signals exchanged within the system, i.e., signals from Sig_{int}, are extended with the lists of corresponding flags.

Inputs from the chaotic environment are always enabled. We must make sure, however, that inputs from the environment do not prevent time progress. Therefore, as in Sec. 5.4, we add a new timer variable t_P for each process, used to guard inputs from outside and assure time progress (cf. T-INPUT$^1_{ext}$, T-INPUT$^2_{ext}$). This timer is set to 0 until a T-NOINPUT step is taken nondeterministically, which sets the timer to 1, thereby postponing the possibility of taking the next input from the environment until time progresses. Flags of variables which received their values from environment signals are set to *false* to indicate that from this point on the value is not reliable any more (cf. T-INPUT$^1_{ext}$). For internal signals, we differentiate two cases: (i) a variable changed by the input has a flag; (ii) the variable has no flag. In the first case, input is extended by the flag of the variable that shows whether a chaotic or non-chaotic value is transferred (rule T-INPUT$^1_{int}$). In the second case, this value does not matter (rule T-INPUT$^2_{int}$).

Outputs to the environment are just removed (cf. rule T-OUTPUT$_{ext}$). Internal outputs are extended as follows: In case the expression e carries a non-chaotic value, this value is transferred together with the flag *true* showing that

$$\frac{l \longrightarrow_{g \,\triangleright\, set\ t:=e} \hat{l} \in Edg \qquad t \in Var_{\mathtt{I}}}{l \longrightarrow_{(g^i \bigwedge b(e)) \,\triangleright\, set\ t:=e} \longrightarrow_{(true) \,\triangleright\, b_t:=true} \hat{l} \in Edg^i}\ \textsc{T-Set}_{11}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, set\ t:=e} \hat{l} \in Edg \qquad t \in Var_{\mathtt{I}}}{l \longrightarrow_{(g^i \bigwedge \neg b(e)) \,\triangleright\, set\ t:=0} \longrightarrow_{(true) \,\triangleright\, b_t:=false} \hat{l} \in Edg^i}\ \textsc{T-Set}_{12}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, set\ t:=e} \hat{l} \in Edg \qquad t \notin Var_{\mathtt{I}}}{l \longrightarrow_{(g^i \bigwedge b(e)) \,\triangleright\, set\ t:=e} \hat{l} \in Edg^i}\ \textsc{T-Set}_{21}$$

$$\frac{l \longrightarrow_{g \,\triangleright\, set\ t:=e} \hat{l} \in Edg \qquad t \notin Var_{\mathtt{I}}}{l \longrightarrow_{(g^i \bigwedge \neg b(e)) \,\triangleright\, set\ t:=0} \hat{l} \in Edg^i}\ \textsc{T-Set}_{22}$$

$$\frac{}{l \longrightarrow_{(g_t \bigwedge \neg b(t)) \,\triangleright\, reset\ t} l \in Edg^i}\ \textsc{T-NoTimeout}$$

$$\frac{l \longrightarrow_{g_t \,\triangleright\, reset\ t} \hat{l} \in Edg \qquad t \in Var_i}{l \longrightarrow_{g_t \,\triangleright\, reset\ t} \longrightarrow_{(true) \,\triangleright\, b_t:=true} \hat{l} \in Edg^i}\ \textsc{T-Timeout}_1$$

$$\frac{l \longrightarrow_{g_t \,\triangleright\, reset\ t} \hat{l} \in Edg \qquad t \notin Var_i}{l \longrightarrow_{g_t \,\triangleright\, reset\ t} \hat{l} \in Edg^i}\ \textsc{T-Timeout}_2$$

$$\frac{l \longrightarrow_{g \,\triangleright\, reset\ t} \hat{l} \in Edg \qquad t \in Var_{\mathtt{I}}}{l \longrightarrow_{g^i \,\triangleright\, reset\ t} \longrightarrow_{b_t:=true} \hat{l} \in Edg^i}\ \textsc{T-Reset}_1$$

$$\frac{l \longrightarrow_{g \,\triangleright\, reset\ t} \hat{l} \in Edg \qquad t \notin Var_{\mathtt{I}}}{l \longrightarrow_{g^i \,\triangleright\, reset\ t} \hat{l} \in Edg^i}\ \textsc{T-Reset}_2$$

Table 14. Implementation of the transformation for timed edges

it is a reliable value (rule T-OUTPUT$^1_{int}$). Otherwise, the same signal is sent parameterized with a default value and the flag *false* demonstrating that a chaotic value is transferred (rule T-OUTPUT$^2_{int}$).

Assignments are treated similarly to outputs. Assignments of chaotic values are skipped (rules T-ASSIGN$_{12}$, T-ASSIGN$_{22}$). The flag of the left side variable is set to *false* in case the variable belongs to $Var_{\mathcal{I}}$ (rule T-ASSIGN$_{12}$). Assignments of non-chaotic values are left unmodified (rule T-ASSIGN$_{11}$ and rule T-ASSIGN$_{21}$). In case the left side variable has a flag, the flag is set to *true* (rule T-ASSIGN$_{11}$).

The transformation rules for timed edges are given in Table 14. Concerning *timers*, the set operation and its transformation are similar to an assignment (rules T-SET$_{11}$, T-SET$_{12}$, T-SET$_{21}$, T-SET$_{22}$). If the expression e in *set* $t := e$ is non-chaotic, the setting is kept unmodified (rules T-SET$_{11}$, T-SET$_{21}$). If the timer has a flag, t's flag b_t gets the value *true* (see rule T-SET$_{11}$). Otherwise, if the expression is chaotic (cf. rules T-SET$_{12}$, T-SET$_{22}$), we set the timer to 0 since in the abstraction, a chaotic timer must be able to expire immediately; the flag of the timer is set to *false* (rule T-SET$_{12}$).

By resetting a timer, the timer variable gets the concrete value *off*, independent of its previous value. So the action stays unchanged while the flag of the timer gets the *true* value if the timer belongs to $Var_{\mathcal{I}}$ (cf. rules T-RESET$_1$, T-RESET$_2$). The same happens with a timeout of the non-chaotic timer (cf. rules T-TIMEOUT$_1$, T-TIMEOUT$_2$). According to this rule the same actions can be taken for the chaotic timer as well, i.e., it can expire immediately. The expiration of the chaotic timer can, however, be postponed according to rule T-NOTIMEOUT by non-deterministically setting the timer to 1 at an arbitrary moment in time.

5.5.3 Experiments

Before we present the results on a larger example — the control-part of a medium-access protocol — we show the effect of the transformation on the state space using a few artificial, small examples and we also give small examples demonstrating the need for both *may* and *must* analyses.

Closing with Chaos

In this subsection we take some simple open systems modelled in DTPROMELA, close them with chaos as a separate process, and illustrate how the state space grows with the buffer length and with the number of signals involved into the communication with the environment.

First, we construct a DTPROMELA model of a process (see Fig. 22) that receives signals a, b, or c from the outside, and reacts by sending back d, e, or f, respectively.

A closing environment will send the messages a, b, and c to the process, and conversely receive d, e, and f in an arbitrary manner. As explained in Sec-

```
proctype proc(){
    start : goto q;
    q: atomic{ if
                :: envch?a -> proch!d; goto q;
                :: envch?b -> proch!e; goto q;
                :: envch?c -> proch!f; goto q;  fi ;
                }
    }
```

Fig. 22. Process

```
    s: atomic{ if
         :: expire(t) -> set(t , 1);  goto s; /* stop sending
                            signals  until the next time slice */
         :: expire(t) -> envch!a; set(t , 0);  goto s;
           ...........
         :: proch?f -> goto s;
         fi }
```

Fig. 23. Environment

tion 5.4, the environment must behave chaotically also wrt. timing behaviour. Therefore, in order to avoid zero-time cycles, the sending actions are guarded by a timeout and an extra clause is added when no more signals are to be sent in the current time slice. A specification of such an environment process is given in Fig. 23.

The queues in the verification model, however, have to be bounded. There are two options in *Spin* for handling queues. The first one is to *block* (option "block" in *Spin*) a process attempting to send a message to a full queue until there is a free cell in the queue. With this option, our "naive" closing leads to a *deadlock* caused by an attempt of a process to send a message to the full queue of the environment while the environment is trying to send a message to the full process queue. Another option is to *lose* new messages in case the queue is full (option "lose" in *Spin*). In this case a large number of messages gets lost. Many properties cannot be verified using this option. Moreover, there is a large class of systems where messages should not get lost, for this would lead to non-realistic behaviour of the system. Still, even when this option is applicable, time and memory consumption grow tremendously fast with the buffer size, as shown in Table 15. We can avoid the deadlock in the system that appears by using option "block" if we limit the number of messages sent by the environment per time slice. For this purpose, we introduce an integer

opt.	buffer	states	trans.	lost messages	memo.(MB)	time (s)
loose	3	3783	13201	5086	2.644	00.24
loose	4	37956	128079	47173	3.976	01.97
loose	5	357015	1.18841e+06	428165	18.936	20.49
loose	6	3.27769e+06	1.08437e+07	3.86926e+06	170.487	4 min 04.74

Table 15. Different buffer sizes, unlimited number of signals per time slice

variable n set to the queue size and modify the options of the *if* statement in such a way that sendings are enabled only if n is positive; n is counted down with every message sent and n is revived every time before a new time slice starts (cf. Fig. 24).

```
:: ( n>0 && expire(t)) -> envch!a; n = n-1; set(t, 0); goto ea;
 ........................
 :: expire(t) -> set(t , 1);  n= BUFFSIZE; goto ea;
```

Fig. 24. Environment with a limited number of messages per time slice

opt.	buffer	states	trans.	mem.(MB)	time (s)
block	3	328	770	2.542	00.06
block	4	1280	3243	2.542	00.10
block	5	4743	12601	2.747	00.24
block	6	16954	46502	3.259	00.78

Table 16. Different buffer sizes (4 signals per time slice)

The verification results for the system closed in such a way are shown in Table 16. Again, though more slowly than in the previous example, the number of states, transitions, memory usage, and time required for the verification grow very fast with the queue length.

Next we fix the length of the queue at 4 and vary the number of different messages sent from the process to the environment and from the environment to the process. Table 17 shows the experimental results. Note that the growth of the state space of the system is now caused by the combinatorial explosion in the queues. (The maximal number of messages that can be sent per time slice is still equal to the length of the queue.)

n-messages	states	trans.	mem.(MB)	time (s)
4	3568	9041	2.644	00.22
5	8108	20519	2.849	00.42
6	16052	40569	3.156	00.75
7	28792	72683	3.771	01.36
8	47960	120953	4.590	02.45
9	75428	190071	5.819	03.86

Table 17. Different numbers of message types

In the experiments for the same process with the embedded environment, the number of states is constant for all the cases considered and equal to 4. As one might have expected, closing a system by a separate environment process behaving chaotically, leads to a state space explosion even for very simple small systems. Tailoring the environment process such that only "relevant" messages can be sent makes the environment process large and complicated, which can also cause the growth of the state space or lead to errors caused by mistakes in the environment design.

Usage of *may* and *must* Analysis

Further we present a couple of illustrative examples showing the difference between the approach of [151, 103] and the one presented in this Chapter. The examples are given in DTPromela. The *may* approach pessimistically removes all data potentially influenced by data from outside and the transformation is based on a *static* may analysis. The approach based on the combined *may/must* analysis treats data from outside *dynamically,* thus achieving a greater precision, but removes parts afterwards which are guaranteed to be chaotic as given by the combined analysis of Section 5.3.

The difference is visible at the locations, where the abstract valuation of some variable can get both $\top$ and $\perp$ depending on the system run. In the *may* approach, the variable instance at this location is handled as chaotic independently of the run; now the value of the variable is treated *dynamically.* The simplest situation of this sort is when the variable gets its value from a signal that can be received both from the environment, and from another process of the system with a reliable value.

As illustration, we take two processes communicating with each other and with the environment. Fig. 25 shows part of the DTPROMELA code of the system specification. Process `A` can receive signal `a(x)` both from process `B` and from the environment. Moreover, `B` always sends this signal with a concrete value.

The DTPROMELA code of the chaotic environment given as external process is shown in Fig. 26. The queues in *DTSpin* are bounded, so we use variable `n` to limit the number of messages that process `B` and the environment can send to `A` during one time slice. Otherwise, the system would deadlock in the attempt of `A` to send a message to the full queue of the environment while the environment is trying to send a message to the full queue of the process.

Furthermore, the environment must behave chaotically also wrt. timing behaviour. Therefore, send actions of the environment are guarded by a timeout, which allows to postpone sendings until the next time slice.

```
proctype A{
  ...
  pa: atomic{
    if :: chA?a,x -> goto decision; fi ;
  };
  decision : atomic{
    if
    :: ( x==0) -> chB!c; goto pa;
    :: ( x==1) -> chEnv!c; goto pa;
    fi }
  ...
}

proctype B{
  ...
  start : atomic{
    set(tB , 5); goto wait_tB;}
    wait_tB:atomic{
      if
      :: ( n>0 && expire(tB)) -> chA!a(0);
          n = n-1; set(tB,5); goto wait_tB;
      :: chB?c -> set(tB, 1); goto wait_tB;
      fi }
}
```

Fig. 25. Open System

```
proctype Env{
  ...
  pe: atomic{
    if
    :: expire(t) -> set(t, 1);
       n=BUFFSIZE; goto pe;
    :: ( n>0 && expire(t)) ->; chA!a, 1;
       n = n-1; set(t, 0); goto pe;
    :: ( n>0 && expire(t)) -> chA!a, 0;
       n = n-1; set(t, 0); goto pe;
    :: proch?c -> goto pe;
    fi ;
  }
}
```

Fig. 26. Environment

Fig. 27 shows the result of the *may* approach and Fig. 28 shows the result obtained with the *may+must* approach. The *may*-analysis marks variable `x` in process `A` as $\top$; therefore, the guards `x==1` and `x==0` are transformed to *true*. As a consequence, the property of the original system, that for every request `a(0, true)` sent by process `B` to process `A`, process `B` eventually gets an answer `c` from `A`, does not hold anymore. `A` can send the answer to the environment instead. According to the approach of this chapter, we do not take the pessimistic view but follow the information about the reliability of the value of `x` dynamically during the system run. Therefore, `B` always gets an answer from `A` for every request. Thus the false negative that is obtained during model checking in the first case does not appear when we model check the closed system in the second case. That justifies the need for the combination of *may* and *must* analysis.

5.5.4 Case Study: a Wireless ATM Medium-access Protocol

To validate the *may* approach, we applied the PML2PML-translator in a series of experiments to the industrial protocol Mascara [169].

Located between the ATM-layer and the physical medium, Mascara is a medium-access layer or, in the context of the ISDN reference model, a transmission convergence sub-layer for wireless ATM communication [9, 105] in local area networks. It has been developed within the *WAND*[1] project [169], a joint European initiative by various telecommunication companies to specify and implement a wireless access system for ATM-LANs.

[1] Wireless ATM Network Demonstrator.

```
proctype A{
  ...
  pa: atomic{
    if
    :: expire(tC) -> set(tC, 0);
       goto decision;
    :: chA?a,x -> goto decision;
    :: expire(tC) -> set(tC, 1); goto pa;
    fi ; };
  decision: atomic{
    if
    :: chB!c; goto pa;
    :: goto pa;
    fi ; }
}

proctype B{
  ...
  :: expire(tB)  -> pAch!a(0);
    set(tB,5); goto wait_tB;
  ...
  }
```

Fig. 27. System transformed using only *may*-analysis

```
proctype A{
  ...
  pa: atomic{
    if
    :: chA?a,x, bx  -> goto decision;
    :: expire(tC) -> set(tC, 1); goto pa;
    :: expire(tC) ->
         set(tC , 0); bx=false; goto decision;
    fi ; };
  decision : atomic{
    if
    :: (( x==0&bx) || (!bx)) -> pBch!c;
         goto pa;
    :: (( x==1&bx) || (!bx)) -> goto pa;
    fi }
}

proctype B{
  ...
  :: expire(tB)  -> chA!a(0, true);
     set(tB,5); goto wait_tB;
  ...
}
```

Fig. 28. System transformed using the combination of *may*- and *must*-analysis

Besides the standard transmission convergence sub-layer tasks such as cell delineation, transmission frame adaptation, header error control, cell-rate decoupling, etc., operating over radio-links, i.e., over a necessarily shared physical medium, adds to the complexity of the protocol. Mascara has to arbitrate *medium access* to the radio environment of a variable number of mobile ATM-stations,[2] provide enhanced error detection and correction mechanisms at various levels to counter the comparatively high bit-error rate of air-borne data-transmission. Last but not least, it has to cater for *mobility* features, allowing a mobile terminal to switch its association with an *access point* in a *handover*.

From the perspective of verification, Mascara is a large protocol. Mascara's specification contains over 300 pages of (graphical) SDL. It is itself composed of various protocol layers and sub-entities (cf. Fig. 29).

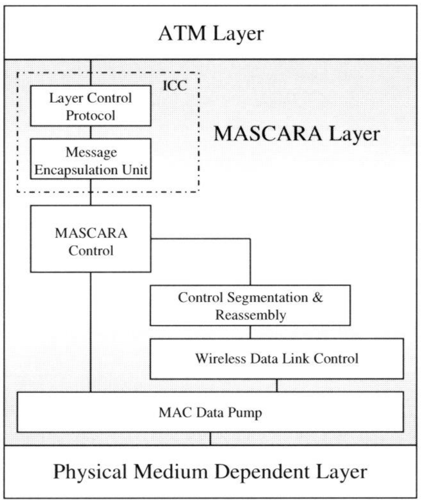
ATM Layer
ICC
Layer Control Protocol
Message Encapsulation Unit
MASCARA Layer
MASCARA Control
Control Segmentation & Reassembly
Wireless Data Link Control
MAC Data Pump
Physical Medium Dependent Layer

Fig. 29. Top-level functional entities

The *layer control protocol* together with the *message encapsulation unit* assists in various ways the information exchange between the Mascara layer and entities located within the upper layers. The *segmentation and reassembly* unit does exactly what its name implies: cutting peer-to-peer control messages (also called MPDUs) into ATM-cell size and putting them together upon reception. All three mentioned top-level entities are comparatively unsophisticated and straightforward, as they mainly perform data transformations. The *WDLC*-layer, operating already on cell-level, is reminiscent of conventional (non-ATM)

[2] Hence the acronym "*M*obile *A*ccess *S*cheme based on *C*ontention *a*nd *R*eservation for *ATM*".

data-link protocols and responsible, per virtual channel, for error- and flow-controlled cell-transmission. The lowest level of Mascara is the *data-pump* including a real-time scheduler, which forms a large portion of the protocol's code-size. Despite its raw size, the functionality offered to the Mascara-layers above is rather simple: the data-pumps of two communicating stations act as duplex, lossy fifo-buffers. The other large part of Mascara, making up almost half of the SDL-code, is its *control entity,* on which we concentrate here. For a more thorough coverage of Mascara's structure and internals, consult the specification material provided by the Wand consortium [169] and [152].

As the name suggests, the *Mascara control* entity (MCL) is responsible for the protocol's control and signalling tasks. It offers its services to the ATM-layer above while using the services of the underlying segmentation and reassembly entity, the sliding-window entities (WLDC's), and in general the low-layer data-pump.

Being responsible for signalling, MCL maintains and manages *associations* linking access points with mobile terminals, and *connections*, i.e., the basic data and signalling transfer channels, corresponding to ATM virtual channels. Mascara control falls into four sub-entities, each divided in various sub-processes themselves. The two important and complex ones are the *dynamic control* (DC) and the *steady-state control* (SSC). The division of work between the dynamic and the steady-state control is roughly as follows: SSC monitors in various ways current associations and the quality of the radio environment in order to ensure an optimal transmission quality, to keep informed about alternative access points, and to initiate in time change of associations, so-called *handovers.* The dynamic control's task, on the other hand, is to set-up and tear down the associations and connections while managing the related administrative work like address management, resource allocation, etc. Of minor complexity are the *radio control* entity (RCL, with the *radio control manager* RCM as its most important process) and the *generic Mascara control* (GMC).

Both are managed by MCL either in response to requests from the upper layer or by taking initiative of its own.

MCL carries out the periodical monitoring of the current radio link quality, gathering the information about radio link qualities of its neighbouring APs to handover to in the case of deterioration of the current association link quality, and switching from one AP to another in the handover procedure. Its functionality is implemented by a dynamic number of processes, using a variety of data-structures, and depending on various timed conditions.

A crucial feature of Mascara is the support of *mobility.* An MT located inside the area cell of an AP is capable of communicating with it. Whenever an MT moves outside the area cell of its current AP, it has to perform a so-called *handover* (HO) to an AP whose area cell MT has moved into. A handover must be managed transparently with respect to the ATM layer, maintaining the agreed quality of service (QoS) for the current connections. So the protocol has to detect the need for a handover, select a candidate AP to switch to and redirect the traffic with minimal interruption.

It is the Mascara control entity that is responsible for the handling of mobility issues. That is why our verification efforts were focused on the Mascara control, particularly on the parts of MT Control managing the handover procedure that was specified within the Vires project [167].

One distinguishes two types of handovers in Mascara: *backward handover* and *forward handover*. The forward handover procedure starts when the connection to the current AP is lost and MT urgently needs to find another AP. Backward handover takes place when MCL notices deterioration of the quality of the current association. Then it looks for the 'best' alternative AP to switch to. After the alternative AP has been found, MT tunes to its old AP. It keeps the association to the old AP until it gets the association to the new one. So MT is able to perform its normal activity in the period the upper layer accomplishes its part of the procedure of associating to the new AP.

In [152], MCL was closed by embedding the chaotic environment *manually*. Not surprisingly, verifying properties of MCL closed with chaos yielded false negatives at first in many cases — the completely chaotic environment was too abstract. Therefore, the traces leading to these false negatives were analyzed, which resulted in a refined environment. The refinement was done by identifying signals that could not be exchanged chaotically lest the verification property was violated, then constructing a specific environment process handling only these signals, and finally closing the obtained still open system by embedding the residual chaos. The conditions imposed on sending the detached signals are in fact the conditions imposed on the behaviour of the rest of the protocol, which later formed the correctness properties for the other protocol entities. Thus, by constructing the environment process we only produce an abstraction of the real environment, keeping it as abstract as possible and leaving the whole model still open, which means that the environment prescribes the order of sendings and receivings for a part of the signals, only. In this way, we can still benefit from embedding chaos into the process.

Of course, closing the system manually is time-consuming and error-prone. With the implemented translator, it became possible to reproduce the same series of experiments quickly, without looking for typos and omissions introduced during the manual closing. Moreover, we performed the same experiments for MCL closed with the chaotic environment modelled as a process. In our experiments we used *DTSpin*, an extension of Spin 3.3.10, using the partial-order reduction and compression options. All the experiments were run on a Silicon Graphics Origin 2000 server on a single R10000/250MHz CPU with 8GB of main memory. Our aim was to compare the state space and resource consumption for the two closing approaches.

Table 18 gives the results for the model checking of MCL with chaos as external process on the left and embedded on the right. The first column gives the buffer size for process queues. The other columns give the number of states, transitions, memory in megabytes and time consumption, respectively. As one can see, the state space as well as the time and the memory consumption are significantly larger for the model with the environment as a process, and they

buf.size	states	trans.	mem.	time(s)	states	trans.	mem.	time(s)
2	9.73e+05	3.64e+06	40.842	15:57	300062	1.06e+06	9.071	1:13
3	5.24e+06	2.02e+07	398.933	22:28	396333	1.85e+06	11.939	1:37
4	2.69e+07	1.05e+08	944.440	1:59:40	467555	2.30e+06	14.499	2:13

Table 18. Model checking MCL with chaos as a process and embedded chaos

grow much faster with the buffer size than for the model with embedded chaos. The model with the embedded environment has a relatively stable state-space size and other verification characteristics.

All variants of closing sketched here were model-checked with *DTSpin*. The results of the experiments confirm that closing the system based on the *may* analysis allows to reduce time and memory consumption compared with the system closed by adding the environment as a process.

5.6 Conclusion

Model checking has gained popularity in industry and is becoming a constituent part of software engineering practice since it is, in principle, a push-button verification technology. The further dissemination of model checking, however, depends on whether it is possible to reduce the significant human involvement in applying techniques like abstraction; automation of these techniques is therefore crucial.

Here, we apply data-flow analysis to transform an open system into a closed, safe abstraction, well-suited for model checking. The approach for *automatic* closing of open systems, based on data and control abstraction of the environment, is taking the most general environment, i.e., the chaotic one. To avoid the detrimental effect of external queues on the state space, the closing environment is embedded into the system. The approach presented here goes beyond [151] in yielding a more refined abstraction. The price for the refinement is a possible (but not necessary) increase of the state space, though the state space of the model is still significantly smaller than the state space of the model closed with the environment built as an outside chaotic process. We partially remove the additional state space without losing precision by an a-priori static analysis, determining variable occurrences that are guaranteed not to be influenced from outside and those which are guaranteed to be chaotic.

Our approach is implemented as a tool that automatically closes DT-Promela translations of SDL-specifications by embedding the timed chaotic environment into the system. The prototype implements the transformation based on *may* analysis only. For future work, we will extend our tool for closing open components with the combined analysis. We also plan to extend the method to account for more complex data types, process creation and more precise transformation for guards and expressions influenced by the environment. Based on the results from [153], another direction for future work is to extend the PML2PML implementation to handle environments more refined than just chaos.

6

Timed Verification with μCRL

μCRL is a process algebraic language for specification and verification of distributed systems. μCRL allows to describe data and behaviour aspects but it has no explicit reference to time. In this work, we propose an approach that allows us to reuse the untimed language and the related toolset for timed verification without extending the language and the toolset. We show some experimental verification results obtained on two timed communication protocols.

The chapter is based on [20].

6.1 Introduction

The specification language μCRL [78] (micro Common Representation Language) is a process algebraic language that was especially developed to take account of data in the study of communicating processes. The μCRL toolset [19] together with the CADP toolset [65] provides support for enumerative model checking. One of the most important application areas for μCRL is the specification and verification of communication protocols. Communication protocols are mostly *timed* systems. A common way to use time is the timeout. In some cases it is possible to abstract from duration and simulate timeouts with a non-deterministic choice. However, in other cases the lengths of the timeouts are essential to the correctness of the protocol. To deal with these cases one needs an explicit notion of time.

In [76], a timed version of the μCRL language is proposed where time is incorporated in μCRL as an abstract data type satisfying a few conditions plus a construct to make an action happen at a specific time. The timed version of the language turned out to be useful as a formalism for the specification and analysis of hybrid systems [82]. However, it is not clear yet whether timed μCRL can be used to analyse systems larger than the examples considered in that paper. Moreover, most of the existing tools cannot be used for timed μCRL without modification. Most importantly, linearisation (translating a specification into the intermediate format) for timed μCRL is not implemented.

The goal of the work we present in this chapter is to establish a framework in which timed verification may proceed using the existing untimed tools. μCRL is powerful framework for data and behaviour aspects of reactive systems that could be reused for timed verification. To achieve the goal of timed verification with untimed tools, we must restrict ourselves to discrete relative time: the state spaces of systems with dense or absolute time are almost always infinite. Techniques, such as regions and zones, which allow finite representations of such infinite state spaces, are not implemented in the untimed tools. Timestamping actions with "absolute" time, as it is done in timed μCRL, leads to infinite state spaces in case of unbounded delays. Consider for example the process `X=sum(t:Time, a@t)` which uses time tags (the `@` symbol must be read as "at time") and thus can do action `a` at any time. The LTS of `X` consists of two states and infinitely many transitions. For this reason, we have chosen a "relative" time solution. Namely, we introduce time through an action `tick`, which by convention expresses one unit of time elapsing. In this case we can specify the process that can do `a` at any time as `Y=tick.Y+a`. The LTS of process Y has two states and two transitions. The advantage of representing time progression as an action is that we stay within the syntax of μCRL. Moreover, the special use of `tick` is compatible with the semantics of μCRL, and hence the existing toolset can be used for analysis and verification.

The proposed discrete time semantics is suitable to express time aspects and analyse time properties of a large class of systems. We argue the usefulness of our approach with verification experiments on μCRL specifications of the positive acknowledgment retransmission protocol (PAR) [155] and the bounded retransmission protocol (BRP) [100], whose behaviour depends on the timers' settings.

To express *timed* properties of systems, we introduce an *LTL*-like timed temporal logic on actions and show how to encode its time constraints with the use of `tick`, which results in *untimed* temporal formulas. These formulas can then be translated to the μ-calculus and checked with the CADP toolset.

The rest of the chapter is organized as follows. In Section 6.2, we sketch the syntax and semantics of μCRL. In Section 6.3 we present the discrete time semantics that we work with, and afterwards in Section 6.4 we explain how timed specifications can be developed within the untimed framework, following the proposed approach. In Section 6.5 we discuss some experimental results. In Section 6.6 we introduce a timed temporal logic. We conclude in Section 6.7 with discussing the related works and directions for future work.

6.2 μCRL: Basic Notions

The specification language μCRL (micro Common Representation Language) is essentially an extension of the process algebra ACP [12] with abstract data types and recursive definitions. The μCRL toolset provides tool support for a subset of the μCRL language. In the remainder of this section, we will give an overview of both the language and its tool support. Details about the language can be found in [78]. Details about the tool support can be found in [19].

Data in μCRL is specified using equational abstract data types. Each data type is declared using a keyword **sort**. Each declared sort represents a non-empty set of data elements. Elements of a data type are declared using keywords **func** and **map**. The keyword **func** is used to declare constructors that define the structure of the data type. The keyword **map** is used to declare function symbols that are not constructors. The keyword **rew** is used to define a set of equations that represent the properties of the data type. The equations following the keyword **rew** are oriented from left to right and used as rewrite rules by the tools, but may be used in both directions for reasoning.

Every μCRL specification must include a specification of the sort `Bool`, which represents the booleans. An example is given in Fig. 30. The sort `Bool` declares two constructors: `true` and `false`. It also declares two functions: `eq : Bool#Bool-> Bool` and `and : Bool#Bool-> Bool`.

The usual way of modelling a system in μCRL is to decompose the system into components and then specify the components and their interactions separately. Components are usually recursively defined using atomic actions, sequential and alternative composition and conditionals.

Actions are abstract representations of events in the real world. They are declared using keyword **act** and are considered to be atomic. A special constant δ is used to represent *deadlocks*, which do not display any behaviour. *Sequential*

```
sort Bool
func true,false: ->Bool
map  eq,and:Bool#Bool->Bool
var  b:Bool
rew  eq(b,b)=true
     eq(true,false)=false
     eq(false,true)=false
     and(true,b)=b
     and(false,b)=false
```

Fig. 30. A μCRL specification of the sort Bool.

```
act  a
     b,c:Bool
proc X=a.X
proc Y=sum(b':Bool,b(b').c(b').Y)

proc Y'(b1:Bool,state:Bool)=
           sum(b':Bool,b(b').Y'(b',false)<|eq(state,true)|>delta)+
           c(b1).Y'(b1,true)<|eq(state,false)|>delta
init Y'(true, true)
```

Fig. 31. Components in μCRL

composition X.Y and *alternative composition* X+Y are two elementary operators that are used to construct processes. There are no priority operators in μCRL. The process X.Y first executes X; when X terminates, it continues with executing Y. The process X+Y behaves either as X or as Y.

The parallel operator can be used to put processes in parallel. The behaviour of X || Y is an arbitrary interleaving of actions of processes X and Y, assuming that there is no communication between X and Y.

It is also possible that X and Y communicate in X || Y. This can be described by declaring on which action names the processes may synchronize. This is done in a *communication* section, which is a section starting with the keyword **comm**. For example, the interaction between actions a, b and c, resulting in action d can be expressed as

```
comm    a|b=ab b|c=bc a|c=ac     a|bc=d b|ac=d c|ab=d
```

The **comm** section says which actions *may* synchronize, but it does not say that they *have to* synchronize. To enforce communication, the unary *encapsulation operator* encap$_H$(X) is introduced. A process encap$_H$(X) can execute all actions of X which are not in H. The encapsulation operator can be used to guarantee that certain actions can occur only in communication.

Sometimes it is convenient to reuse a given specification with different action names. A *renaming operator* rename mapping action names to action names is used for this purpose. The process `rename({a` $\rightarrow$ `b}, X)` behaves as `X` with action `a` renamed to `b`. To make actions invisible, a *hiding operator* `hide`$_I$`(X)` is used. This operator renames action names to τ.

μCRL combines abstract data types with process algebra by allowing atomic actions parameterized by data terms. For example, `send(frame(x,y))` stands for the action `send` parameterized by a data frame with two data parameters `x` and `y`. Data can influence the behaviour of a process via a *conditional operator*. For example, a process `X<|B|>Y`, where `X` and `Y` are processes, behaves as `X` if the boolean condition `B` is *true* and as `Y` otherwise. The *summation operator* `sum(d:D, X(d))`, defined for some process `X(d)` and data type `D`, behaves as `X(t`$_1$`)+X(t`$_2$`)+`..., i.e. as possible choice between `X(d)` for any data term `t`$_i$ taken from `D`.

The heart of a μCRL specification is the **proc** section, where the behaviour of the system is declared. This section consists of equations of the form: `X(`$x_1 : S_1, \ldots, x_n : S_n$`)=t`. Here `X` is the process name, x_i are variables, expressing data parameters of type S_i. Term `t` is a process expression built from actions and expressions of the form `Y(`$d_1, \ldots, d_n$`)` (where `Y` is a process name and d_i are data terms or variables) using the above mentioned operators. A process declaration can thus be recursive. The initial state of the specification is declared in a separate *initial declaration* section **init**. See below for an explanation for `Y'`.

For example, in Fig. 31 we have specified processes X and Y. Process X simply repeats action `a` infinitely often. Process Y infinitely often chooses between T and F nondeterministically and performs `b` and `c` with the same choice as argument.

Besides the parallel composition operator provided by μCRL, other types of parallel composition operators can be defined in terms of the basic operators. For example, the operator `X|{tick}|Y` lets the processes X and Y run interleaved except for the action `tick`, which must be performed synchronously by both X and Y. It can be encoded as follows:

```
act  tick tick'
comm tick|tick=tick'
     X |{tick}| Y = rename({tick'->tick},encap({tick},X||Y))
```

In the process `X||Y`, `tick` actions from X and Y may be performed interleaved or at the same time, resulting in a `tick'`. In the process `encap({tick}, X||Y)`, the interleaved execution is disallowed by means of encapsulation. Finally, the `tick'` is renamed to `tick` to get the desired result. Note that the interaction is not limited to two parties. The result of the interaction may itself interact.

μCRL was successfully applied in the analysis of a wide range of protocols and distributed systems. Recently it was used to support the optimized redesign of the Transactions Capabilities Procedures in the SS No. 7 protocol stack for telephone exchanges [8], to detect a number of mistakes in an industrial protocol over the CAN bus for lifting trucks [77], and to analyse the coordination languages SPLICE [54, 97] and JavaSpaces [164], and to arrive at a formally verified prototype implementation of a multi-channel on-board data acquisition system for Lynx helicopters [67].

Tool support for μCRL is centered around the linear process format [18]. A linear specification consists of a single recursive process, which can choose between possibilities of the form "action followed by a recursive call", provided guard holds:

$$\begin{array}{l} \texttt{proc}\ \mathtt{X}(d_1 : D_1 \cdots, d_n : D_n) = \\ \qquad \sum_{e_{11}:D_{11}} \cdots \sum_{e_{1n_1}:D_{1n_1}} a_1(\boldsymbol{s}_1).X(\boldsymbol{t}_1) \lhd c_1 \rhd \delta + \\ \qquad \vdots \\ \qquad \sum_{e_{k1}:D_{k1}} \cdots \sum_{e_{kn_k}:D_{kn_k}} a_k(\boldsymbol{s}_k).X(\boldsymbol{t}_k) \lhd c_k \rhd \delta + \\ \texttt{init}\ \mathtt{X}(\boldsymbol{t}_0) \end{array}$$

The toolset allows the transformation of μCRL specifications into a linear form [162] (in Fig. 31, process Y' is a linear equivalent of process Y), the optimization of a specification in linear form, the simulation of a linear specification, and the generation of an LTS from a linear specification. The toolset allows the user to apply a reduction method [81, 21] based on τ-confluence [79]. The reduction method guarantees that the reduced LTS is branching bisimilar to the original one.

LTSs generated by μCRL toolset are used as input for the CADP toolset [65]. This provides support for enumerative model checking. Properties to be verified are usually expressed by formulas of *regular alternation-free μ-calculus*. In the *regular alternation-free μ-calculus* [126], one is allowed to use expressions $\langle r\rangle\phi$ and $[r]\phi$ where r is a so-called *regular expression* built from action formulas.

Let $T = (S, Lab, \rightarrow, s_0)$ be an LTS (cf. Def. 2.6). An *action formula* α is defined as follows:

$$\alpha ::= action \mid any \mid \alpha_1 \ \vee\ \alpha_2 \mid \neg\alpha$$

where $action \in Lab$ is an action formula satisfied by the corresponding label only. Any label from *Lab* satisfies *any*. A label satisfies $\neg\alpha$ iff it does not satisfy α, and a label satisfies $\alpha_1 \ \vee\ \alpha_2$ iff it satisfies α_1 or α_2. A *regular expression* r is defined as follows:

$$r ::= \alpha \mid r_1.r_2 \mid r_1 + r_2 \mid r^*$$

Here α is an action formula, $r_1.r_2$ is concatenation, $r_1{+}r_2$ is the choice operator, and r^* is the transitive-reflexive closure. For each regular expression r, we refer to the language it represents as $L(r)$. Intuitively, $\langle r\rangle\phi$ means that ϕ holds after some trace from $L(r)$, and $[r]\phi$ means that ϕ holds after all traces from $L(r)$.

6.3 Semantics of Time

In this section we discuss which time semantics is appropriate for our purpose.

The first choice to be made is between dense and discrete time. It is normally assumed that real-time systems operate in "real", continuous time (though some physicists contest against the statement that the changes of a system state may occur at any real-numbered time point). Due to the development of regions and zones techniques [2], the verification of real-time systems became possible. However, a less expensive, discrete time solution is for many systems as good as dense time in the modelling sense, and better than the dense one when verification is concerned; [88] showed that discrete time suffices for a large and important class of systems and properties, including all systems that can be modelled as timed LTSs and such properties as time-bounded invariance and time-bounded response. Another work that compares the use of dense and discrete time is [29]; the authors state that discrete time automata can be analyzed using any representation scheme used for dense time, and *in addition* can benefit from enumerative and symbolic techniques (such as BDDs) which are not naturally applicable to the dense time systems. Having in mind that we prefer not to step out of the current non-timed framework of μCRL, the choice for discrete time is obvious.

Timers are usually used to express time constraints imposed on a reactive system. An expiration of a timer is a natural way to model an interrupt from hardware or a trigger for a software event. Both interrupt and software event must be handled, and they must be handled exactly once, i.e. when taking an event guarded by a timer condition, we assume that the timer which triggered this event became deactivated (otherwise, the system could handle one event several times). Time progresses by decreasing the values of all active timers by one time unit. We will refer to the time progress action as `tick` and to the period of time between two `tick`'s as a time slice.

We consider a class of systems where delays are significantly larger than the duration of normal events within the system. Therefore, we assume system transitions to be instantaneous. It has been argued in some works that models where any action takes some non-zero time allow more faithful descriptions. However, we believe that such an assumption destroys abstractness of time, as specifications depend on specific implementation choices. In [130], it was shown that the zero duration assumption for atomic actions is more general and leads to much simpler theories. Moreover, this assumption does not prevent from modelling actions that take some time. Whenever it is necessary, we can put an explicit time delay before an atomic action or split it into start- and finish-events.

The assumption about instantaneity of actions leads us to the conclusion that time progress can never take place if there is still an untimed action enabled, or in other words, the time-progress transition has the least priority in the system and may take place only when the system is *blocked*: there is no transition enabled except for time progress and communication with the environment. It means that some actions are urgent, as a process may block the progress of time and enforce the execution of actions before some delay.

```
sort Timer
func off:-> Timer
     on:Nat->Timer
map  pred:Timer->Timer
     expired:Timer->Bool
     set: Timer # Nat -> Timer
     reset: Timer -> Timer
```

```
var  t:Timer
     n:Nat
rew  expired(off)=F
     expired(on(n))=eq(0,n)
     pred(on(n))=on(pred(n))
     pred(off)=off
     set(t, n)=on(n)
     reset(t)=off
```

Fig. 32. A μCRL specification of the sort Timer.

This property is usually called *minimal delay*, *maximal progress* or τ*-urgency* [130]. In CCS-based process algebras it is strongly related to the communication mechanism. Indeed, a communication in CCS yields a τ-action; thus, this property allows to ensure that two processes communicate as soon as they are ready to do so. In the timed process algebras TPL [86], TCSP [150] and TiCCS [170], action urgency is enforced for τ-actions only.

6.4 Specifying Timed Systems in μCRL

In the μCRL framework, we can implement timers as data parameters for processes. Fig. 32 shows the specification of the sort `Timer`. Terms `on(n)` stand for active timers (`n` is of sort `Nat` of natural numbers), while deactivated timers are represented by `off` terms. (Note that μCRL specifications containing the sort `Timer` should also include the sort `Nat` providing an operation `pred` that decreases a non-zero natural number by one and an operation `eq` for checking the equality of two numbers.) The operations we allow on timers are (1) setting a timer to a value given by a natural number that shows the time delay left until the timer expiration; (2) resetting a timer (setting it to `off`). The timer expiration condition given by the predicate `expired` is the check whether the delay until the timer expires is zero. Normally, the action guarded by a timer expiration resets the timer or sets it to a positive value.

Following the time semantics described in Sec. 6.3, we want to model time progression by the `tick` action, which is a global action decreasing all active timers of the system by one and enabled only when the system is blocked. To achieve this, we enable the `tick` action in a component if that component is blocked and if every timer in that component is off or non-zero. By combining components with the `|{tick}|` operator as defined in Sec. 6.2, we get precisely the desired behaviour.

A system is considered blocked if there are no *urgent* actions possible. As μCRL has no priority mechanism, we capture urgency by following a specification discipline.

First, we classify a number of actions as *urgent*. Actions that a component can perform independently of the other components are *internal*. Enabled internal actions are urgent — they take zero time and, hence, they may not be postponed until later, and `tick` may not be proposed as an alternative to an internal action.

The situation with *communication* is more complicated: When the two communicating parties are both ready to communicate, communication should take place in the current time slice. Thus, no `tick` action can be given as an alternative to a communication action. However, when only one of the parties is willing to communicate, time progress should *not* be disabled, meaning that the process willing to communicate but not having this chance yet, should be able to take the `tick` action.

We resolve the problem by introducing asymmetry into communication. Though μCRL has no notions of "sender" and "receiver", it is rather usual for a large class of systems to distinguish between the sending and the receiving party in a communication action. Moreover, it is logical to expect for a correct specification that send events take place in the same time slice in which they become enabled; otherwise communication cannot be seen as synchronous and a message exchanged between the sender and the receiver should be stored in a buffer. Reception, however, can be postponed until the next time slice. Consequently, we allow `tick` as an alternative to a receive action and not to a send action.

The classification of actions results in the classification of component states: We require every state to be either a *receive state*, i.e. a state where only "receive" actions are enabled, or a *non-receive state*, i.e. a state where only "send" and internal actions can be taken. The check that a μCRL specification meets this requirement can be easily automated by introducing conventional names for input and output actions. To simplify matters further, we have used patterns for specifying states of components as μCRL processes.

$$
\begin{aligned}
&\textbf{proc } A(t_1 : \mathit{Timer} \cdots, t_m : \mathit{Timer}, d_1 : D_1 \cdots, d_n : D_n) = \\
&\qquad a_1.X_1(\boldsymbol{t}_1, \boldsymbol{y}_1) \triangleleft \mathit{expired}(t_1) \triangleright \delta + \\
&\qquad \vdots \\
&\qquad a_m.X_m(\boldsymbol{t}_m, \boldsymbol{y}_m) \triangleleft \mathit{expired}(t_m) \triangleright \delta + \\
&\qquad \mathit{tick}.A(\mathit{pred}(\boldsymbol{t}), \boldsymbol{d}) \triangleleft \mathit{not}(\textstyle\bigvee_{j=1}^{m} \mathit{expired}(t_j)) \triangleright \delta + \\
&\qquad \textstyle\sum_{e_{11}:D_{11}} \cdots \sum_{e_{1n_1}:D_{1n_1}} \mathit{in}_1(\boldsymbol{s}_1).Y_1(\boldsymbol{t}_1, \boldsymbol{x}_1) \triangleleft c_1 \triangleright \delta + \\
&\qquad \vdots \\
&\qquad \textstyle\sum_{e_{k1}:D_{k1}} \cdots \sum_{e_{kn_k}:D_{kn_k}} \mathit{in}_k(\boldsymbol{s}_k).Y_k(\boldsymbol{t}_k, \boldsymbol{x}_k) \triangleleft c_k \triangleright \delta
\end{aligned}
$$

Fig. 33. Pattern of a receive state.

$$
\begin{aligned}
&\textbf{proc } B(t_1 : \mathit{Timer} \cdots, t_m : \mathit{Timer}, d_1 : D_1 \cdots, d_n : D_n) = \\
&\qquad \textstyle\sum_{e_{11}:D_{11}} \cdots \sum_{e_{1n_1}:D_{1n_1}} b_1(\boldsymbol{s}_1).Z_1(\boldsymbol{t}_1', \boldsymbol{x}_1) \lhd c_1 \rhd \delta + \\
&\qquad \vdots \\
&\qquad \textstyle\sum_{e_{l1}:D_{l1}} \cdots \sum_{e_{ln_l}:D_{ln_l}} b_l(\boldsymbol{s}_l).Z_l(\boldsymbol{t}_l', \boldsymbol{x}_l) \lhd c_l \rhd \delta
\end{aligned}
$$

Fig. 34. Pattern of a non-receive state.

In μCRL, we use process declarations to specify states of a component. All μCRL processes which correspond to states in one component have the same list of parameters. For a component with m timers and n other variables, the first m parameters are timers and the next n are the other variables. The patterns of receive and non-receive states are given in Fig. 33 and 34, respectively. We use $\boldsymbol{t}$ to denote a vector of timer terms (data terms of the type `Timer`) and $\boldsymbol{x}$, $\boldsymbol{y}$ to denote vectors of untimed data terms. After a receive, internal or send action, timers of the component can be set or reset, data parameters can be modified and the state of the component may change. After a `tick` action, all active timers of the component are decreased by operation $pred(\boldsymbol{t})$ and everything else remains unchanged. Receive and non-receive states of the component have different transitions: In a receive state, we have timer expiration events $a_1, \ldots, a_m$ for expired timers, `tick` if no timer is expired, and receive actions $in_1, \ldots, in_k$. In non-receive states, we only have send and internal actions $b_1, \ldots, b_l$.

When we build a system from components, we must not only make sure that time progression is handled correctly, but also that all enabled communications within the system are enforced. The first requirement means using `|{tick}|`, the latter means encapsulation of send and receive actions. If we specify a closed system that does not communicate with the outside world, all send and receive actions should be encapsulated. If a system is open, i.e. it sends and receives messages from the outside world, then only the sends and the receives within the system should be encapsulated. Let H be the set of send and receive actions that take place within the system. Then a system with N components is described by the following μCRL init statement:

$$\texttt{init encap}(\{H\}, C_1 \mid \{tick\} \mid C_2 \cdots \mid \{tick\} \mid C_N)$$

Fig. 35 contains the μCRL code of a simple watchdog. The watchdog's task is to watch a component working properly. The component is supposed to send a signal `ok` every m time units to inform the watchdog that it is functioning normally. The watchdog is ready to accept signals from the component at any time. In case it does not receive the signal `ok` within m time units, it will send out a warning signal `alarm` immediately.

The watchdog is specified by two μCRL process declarations. Process `A` waits either for a signal `ok` or for an expiration of timer `t`. If `ok` comes, the

```
proc A(t:Timer,m:Nat)=
    expire.B(reset(t),m)<|expired(t)|>delta+
    tick.A(pred(t),m)<|not(expired(t))|>delta+
    recv(ok).A(set(t,m),m)<|true|>delta
proc B(t:Timer,m:Nat)=
    send(alarm).A(set(t,m),m)<|true|>delta
init A(on(5),5)
```

Fig. 35. A μCRL watchdog

timer is set to m again. Otherwise, an alarm signal is issued by process B and the timer is set to m. The watchdog is an open system, i.e. it communicates with outside by receiving ok and sending alarm, so none of the send and receive actions is encapsulated.

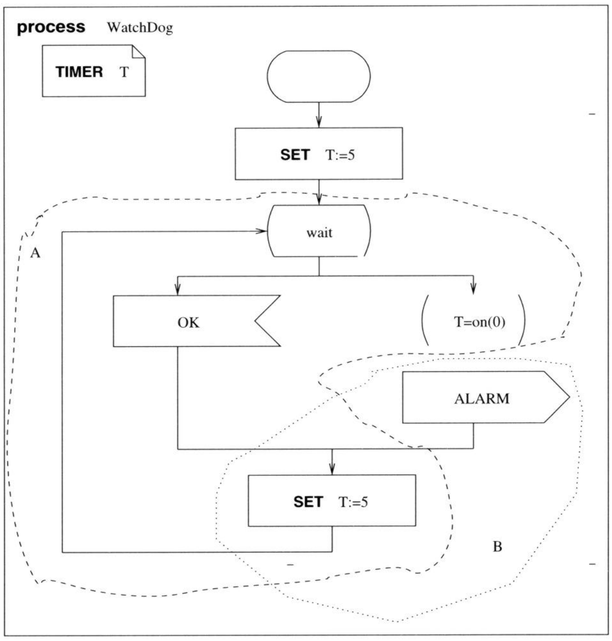

Fig. 36. An SDL watchdog

The discrete time semantics that we have chosen is similar to SDL time semantics (cf. Sec. 3.2). An analogous watchdog can also be specified in SDL. In Fig. 36, an SDL specification of a watchdog process is given. It waits either for a signal OK or for a timeout of timer T. If the signal comes in time, the timer is set to 5 again. Otherwise, the signal ALARM is sent upon expiration of the timer, the timer is set to 5, and the process comes back to the state wait. Intuitively, a set of μCRL process declarations that represent one component corresponds to an SDL process specification. Receive and send actions in a μCRL specification correspond to input and output actions of SDL processes, respectively.

6.5 Experiments

We have tested our approach on two protocols: the positive acknowledgment retransmission protocol (PAR) [155] and the bounded retransmission protocol (BRP) [100, 49]. These are two classical examples of communication protocols where time issues are essential for the correct functionality of the protocol. The goal of the experiments was to show how our approach can be applied to the specification of time aspects of the protocols and to the verification of properties that depend on time issues.

BRP

BRP is a variation of the Alternating Bit Protocol [155] where only a bounded number of retransmission of packets is allowed and timeouts are used to detect a packet loss or an abortion of transmission. BRP behaves like a buffer, i.e. it reads data from one sender client and then delivers it at a receiver client.

The usual scenario includes a sender, a receiver, a message channel and an acknowledgment channel. The two channels are assumed to either lose a message or deliver it correctly. They also delay messages for time TD. Here we consider BRP together with its environment consisting of a sender client and a receiver client (see Fig. 37). The description of BRP and BRP's environment is adopted from [52].

The sender client gives a list $(d_1, \ldots, d_n)$ to the sender. Ideally, each element d_i should be delivered to the receiver client. When delivered, the element d_i of the list is accompanied by an indication: I_FST, I_INC, I_OK, I_NOK. Indication I_FST is used if d_i is the first element of the list and more elements will follow. All intermediate elements of the list are accompanied by indication I_INC. The last element of the list is delivered together with the I_OK indication. If something goes wrong, I_NOK, "not OK" indication is delivered without data.

The sender client is informed after the transmission of the whole list, or when the transmission is aborted. The indication for the sender client is one of the following values: I_OK, I_NOK, I_DK. After an I_OK or an I_NOK indication the sender client can be sure that the receiver has got the same indication. An I_DK indication may occur after the delivery of the last element. The information about a successful delivery of the last element is transported over a lossy channel. If the acknowledgment for the last element fails, there is no way to know that the last element has been delivered correctly. After I_OK and I_DK, the sender client is ready to transmit the next package.

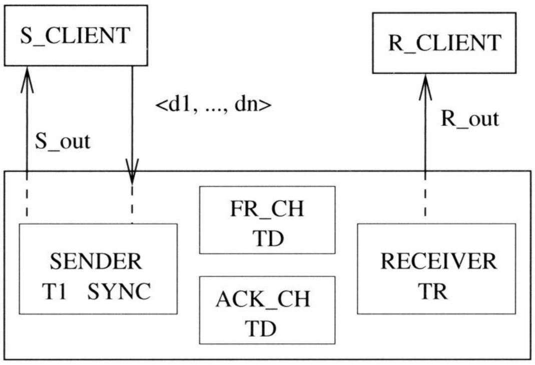

Fig. 37. BRP

From the sender client, the sender receives a package $(d_1, \ldots, d_n)$ to transmit ([80]). It sends the elements of the list one by one over the message channel. For each element of the list, the sender forms a frame consisting of two indication elements, a bit and the list element. The first indication shows whether the element is the first element of the list. The second one indicates whether the element is the last element of the list. The bit is an alternating bit that is used to guarantee that data do not get duplicated. The sender sends the frame via the message channel. To detect the loss of frames and/or acknowledgments, the sender sets timer `T1` and waits for the acknowledgment from the receiver.

In the waiting state, the sender considers several possibilities. If the sender receives an acknowledgment, the sender negates its alternating bit and proceeds with sending the next frame. If the sender receives an acknowledgment for the last element of the list, it sends `I_OK` to the sender client and is ready to start with another list.

If the sender does not get the acknowledgment in time, it wakes up when timer `T1` expires. The number of retransmissions for each element of the list is limited by `MAX`. If the number of retransmissions attempted has not reached `MAX`, the sender resends the same frame. Otherwise, the sender sends an `I_DK` or an `I_NOK` indication to the sender client, depending on whether the current list element is the last element of the list or not and waits until timer `SYNC` expires. This timer ensures that the sender does not start the transmission of another list before the receiver has properly reacted to the failure. Upon the expiration of timer `SYNC`, the rest of the list is skipped and the sender becomes ready to start with a new list.

The receiver receives frames from the message channel. Upon the reception of a frame, the receiver checks whether the frame came with the correct alternating bit. If the alternating bit coincides with the one expected by the receiver, it delivers the data element together with the proper indication to the receiver client and sends an acknowledgment over the acknowledgment channel.

If the frame comes with a wrong alternating bit, the receiver discards the frame and sends the acknowledgment. The receiver is able to detect situations when the sender has given up, namely, the receiver sets timer TR after receiving a frame and waits for the next frame. If the timer expires, the receiver delivers indication I_NOK to the receiver client and becomes ready to receive a next list.

To ensure that no premature timeout is possible for the sender, the sender sets timer T1 to a value longer than twice delay on the channels ($\mathtt{T1} > 2{\cdot}\mathtt{TD}$) and waits for the acknowledgment. It is enough for the message channel to deliver a frame and for the acknowledgment channel to deliver an acknowledgment [52].

A premature timeout at the receiver would abort the connection when there is still a possibility for some frame to arrive. To ensure that no premature timeout is possible for the receiver, the receiver's timer TR should be set to a value that satisfies the following condition [52]: $\mathtt{TR}\geq 2{\cdot}\mathtt{MAX}{\cdot}\mathtt{T1}{+}3{\cdot}\mathtt{TD}$.

In the case of a failure, the sender should not start transmitting a new list until the receiver has reacted properly to the failure. Timer SYNC, which is used for synchronisation in case of a failure, should be set to a value that satisfies the following condition [52]: $\mathtt{SYNC}\geq\mathtt{TR}$.

We assume that the packets transmitted by BRP are lists of non-repeating natural numbers. It means that the system is infinite. Therefore, we apply data abstractions in order to arrive at a finite state verification model. The protocol should ensure that if the sender is transmitting a list l, the sequence of elements that the receiver client gets forms a prefix of l. It can be shown that this holds if in a list of non-repeating naturals the following properties hold ([49]):

- for any two values e_1 and e_2 on positions i and j respectively in the list, with $i < j$, either e_2 is not delivered to the receiver client, or e_1 is delivered to the receiver client before e_2.
- for any two values e_1 and e_2, where e_1 is delivered before e_2 to the receiver client, e_1 and e_2 occur on positions i and j resp. in the list with $i < j$.

This gives the following idea for a data abstraction: We distinguish two natural numbers p_1, p_2, which are abstracted into e1, e2 respectively, while all the other naturals are non-distinguishable and they are abstracted into an abstract element nd.

An arbitrary nonempty list of non-repeating naturals is represented by an abstract list of one of the forms: e1l, e2l, e1e2l, e2e1l, or nemp. nemp represents non-empty lists that contain neither e1 nor e2, e1l represents lists containing only e1, e1e2l represents lists containing first e1 and then e2. An empty list is represented by emp.

Given this data abstraction for lists, we can define an abstract operation on abstract lists for each concrete operation on concrete lists. We are interested in the operation that splits a list into two parts: the first element of the list and the tail. The abstraction of the operation is illustrated by a directed graph in Fig. 38. The nodes of the graph in Fig. 38 are labelled by abstract lists. An arrow from node n to node n' is labelled by the first element of the list in n.

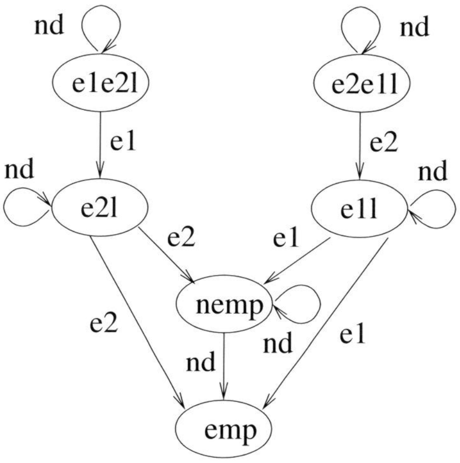

Fig. 38. Abstract split operation

The destination node n' is labelled by the tail of the list in n. This abstraction is analogous to a well-known canonical abstraction proposed in [75].

We have specified BRP in μCRL using timers to represent delays on the channels and timeouts at the receiver and the sender side, and list abstraction to represent all possible lists. Since the system is open, we have closed the system by the sender client process that provides abstract lists for the sender and the receiver client process that receives frames and indications delivered by the receiver.

Using the μCRL toolset we have generated the LTS for the μCRL specification of the protocol. Then, with the CADP toolset, we have verified a number of properties expressed by formulas of the regular alternation-free μ-calculus [126].

One of the properties is the absence of reordering in the delivery of elements. For example, if the sender receives an abstract list `e2e1l` from the sender client, it is never the case that element `e1` is delivered to the receiver client before element `e2`. That can be expressed by the following formula of the regular alternation-free μ-calculus:

```
[T*."get_lst(e2e1l)".(not('rdeliver(e2.*)' or 'get_lst(.*)'))*.
     'rdeliver(e1.*)'.(not('rdeliver(e2.*)' or 'get_lst(.*)'))*.
     'rdeliver(e2.*)']F.
```

Here `get_lst(e2e1l)` means that the sender gets a list containing `e2` before `e1`, `get_lst(.*)` stands for getting a new list to transmit, and `rdeliver(e1.*)` stands for a pair that consists of element `e1` and some indication delivered to the receiver client.

Another property is that the sender client and the receiver client should have corresponding indications. For example, if the receiver client gets indication `I_OK`, then the sender client should receive either `I_OK` or `I_DK`. This property can be expressed as inevitable reachability (cf. [56]) of indication I_OK or I_DK by the server client after indication I_OK given to the receiver client:

```
[T*.’get_lst(.*)’.(not(’get_lst(.*)’))*.’rdeliver(.*I_OK)’]
     mu X.([’get_lst(.*)’]F and <T>T and
            [not(‘‘sdeliver(I_OK)’’ or ‘‘sdeliver(I_DK)’’)]X)
```

Here, sdeliver(I_OK) stands for the delivery of the indication I_OK to the sender client. Proving property

```
[(not(’rdeliver(.*I_NOK)’))*.’’sdeliver(I_NOK)’’]F,
```

we show that indication I_NOK for the sender client is always preceded (cf. [56]) by indication I_NOK for the receiver client.

These properties hold for the system with correct timeouts and do not hold for the system with premature timeouts. The μCRL specification for BRP and the properties are available at www.cwi.nl/˜ ustin/tmcrl.html.

PAR

The usual scenario for PAR includes a sender, a receiver, a message channel and an acknowledgment channel. The channels delay the delivery of messages. Moreover, they can lose or corrupt messages. The sender receives a frame from the upper layer, sends it to the receiver via the message channel, and waits for a positive acknowledgment from the receiver via the acknowledgment channel. When the receiver has delivered the message to the upper layer it sends an acknowledgment to the sender. After the positive acknowledgment is received, the sender becomes ready to send a next message. The receiver needs some time to deliver the received frame to an upper layer. The sender handles lost frames by timing out. If the sender times out, it re-sends the message.

The following is a an example of an erroneous scenario. The sender times out while the acknowledgment is still on the way. The sender sends a duplicate, then receives the acknowledgment and believes that this is the acknowledgment for the duplicate. The sender sends the next frame, which gets lost. However, the sender receives the acknowledgment for the duplicate, which it believes to be the acknowledgment for the last frame. Thus the sender does not retransmit the lost message and the protocol fails. To avoid this erroneous behaviour, the timeout interval must be long enough to prevent a premature timeout, which means that the timeout interval should be larger than the sum of delays on the message channel, the acknowledgment channel and the receiver [155].

We have specified PAR in μCRL using timers to represent delays on the channels and the receiver and the timeout for the sender. Since the system is open, i.e. both the sender and the receiver communicate with upper layers, we have closed the system by the environment process that provides frames for the sender and receives frames delivered by the receiver. If the sender is ready to send the next frame before the environment gets the previous frame delivered by the receiver, the environment process issues an error action err. The err action also occurs if the environment gets a wrong (not sent to the sender) frame from the receiver.

Using the μCRL toolset we have generated the LTS for the μCRL specification of the protocol. Then, with the CADP toolset, we have verified a number of properties expressed by formulas of the regular alternation-free μ-calculus. One of the properties is the absence of traces containing the error action `err`:

```
[T*."err"]F,
```

which holds when the sender's timeout is large enough to avoid premature timeouts.

Another property we have checked was inevitable reachability of an `__out` action after an `__in` action, meaning that the frame sent by the sender to receiver will always be delivered by the receiver to the environment:

```
[T*."__in"] "mu" X.(<T>T and [not("__out")]X).
```

This property holds neither for the system with correct timeout intervals nor for the system with premature timeouts. This can be explained by the fact that the message channel may continuously lose or corrupt a frame, so the frame will never be delivered to the environment. Using a pattern for fair reachability given in [126], we have specified the property stating fair reachability of an `__out` action after an `__in` action:

```
[T*."__in".(not("__out"))*]<(not("__out"))*."__out">T.
```

This property holds for the system with correct timeout intervals and not for the system with wrong ones. The μCRL specification for PAR and the properties are available at `www.cwi.nl/~ ustin/tmcrl.html`.

6.6 Timed Verification

In the previous sections we showed how to specify a timed system in μCRL and how to verify properties dependent on the settings of timers. The considered properties are "qualitative", i.e. they concern only the order of events. In this section, we discuss how to verify "quantitative" *timed* properties, like "event a happens within 3 time units after event b". For this purpose, we introduce an *LTL*-like language that allows the direct use of timed constraints, and then show how to encode these timed constraints with the use of `tick`.

6.6.1 Regular *LTL*

First, we will give an untimed version of the action-based linear temporal logic. It is a variation of *tLTL* of Kaivola [106] extended with regular expressions [126]. As interpretation model, we consider finite LTSs. Let $T = (S, Lab, \rightarrow, s_0)$ be an LTS (cf. Def. 2.6). Action formulas and regular expressions are defined as in Sec. 6.2.

The logic we consider here is action-based, so we are interested in paths of the form $\alpha_1\alpha_2 \ldots \in Lab^\omega$, but not in paths of the form $\pi = s_0s_1 \ldots \in S^\omega$ (cf. Sec. 2.3). We say that a sequence of labels $w = \alpha_1\alpha_2 \ldots \in Lab^\omega$ is a *path* of LTS T iff there is an trace ζ of T such that $\zeta_\lambda(i) = \alpha_i$ for all $i \geq 1$ and $\zeta_\gamma(0) = s_0$ (cf. Sec. 2.2).

Definition 6.1. [SEMANTICS OF REGULAR EXPRESSION]
For a path $w = \alpha_1\alpha_2 \ldots \in Lab^\omega$, we define $w, i, k \models_{\mathcal{R}} r$ for a sequence $w, i, k = \alpha_i \ldots \alpha_k$ if $\alpha_i \ldots \alpha_k \in L(r)$, where $L(r)$ is the language defined by r.

Further we define regular *LTL*, where *LTL* modalities are parameterized by regular expressions.

Definition 6.2. [SYNTAX OF REGULAR *LTL*]

$$\phi ::= \top \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \, U(r) \, \phi_2$$

where r stands for a regular expression.

We use $\bot$, $\wedge$ and $\Rightarrow$ as derived operators in the usual way, and define $\langle r\rangle\phi = \top \, U(r) \, \phi$, $[r]\phi = \neg\langle r\rangle\neg\phi$.

First we give an intuition for the formulas of regular *LTL* and then we provide a more formal semantics.

$\langle r\rangle\phi$ holds on a path w if there exists a prefix $w, 1, i$ of w that satisfies r and ϕ holds on the suffix of w starting at $\alpha_{(i+1)}$.

$[r]\phi$ holds on a path w if for $w, 1, i$ that satisfies r, ϕ holds on the suffix of w starting at $\alpha_{(i+1)}$.

$\psi \, U(r) \, \phi$ holds on a path w if there exists such an $i \geq 1$ on the path such that the sequence $w, 1, i$ satisfies r, the path starting at $\alpha_{(i+1)}$ satisfies ϕ, and the path starting at any action before $\alpha_{(i+1)}$ satisfies ψ.

Definition 6.3. [SEMANTICS OF REGULAR *LTL*]
Let w, i be the suffix of w starting at $\alpha(i)$. Then

- $w, i \models \top$;
- $w, i \models \neg\phi$ *if* $w, i \not\models \phi$;
- $w, i \models \psi \vee \phi$ *if* $w, i \models \psi$ *or* $w, i \models \phi$;
- $w, i \models \psi \, U(r) \, \phi$ *if there exists some* $k \geq i$ *such that*
  - $w, i, k \models_{\mathcal{R}} r$, *and*
  - $w, k+1 \models \phi$, *and*
  - *for all* $j : i \leq j \leq k$, $w, j \models \psi$ *holds.*

We say that w *satisfies* ϕ, denoted as $w \models \phi$, if $w, 1 \models \phi$. Formula ϕ is *satisfied* by an LTS T if all paths of T starting at the initial state satisfy the formula.

6.6.2 Regular *LTL* with Time

Now we extend regular *LTL* with *time constraints* of the form: $\geq c$, $\leq c$, $= c$, where c is a non-negative integer constant. Further, we refer to a time constraint of this form as *tc*.

Definition 6.4. [SEMANTICS OF TIME CONSTRAINTS]
Let $d(w, i, k)$ denote the number of `tick` *steps in a finite sequence w, i, k. Then:*

- *$w, i, k \models \leq c$ if $d(w, i, k) \leq c$;*
- *$w, i, k \models \geq c$ if $d(w, i, k) \geq c$;*
- *$w, i, k \models = c$ if $d(w, i, k) = c$.*

Definition 6.5. [SYNTAX OF REGULAR *LTL* WITH TIME]

$$\phi ::= \top \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 U(r)_{tc}\phi_2$$

where tc is a time constraint and r stands for a regular expression that does not mention the action `tick`.

We use $\perp$, $\wedge$ and $\Rightarrow$ as derived operators in the usual way, and define $\langle r\rangle_{tc}\phi = \top\, U(r)_{tc}\, \phi$, $[r]_{tc}\phi = \neg\langle r\rangle_{tc}\neg\phi$.

The intuition about regular expressions is that they hold on traces regardless of time progression. This means that a path with `tick`s satisfies a regular expression if the path with the `tick` steps projected out satisfies the path formula. We refer to a path w with all `tick` steps projected out as $\pi(w)_{tick}$.

Definition 6.6. [*tick*-SEMANTICS OF REGULAR EXPRESSIONS]
We define

$$w, i, k \models_{\mathcal{R}}^{tick} r \text{ iff } \pi(w)_{tick} \models_{\mathcal{R}} r$$

The intuitive semantics of the formulas is similar to those of regular *LTL*. $\langle r\rangle_{tc}\phi$ holds on a path w if there exists a prefix $w, 1, i$ of w that satisfies r and the time constraint tc, and ϕ holds on the suffix of w starting at $\alpha_{(i+1)}$.

$[r]_{tc}\phi$ holds on a path w if for $w, 1, i$ that satisfies r and time constraint tc, ϕ holds on the suffix of w starting at $\alpha_{(i+1)}$.

$\psi\, U(r)_{tc}\, \phi$ holds on a path if there exists an action on the path such that the path up to that action matches both r and tc, the suffix of the path starting after this action satisfies ϕ and the path starting at any action before satisfies ψ.

Definition 6.7. [SEMANTICS OF REGULAR *LTL* WITH TIME]
Let w, i be the suffix of w starting at $\alpha(i)$. Then

- *$w, i \models^{tick} \top$;*
- *$w, i \models^{tick} \neg\phi$ if $w, i \not\models^{tick} \phi$;*
- *$w, i \models^{tick} \psi \vee \phi$ if $w, i \models \psi$ or $w, i \models \phi$;*
- *$w, i \models^{tick} \psi\, U(r)_{tc}\, \phi$ if there exists some $k \geq i$ such that*
  - *$w, i, k \models_{\mathcal{R}}^{tick} r$, and*
  - *$w, i, k \models tc$, and*
  - *$w, k + 1 \models^{tick} \phi$, and*
  - *for all $j : i \leq j \leq k$ $w, j \models^{tick} \psi$ holds.*

We say that w *satisfies* ϕ, denoted $w \models^{tick} \phi$, if $w, 1 \models^{tick} \phi$. Formula ϕ is *satisfied* by an LTS T if all paths of T starting from the initial state satisfy the formula.

Example 1: Each *request* is followed by an *answer* in at most 5 time units:

$$[any^*.request]\langle any^*.answer\rangle_{\leq 5}\top$$

Example 2: *request* is never followed by *fail* within 2 time units.

$$[any^*.request][any^*.fail]_{\leq 2}\bot$$

6.6.3 `tick`-encoding of Regular *LTL* with Time

In this section we present a construction for translating a formula from regular *LTL* with time into regular *LTL* with `tick`. The key to this translation is the construction of a regular expression over an action domain with `tick` from a regular expression over a domain without `tick` but with a time constraint. This is done by translating both the regular expression and the time constraint into deterministic finite automata, combining these automata into a single automaton and translating this automaton back into a regular expression.

Regular expressions and deterministic finite automata have the same expressive power and can be translated into each other [98]. Let $\mathcal{RE}(A)$ be the translation from a deterministic finite automaton A to an equivalent regular expression r and let A_r be the deterministic finite automaton obtained by the transformation of a regular expression r into a deterministic finite automaton.

Next, we will give the translation of time constraints into deterministic finite automata. But first, we give the formal definition of deterministic finite automata and languages recognized by finite automata.

Definition 6.8.
*A deterministic finite automaton (*DFA*) A is a tuple $(S,\ \Sigma,\ T,\ s_0,\ F)$, where*

- *S is a set of states;*
- *Σ is a set of labels;*
- *$T\colon S \times \Sigma \to S$ is a transition function;*
- *s_0 is an initial state;*
- *$F \subseteq S$ is a set of final states.*

The set of strings recognized by $\mathcal{A}$ is given by

$$L(\mathcal{A}) = \{\alpha_1 \ldots \alpha_n \mid \exists s_1, \ldots s_n \in S,\ s_n \in F, \forall j = 0..n-1 : (s_j, \alpha_{j+1}, s_{j+1}) \in T\}$$

Lemma 6.1.
For each time constraint tc there is a deterministic finite automaton A_{tc} recognizing it.

Proof. The deterministic finite automata recognizing time constraints can be built as follows:

$$A_{\leq c} =$$

$$(\{0, 1, \ldots, c+1\}, \{\texttt{tick}\}, \{(c+1, \texttt{tick}, c+1), (i, \texttt{tick}, i+1) \mid i = 0 \ldots c\},$$

$$\{0\}, \{0, 1, \ldots, c\})$$

$$A_{=c} =$$

$$(\{0, 1, \ldots, c+1\}, \{\texttt{tick}\}, \{(c+1, \texttt{tick}, c+1), (i, \texttt{tick}, i+1) \mid i = 0 \ldots c\},$$

$$\{0\}, \{c\})$$

$$A_{\geq c} =$$

$$(\{0, 1, \ldots, c\}, \{\texttt{tick}\}, \{(i, \texttt{tick}, i+1), (c, \texttt{tick}, c) \mid i = 0 \ldots c-1\}, \{0\}, \{c\})$$

□

We now have a deterministic finite automaton A_r corresponding to the regular expression and a deterministic finite automaton A_{tc} corresponding to the time constraint. All we need to do is to build the product automaton, which will recognize all interleavings of strings recognized by these two automata. The following definition gives such a construction:

Definition 6.9.
Given two deterministic finite automata $A_r \equiv (S_1, \Sigma_1, T_1, I_1, F_1)$ and $A_{tc} \equiv (S_2, \{tick\}, T_2, I_2, F_2)$, we define $A = A_r \times A_{tc}$ as (S, Σ, T, I, F), where

- $S = S_1 \times S_2$;
- $\Sigma = \Sigma_1 \bigcup \{tick\}$;
- $T \colon S \times \Sigma \rightarrow S$ *such that*
  - $T((s_1, s_2), a) = (\hat{s}_1, s_2)$ *iff* $s_1, \hat{s}_1 \in S_1$, $s_2 \in S_2$ *and* $T_1(s_1, a) = \hat{s}_1$,
  - $T((s_1, s_2), tick) = (s_1, \hat{s}_2)$ *iff* $s_1 \in S_1$, $s_2, \hat{s}_2 \in S_2$ *and* $T_2(s_2, tick) = \hat{s}_2$;
- $I = I_1 \times I_2$;
- $F = F_1 \times F_2$.

Lemma 6.2.
Let A_r be a DFA obtained from a regular expression r that does not mention tick, and A_{tc} be the DFA obtained from a time constraint tc. Then $A_r \times A_{tc}$ is a deterministic finite automaton.

Proof. The alphabet of A_r does not intersect with the one of A_{tc}. Both A_r and A_{tc} are deterministic, so for each (s_1, s_2) in S there is at most one outgoing arrow for each element from Σ_1 and at most one outgoing arrow labelled by *tick*. □

Lemma 6.3.
Let w, i, k be some sequence, r be a regular expression, tc be a time constraint, and A_{tc} be a DFA recognizing tc and A_r be a DFA recognizing r. Then $w, i, k \models_{\mathcal{R}}^{tick} r$ and $w, i, k \models tc$ iff $w, i, k \models_{\mathcal{R}} r'$ where $r' = \mathcal{RE}(A_r \times A_{tc})$.

Proof. Straightforward. □

We can now define the translation of regular *LTL* with time to regular *LTL* with `tick`.

Definition 6.10.
The function $\mathcal{T}$ translating a formula ϕ of regular LTL with time to a formula of regular LTL with `tick` *is given by:*

$$\begin{aligned}
\mathcal{T}(\top) &= \top \\
\mathcal{T}(\neg\phi) &= \neg\mathcal{T}(\phi) \\
\mathcal{T}(\psi \vee \phi) &= \mathcal{T}(\psi) \vee \mathcal{T}(\phi) \\
\mathcal{T}(\psi\, U(r)_{tc}\, \phi) &= \mathcal{T}(\psi) U(r') \mathcal{T}(\phi)
\end{aligned}$$

where

$$r' = \mathcal{RE}(A_r \times A_{tc})$$

This translation preserves satisfaction:

Lemma 6.4.
For an LTS $T = (S, Lab, \rightarrow, s_0)$ and a formula ϕ of regular LTL with time, we have

$$T \models^{tick} \phi \Longleftrightarrow T \models \mathcal{T}(\phi) \ .$$

Proof. The proof is by induction on the structure of the formula. The induction hypothesis is given by the following:

$$T \models^{tick} \phi \Longleftrightarrow T \models \mathcal{T}(\phi) \qquad (1)$$

(1) can be reformulated as:

$$w, i \models^{tick} \phi \text{ implies } w, i \models \mathcal{T}(\phi) \qquad (2)$$

and

$$w, i \models \mathcal{T}(\phi) \text{ implies } w, i \models^{tick} \phi \qquad (3)$$

for every sequence w, i.

The basis case is for ϕ being $\top$. $w, i \models^{tick} \top$ and $w, i \models \top$, and thus $w, i \models \mathcal{T}(\top)$.

We proceed by considering the inductive step. Let ψ and ϕ be two temporal formulas satisfying the induction hypothesis. We have to show that each of the formulas $\neg\phi$, $\psi \vee \phi$ and $\psi\, U(r)_{tc}\, \phi$ satisfies the hypothesis.

Case: $\neg\phi$
Assume that $w, i \models^{tick} \neg\phi$. By Def. 6.7, this implies $w, i \not\models^{tick} \phi$. By the counter-positive of (3), we can conclude that $w, i \not\models \mathcal{T}(\phi)$, which by Def. 6.3 leads to $w, i \models \neg\mathcal{T}(\phi)$. By Def. 6.10, this leads to $w, i \models \mathcal{T}(\neg\phi)$

Assume that $w, i \models \mathcal{T}(\neg\phi)$. By Def. 6.10, $w, i \models \neg\mathcal{T}(\phi)$. By Def. 6.3, this implies $w, i \not\models \mathcal{T}(\phi)$. By the counter-positive of (2), we can conclude that $w, i \not\models^{tick} \phi$. By Def. 6.7, this leads to $w, i \models^{tick} \neg\phi$.

Case: $\psi \vee \phi$
Assume that $w, i \models^{tick} \psi \vee \phi$. By Def. 6.7, this implies $w, i \models^{tick} \psi$ or $w, i \models^{tick} \phi$. By (2), we can conclude that $w, i \models \mathcal{T}(\psi)$ or $w, i \models \mathcal{T}(\phi)$. By Def. 6.3, it leads to $w, i \models \mathcal{T}(\psi) \vee \mathcal{T}(\phi)$. By Def. 6.10, $\mathcal{T}(\psi \vee \phi) = \mathcal{T}(\psi) \vee \mathcal{T}(\phi)$, so $w, i \models \mathcal{T}(\psi \vee \phi)$.

Assume that $w, i \models \mathcal{T}(\psi \vee \phi)$. By Def. 6.10, this implies $w, i \models \mathcal{T}(\psi) \vee \mathcal{T}(\phi)$. By Def. 6.3, $w, i \models \mathcal{T}(\psi)$ or $w, i \models \mathcal{T}(\phi)$. By (3), we obtain $w, i \models^{tick} \psi$ or $w, i \models^{tick} \phi$. By Def. 6.7, this leads to $w, i \models^{tick} \psi \vee \phi$.

Case: $\psi\, U(r)_{tc}\, \phi$
Assume that $w, i \models^{tick} \psi\, U(r)_{tc}\, \phi$. By Def. 6.7, this implies that there exists some $k \geq i$ such that

- $w, i, k \models^{tick}_{\mathcal{R}} r$, and
- $w, i, k \models tc$, and
- $w, k+1 \models^{tick} \phi$, and
- for all $j : i \leq j \leq k$ $w, j \models^{tick} \psi$ holds.

By Lemma 6.3, $w, i, k \models^{tick}_{\mathcal{R}} r$ and $w, i, k \models tc$ implies $w, i, k \models_{\mathcal{R}} r'$. Together with (1), $w, k+1 \models^{tick} \phi$ leads to $w, k+1 \models \mathcal{T}(\phi)$. Since $w, j \models^{tick} \psi$ holds for all $j : i \leq j \leq k$, $w, j \models \mathcal{T}(\psi)$ holds by (2) for all $j : i \leq j \leq k$. By Def. 6.7, we can conclude that $w, i \models \mathcal{T}(\psi)\, U(r')\, \mathcal{T}(\phi)$, which leads to $w, i \models \mathcal{T}(\psi\, U(r)_{tc}\, \phi)$ by Def. 6.10.

Assume that $w, i \models \mathcal{T}(\psi\, U(r)_{tc}\, \phi)$. By Def. 6.10, this implies the following: $w, i \models \mathcal{T}(\psi)\, U(r')\, \mathcal{T}(\phi)$. By Def. 6.3, this means that there exists some $k \geq i$ such that

- $w, i, k \models_{\mathcal{R}} r'$, and
- $w, k+1 \models \mathcal{T}(\phi)$, and
- for all $j : i \leq j \leq k$ $w, j \models \mathcal{T}(\psi)$ holds.

By Lemma 6.3, $w, i, k \models_{\mathcal{R}} r'$ implies $w, i, k \models^{tick}_{\mathcal{R}} r$ and $w, i, k \models tc$. Together with (3), $w, k+1 \models \mathcal{T}(\phi)$ leads to $w, k+1 \models^{tick} \phi$. Since $w, j \models \mathcal{T}(\psi)$ holds for all $j : i \leq j \leq k$, $w, j \models^{tick} \psi$ holds by (2) for all $j : i \leq j \leq k$. By Def. 6.7, this leads to $w, i \models^{tick} \psi\, U(r)_{tc}\, \phi$.

From the considered cases, we conclude that $w, 1 \models^{tick} \phi \Longleftrightarrow w, 1 \models \mathcal{T}(\phi)$ for all paths w of T, i.e. $T \models^{tick} \phi \Longleftrightarrow T \models \mathcal{T}(\phi)$. □

6.7 Conclusion

In this chapter we proposed an approach to specification and verification of timed systems within the untimed μCRL framework. The experimental results confirmed the usefulness of the approach.

Timed process algebras can be classified using three criteria. First, whether they use dense or discrete time. Second, whether they use absolute or relative time. Third, whether they use time progression constructs or time stamping of actions. For example, timed μCRL [76] uses absolute time, time stamping of actions and leaves the choice between dense and discrete time open. Several versions of process algebra ACP with time have been studied (e.g. [11, 10]). These algebras use an operator σ to express time progression rather than an action. For example, the process $\sigma(P)$ in ACP with discrete relative time (ACP$_{\mathrm{drt}}$ [10]) is intuitively the same as the process `tick`.P in μCRL with the `tick`-convention. For theoretical work the σ operator is more convenient. For tool support the `tick` action is easier because we do not need to implement the operator and can stay within the μCRL framework.

The use of the `tick` action results in a time semantics which is similar to the semantics used in others tools, such as *DTSpin* [24] and ObjectGeode [132]. However, the input languages of those tools restrict to one particular message passing model, and in μCRL we are free to use whatever model we want. Moreover, *Spin* restricts to *LTL* model checking, while in CADP which serves as a back-end to μCRL we can use the regular alternation-free μ-calculus.

It will be interesting to find out if the framework presented in this chapter can be extended to provide tool support for timed μCRL. Another research topic is the development of time-specific optimization techniques, such as a `tick`-confluence based partial order method.

7

Conclusion

In the introduction of this thesis, we have posed several research questions. In this chapter, we show how we have answered these questions in this thesis.

Modelling time aspects

In Chapter 3, we considered the interpretation of time and timers supported by the commercial SDL design-tools [166, 156] and the standard dynamic semantics of SDL [146]. In this interpretation, system time and the settings of timers are unbounded and timeouts are treated as messages, which leads to infinite systems in the context of enumerative model-checking.

We proposed a transformation that substitutes traditional SDL timers by timer variables. Timers are set not to values expressing a moment of time when the timer should expire, but to values expressing delays left until the timer expiration. This allows to avoid unbounded timer settings, and thus to eliminate the time-related factor leading to infinite systems. To optimize the size of systems further, timeouts are not placed into the input queues but modelled by timeout guards, so we have fewer possible combinations of messages in the input queues.

We proved the path equivalence up to stuttering between the original system and the transformed one, and thus showed that both positive and negative verification results can be safely transferred from the transformed system to the original one for all properties expressible by formulas of *LTL–X*. Each counterexample found in the transformed system can also be found in the original system, and all *LTL–X* formulas satisfied by the transformed system hold on the original one as well. This proof relates the implementation-oriented interpretation of time and timers in SDL to the verification-oriented interpretation of the same concepts in *DTSpin*, which can be considered as an implementation of the "timers as variables" idea, and formally argues the validity of the use of *DTSpin* for the verification of SDL specifications.

Abstracting timers

The correct functionality of reactive systems often depends on time constraints that are modelled by timers. In practice, it can be important to know whether the system works correctly for *all* settings of a timer that satisfy some condition, which is a parameterized problem. Solving this problem by model checking is in general impossible. In a number of cases, we can apply abstractions to obtain a finite-state model for a parameterized system.

In Chapter 4, we considered the time constraints of the form "settings of a timer are larger than or equal to some k". We proposed a timer abstraction that allows to express a family of finite-state systems satisfying such a constraint by a single finite-state system. We showed that the abstract system can safely be used for verification purposes. Any property that can be expressed by a formula of the universal fragment of μ-calculus satisfied on the abstract system also holds on each system of the family.

The timer abstraction turned out to be useful for the verification of a wide range of properties, in particular safety properties. However, checking some liveness properties with the timer abstraction, we encountered false negatives: *DTSpin* reported that the properties are violated and provided counterexample traces that are not present in any of the original systems. To get rid of these traces, we imposed a strong fairness constraint on the abstract system. However, imposing the strong fairness constraint caused a noticeable growth of the state space. Due to the fact that the timer abstraction introduces a self-loop, it was possible to render the strong fairness constraint to a weak fairness constraint. Further, we embedded the weak fairness constraint into the verification algorithm.

The experiments that we performed on PAR and BRP showed that rendering to the weak fairness constraint and further embedding the weak fairness into the verification algorithm are much more efficient in the context of enumerative model checking than imposing the strong fairness constraint. We plan to extend the approach to handle not only the timer abstractions but also more general data abstractions introducing self-loops.

Closing open systems

Model checkers usually do not work with open systems. Therefore, the step that follows decomposing a system into components is *closing* the components with some environment. Closing, when it is done *manually*, is slow and errorprone. In Chapter 5, we provided an approach to the *automatic closing* of open asynchronous timed systems with the most general, *chaotic*, environment. The approach goes beyond [151] in providing a more refined abstraction which gives fewer false negatives in the verification.

The approach involves static analysis, abstraction and program transformation. To close a system, we need to differentiate variable instances (variables at locations) that are definitely *influenced* by the environment, variable instances that are definitely *not influenced* by the environment, and variable instances whose values *depend* on a system run. For this purpose, we combine *may*-analysis marking variables instances potentially influenced by the environment with *must*-analysis marking variable instances definitely influenced by the environment. Further, we abstract the infinity of data from the environment into a single abstract value. For timers, we use a more complex, three-valued, abstraction. Abstracting data coming from environment, we eliminate one factor causing state explosion.

Posterior to the combined analysis and abstraction, we provide a program transformation that closes the system by embedding the behaviour of the environment into the system. The transformation is based on the results of the combined analysis. It removes all manipulations with data that are definitely influenced by the environment. The rest of the data is treated dynamically. Embedding excludes asynchronous communication between the system and its environment and thus eliminates another factor causing state explosion.

Embedding is done in such a way that the closed system shows *more* behaviour than the original one. This claim is justified by a proof showing that for every trace of the original system there is a trace in the closed system with the same stuttering-free projection. It gives us the preservation of the properties expressed by formulas of *LTL–X* mentioning only variables not influenced by the environment in the direction from the closed system to the original one. Therefore, the closed system may *safely* be used for the verification.

We implemented the closing approach as a tool that automatically closes DTPromela translations of SDL specifications. The prototype implementation is based on *may*-analysis only, which gives a less refined abstraction than the one based on the combined analysis. In future, we plan to extend the implementation by the combined analysis. The approach itself will be extended to deal with process creation and complex data types.

Reuse of untimed verification methods for timed verification

The specification language μCRL [78] (micro Common Representation Language) is a process algebraic language that covers both data aspects and behaviour aspects of reactive systems. The μCRL toolset [19] provides support for state space generation, abstraction and optimization prior to enumerative model checking that can be performed with the Cadp toolset [65].

In Chapter 6, we provided a framework that allows to use the existing untimed language and the toolset for timed verification without introducing any syntactical or semantical changes into the language and without modifying the toolset.

We restricted ourselves to relative discrete time. Time progression is modelled as a `tick`-action that represents elapsing one unit of time. A timed parallel composition operator is defined in terms of basic μCRL operators. Time progression has the least priority in the system. This property, called *maximal progress*, is usually expressed by introducing priority operators. We avoid the introduction of new operators providing a special specification discipline that allows us to stay within μCRL syntax and semantics.

The proposed discrete-time semantics is suitable to express time aspects and to analyse time properties of a large class of reactive systems. We justified the usefulness of our approach by the verification experiments on μCRL specifications of the positive acknowledgment retransmission protocol (PAR) [155] and the bounded retransmission protocol (BRP) [100], whose behaviour depends on the timers' settings.

To express not only qualitative but also *quantitative timed* properties of systems, we introduced an *LTL*-like action-based timed temporal logic and showed how to encode its time constraints with the use of `tick`, which results in *untimed* temporal formulas. These formulas can then be translated to the regular alternation free μ-calculus and checked with the μCRL and Cadp toolsets.

The *LTL*-like action-based timed temporal logic together with the specification discipline provide the framework for the timed verification with untimed

μCRL and CADP toolsets. In future, we are interested in applying this framework for the verification of real industrial systems. Another direction for future work is development of time-specific optimization techniques.

References

1. R. Alur. Timed Automata. In *Proc. of CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer-Verlag, 1999.
2. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
3. R. Alur and T. A. Henzinger. Logics and models of real time: A survey. In *Proc. of the Real-Time: Theory in Practice, REX Workshop*, pages 74–106. Springer-Verlag, 1992.
4. R. Alur and T. A. Henzinger. Reactive modules. In *Proceedings of LICS '96*, pages 207–218. IEEE, Computer Society Press, July 1996.
5. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proceedings of the IEEE Symposium on Foundations of Computer Science, Florida*, Oct. 1997.
6. 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, *Proc. of CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 521–525. Springer-Verlag, 1998.
7. K. Apt and D. Kozen. Limits for automatic verification of finite-state systems. *Information Processing Letters*, 15:307–309, 1986.
8. T. Arts and I. A. van Langevelde. Correct performance of transaction capabilities. In *Proc. of 2nd Conference on Applications of Concurrency to System Design (ICACSD'2001), Newcastle upon Tyne, UK*, pages 35–42. IEEE Computer Society Press, 2001.
9. The ATM forum. `http://www.atmforum.com`, 2000.
10. J. C. M. Baeten and J. A. Bergstra. Discrete time process algebra. *Formal Aspects of Computing*, 8(2):188–208, 1996.
11. J. C. M. Baeten and C. A. Middelburg. Process Algebra with Timing: Real Time and Discrete Time. In Bergstra et al. [17].
12. J. C. M. Baeten and W. P. Weijland. Process algebra. *Cambridge Tracts in Theoretical Computer Science*, 18, 1990.
13. T. Basten. Branching bisimilarity is an equivalence indeed! *Information Processing Letters*, 58(3):141–147, 1996.
14. K. Baukus, Y. Lakhnech, and K. Stahl. Verification of parameterized protocols. *Journal of Universal Computer Science*, 7(2):141–158, 2001.
15. G. Behrmann, A. David, K. G. Larsen, O. Möller, P. Pettersson, and W. Yi. UPPAAL - present and future. In *Proc. of 40*th *IEEE Conference on Decision and Control.* IEEE Computer Society Press, 2001.
16. J. A. Bergstra, C. A. Middelburg, and Y. S. Usenko. Discrete time process algebra and the semantics of SDL. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of process algebra*, pages 1209–1268. Elsevier Science BV, 2001.
17. J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra.* Elsevier, 2001.
18. M. Bezem and J. F. Groote. Invariants in process algebra with data. In *Proc. of the Concurrency Theory*, pages 401–416. Springer-Verlag, 1994.
19. S. C. C. Blom, W. J. Fokkink, J. F. Groote, I. A. van Langevelde, B. Lisser, and J. C. van de Pol. μCRL: a toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. of 13th Conference on*

Computer Aided Verification (CAV'01), Paris, France, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer-Verlag, 2001.

20. S. C. C. Blom, N. Ioustinova, and N. Sidorova. Timed verification with μCRL. In M. Broy and A. Zamulin, editors, *Proc. of the 5th Int. Conf. Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2003.
21. S. C. C. Blom and J. C. van de Pol. State space reduction by proving confluence. In E. Brinksma and K. Larsen, editors, *Computer Aided Verification: 14th Int. Conference, CAV 2002 Copenhagen, Denmark, July 2002 Proc.*, volume 2404 of *Lecture Notes in Computer Science*, pages 596–609. Springer Verlag, 2002.
22. D. Bošnački. Partial-order reduction in presence of rendez-vous communication with unless constructs and weak fairness. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th Int. SPIN Workshops*, volume 1680 of *Lecture Notes in Computer Science*. Spriner-Verlag, 1999.
23. D. Bošnački. *Enhancing State Space Reduction Techniques for Model Checking.* PhD dissertation, Eindhoven University of Thechnology, 2001.
24. D. Bošnački and D. Dams. Integrating real time into Spin: A prototype implementation. In S. Budkowski, A. Cavalli, and E. Najm, editors, *Proc. of Formal Description Techniques and Protocol Specification, Testing, and Verification (FORTE/PSTV'98).* Kluwer Academic Publishers, 1998.
25. D. Bošnački, D. Dams, L. Holenderski, and N. Sidorova. Verifying SDL in Spin. In S. Graf and M. Schwartzbach, editors, *TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science.* Springer-Verlag, 2000.
26. D. Bošnački, N. Ioustinova, and N. Sidorova. Using fairness to make abstractions work. In S. Graf and L. Mounier, editors, *Proc. of the 11th Int. Spin Workshop on Model Checking of Software*, volume 2989 of *Lecture Notes in Computer Science*, pages 198–215. Springer, 2004.
27. M. Bozga, J. C. Fernandez, and L. Ghirvu. State space reduction based on Live. In A. Cortesi and G. Filé, editors, *Proc. of SAS '99*, volume 1694 of *Lecture Notes in Computer Science.* Springer-Verlag, 1999.
28. M. Bozga, J. C. Fernandez, L. Ghirvu, S. Graf, J. P. Krimm, and L. Mounier. IF: An intermediate representation and validation environment for timed asynchronous systems. In J. Wing, J. Woodcock, and J. Davies, editors, *Proc. of Symposium on Formal Methods (FM 99)*, volume 1708 of *Lecture Notes in Computer Science.* Springer-Verlag, Sept. 1999.
29. M. Bozga, O. Maler, and S. Tripakis. Efficient verification of timed automata using dense and disrete time semantics. In T. Kropf and L. Pierre, editors, *Proc.of CHARME'99*, volume 1703 of *Lecture Notes in Computer Science*, pages 125–141. Springer, September 1999.
30. M. Broy. Towards a formal foundation of the Specification Description Language SDL. *Formal Aspects of Computing*, 3:21–57, 1991.
31. M. Broy, F. Diderichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. Thedesign of distributed systems - an introduction to FOCUS. Technical Report TUM-19202-2, Institut für Informatik Technische Universität München, 1993.
32. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.*, C-35(8):677–691, Aug. 1986.
33. Bs 7925-2, Software Testing. Software Component Testing. BCS SIGIST, 1998.

34. J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. of the Int. Congress On Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1960.
35. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proc. of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.
36. Y. Choueka. Theories of automata on ω-tapes: a simplified approach. *Journal of Computer and System Science*, 8:117–141, 1974.
37. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.
38. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic specifications. In D. Kozen, editor, *Proc. of the Workshop on Logic of Programs 1981*, volume 131 of *Lecture Notes in Computer Science*, pages 244–263. Springer-Verlag, 1982.
39. E. M. Clarke, E. A. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
40. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Int. Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*. Springer, 2000.
41. E. M. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994. A preliminary version appeared in the Proceedings of POPL 92.
42. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
43. E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, Dec. 1996. Available also as Carnegie Mellon University technical report CMU-CS-96-178.
44. C. Colby, P. Godefroid, and L. J. Jagadeesan. Automatically closing of open reactive systems. In *Proc. of 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 1998.
45. 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.
46. P. Cousot and R. Cousot. Abstract Interpretaion: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL '73*. ACM, January 1973.
47. D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD dissertation, Eindhoven University of Thechnology, July 1996.
48. D. Dams. Abstraction in software model checking: Principles and practice (tutorial overview and bibliography). In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, pages 14–21. Springer-Verlag, 2002.
49. D. Dams and R. Gerth. The bounded retransmission protocol revisited. *Electronic Notes in Theoretical Computer Science*, 9, 1999.
50. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems: Abstraction preserving $\forall$CTL*,$\exists$CTL*, and CTL*. In E.-R. Olderog, editor, *Proc. of PROCOMET '94*. IFIP, North-Holland, June 1994.

51. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(2), 1997.
52. P. R. D'Argenio, J. P. Katoen, T. C. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In *Proc. of the Third Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 416–431. Springer-Verlag, 1997.
53. B. A. Davey and H. A. Pristley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.
54. P. F. G. Dechering and I. A. van Langevelde. The verification of coordination. In A. Porto and G. C. Roman, editors, *Proc. of 4th Conference on Coordination Languages and Models (COORDINATION'2000)*, volume 1906 of *Lecture Notes in Computer Science*, pages 335–340. Springer-Verlag, 2000.
55. Discrete-time Spin. `http://win.tue.nl/~dragan/DTSpin.html`, 2000.
56. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on Software engineering*, pages 411–420. IEEE Computer Society Press, 1999.
57. M. B. Dwyer and J. Hatcliff. Slicing software for model construction. In *Proc. of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, Jan. 1999.
58. M. B. Dwyer and C. S. Pasareanu. Filter-based model checking of partial systems. In *Proc. of the 6th ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT '98)*, pages 189–202, 1998.
59. M. B. Dwyer and D. Schmidt. Limiting state explosion with filter-based refinement. In *Proc. of the 1st International Workshop in Verification, Abstract Interpretation, and Model Checking*, Oct. 1997.
60. E. A. Emerson. Temporal and modal logic. In J. van. Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 995–1072. Elsevier, 1990.
61. E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronisation skeletons. *Science of Computer Programming*, 2:241–266, 1982.
62. E. A. Emerson and J. Y. Halpern. Sometimes and not never revisited: On branching versus linear time. *Journal of the Association on Computing Machinery*, 33(1):151–178, 1986.
63. E. A. Emerson and C.-L. Lei. Modalities for model checking:Branching time strikes back. *Science of Computer Programming*, 8:275–306, 1987.
64. R. Eschbach, U. Glässer, R. Gotzhein, and A. Prinz. On the formal semantics of SDL-2000: a compilation approach based on an abstract SDL machine. In Y. Gurevich, editor, *Proc. ASM 2000*, volume 1912 of *Lecture Notes in Computer Science*, pages 242–265, 2000.
65. J. C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In *Proc. of the 8th Conference on Computer-Aided Verification (New Brunswick, New Jersey, USA)*, pages 437–440, 1996.
66. J. Fischer and E. Dimitrov. Verification of SDL protocol specifications using extended petri nets. In *Proc. of the Workshop on Petri Netx and Protocols of the 16th Intern. Conf. on Application and Theory of Petri Nets*, pages 1–12, 1995.

67. W. J. Fokkink, N. Ioustinova, E. Kesseler, J. C. van de Pol, Y. Usenko, and Y. A. Yushtein. Refinement and verification applied to an in-flight data acquisition unit. In L. Brim, P. Jancar, M. Kretinsky, and A. Kucera, editors, *Proc. of 13th Conference on Concurrency Theory - CONCUR'02, Brno*, volume 2421 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2002.
68. N. Francez. *Fairness*. Springer-Verlag New York, Inc., 1986.
69. M. M. Gallardo, J. Martine, P. Merino, and E. Pimentel. αSPIN: Extending SPIN with abstraction. In *Proc. of 9th Int. SPIN Workshop, Grenoble, France 2002*, volume 2318 of *Lecture Notes in Computer Science*, pages 254–258, 2002.
70. R. J. van Gabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, 1996.
71. R. J. van Glabbeek. The linear time - branching time spectrum. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR'90. Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.
72. R. J. van Glabbeek. The linear time - branching time spectrum ii. In *Proc. of CONCUR 93*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81, 1993.
73. R. J. van Glabbeek. What is the branching time semantics and why to use it? In M. Nielsen, editor, *Bulletin of the EATCS 53*, pages 190–198, 1994.
74. P. Godefroid. Using partial orders to improve automatic verification methods. In E. M. Clarke and R. P. Kurshan, editors, *Computer Aided Verification 1990*, volume 531 of *Lecture Notes in Computer Science*, pages 176–449. Springer-Verlag, 1991.
75. S. Graf. Verification of a distributed cache memory by using abstractions. In *Workshop on Computer-Aided Verification, CAV'94, Stanford*, volume 818 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.
76. J. F. Groote. The syntax and semantics of timed μCRL. SEN R9709, CWI, Amsterdam, 1997.
77. J. F. Groote, J. Pang, and A. G. Wouters. A balancing act: Analyzing a distributed lift system. In S. Gnesi and U. Ultes-Nitsche, editors, *Proc. of 6th Workshop on Formal Methods for Industrial Critical Systems (FMICS'2001), Paris, France*, pages 1–12, 2001.
78. J. F. Groote and M. Reniers. Algebraic process verification. In Bergstra et al. [17], pages 1151–1208.
79. J. F. Groote and M. P. A. Sellink. Confluence for process verification. *Theoretical Comput. Sci.*, 170:47–81, 1996.
80. J. F. Groote and J. C. van de Pol. A bounded retransmission protocol for large data packets. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST'96)*, volume 1101 of *Lecture Notes in Computer Science*. Springer, 1996.
81. J. F. Groote and J. C. van de Pol. State space reduction using partial τ-confluence. In M. Nielsen and B. Rovan, editors, *Proc. of MFCS 2000*, volume 1893 of *Lecture Notes in Computer Science*, pages 383–393. Springer, 2000.
82. J. F. Groote and J. J. van Wamel. Analysis of three hybrid systems in timed μCRL. *Science of Computer Programming*, 39:215–247, 2001.
83. R. Hardin, Z. HarEl, and R. P. Kurshan. COSPAN. In R. Alur and T. A. Henzinger, editors, *Proc. of the 1996 Wokshop on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 423–427, 1996.

84. D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts, The STATEMATE Approach.* McGraw-Hill, 1998.
85. M. S. Hecht. *Flow Analysis of Programs.* North-Holland, 1977.
86. M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117:221–239, 1995.
87. T. A. Henzinger, O. Kupferman, and R. Majumdar. On the universal and existential fragments of the mu-calculus. In *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 49–64, 2003.
88. T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In W. Kuich, editor, *ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 545–558. Springer, 1992.
89. T. A. Henzinger, Z. Manna, and A. Pnueli. Temporal proof methodologies for timed transition systems. *Inf. Comput.*, 112(2):273–337, 1994.
90. U. Hinkel. *Formale, semantische Fundierung und eine darauf abgestützte Verifikationsmethode für SDL.* PhD thesis, Tech. Univ. München, 1998.
91. U. Hinkel. Verification of SDL specifications on the basis of stream semantics. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC (SAM'98)*, pages 241–250, 1998.
92. E. Holz and K. Stølen. An attempt to embed a restricted version of SDL as a target language to FOCUS. In S. Leue and D. Hogrefe, editors, *Proc. of Forte'94*, pages 324–339. Chapmann & Hall, 1994.
93. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual.* Addison Wesley, 2003.
94. G. J. Holzmann and J. Patti. Validating SDL specifications: an experiment. In E. Brinksma, editor, *International Workshop on Protocol Specification, Testing and Verification IX (Twente, The Netherlands)*, pages 317–326. North-Holland, 1989. IFIP TC-6 Int. Workshop.
95. G. J. Holzmann and D. Peled. An improvement in formal verification. In D. Hogrefe and S. Leue, editors, *Formal Description Techniques VII, Proc. of the 7th IFIP WG6.1 Int. Conference on Formal Description Techniques, Berne, Switzerland, 1994*, volume 6 of *IFIP Conference Proceedings*. Chapman & Hall, 1995.
96. G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth-first search. In *Second SPIN Workshop*, pages 23–32. AMS, 1996.
97. J. Hooman and J. C. v. d. Pol. Formal verification of replication on a distributed data space architecture. In *Proc. of 17th Symposium on Applied Computing (SAC'2002) -Coordination Models, Languages and Applications*, pages 351–358. ACM Press, 2002.
98. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computations.* Addison-Wesley, 2001.
99. T. Huckle. Kleine BUGs, groe GAUs: Softwarefehler und ihre Folgen. `http://www5.in.tum.de/ huckle/bugs.html`.
100. Infrared remote control system RC6. Philips Consumer Electronics B.V., April 1997.
101. N. Ioustinova and N. Sidorova. Transformation of SDL specifications- a step towards the verification. In D. Bjorner, M. Broy, and A. Zamulin, editors, *Post-proceedings of Andrei Ershov Fourth International Conference Perspectives*

of System Informatics (PSI 01), volume 2244 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2001.

102. N. Ioustinova, N. Sidorova, and M. Steffen. Abstraction and flow analysis for model checking open asynchronous systems. In *Proc. of the 9th Asia Pacific Software Engineering Conference (APSEC 2002)*, pages 227–235. IEEE Computer Society, 2002.
103. N. Ioustinova, N. Sidorova, and M. Steffen. Closing open SDL-systems for model checking with DTSpin. In L. H. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods - Getting IT Right, Proc. of Int. Symposium of Formal Methods Europe, FME 2002*, volume 2391 of *Lecture Notes in Computer Science*, pages 531–548. Springer, 2002.
104. N. Ioustinova, N. Sidorova, and M. Steffen. Synchronous closing and flow abstraction for model checking timed systems. In *Proc. of the Second Int. Symposium on Formal Methods for Components and Objects (FMCO'03)*, volume (to appear) of *Lecture Notes in Computer Science.* Springer, 2004.
105. Integrated services digital networks (ISDN). ITU-I, 2000.
106. R. Kaivola. Using compositional preorders in the verification of sliding window protocol. In *Proceedings of 9th International Conference on Computer Aided Verification (CAV'99)*, volume 1663 of *Lecture Notes in Computer Science*, pages 184–195, 1999.
107. Y. Kesten and A. Pnueli. Modularization and abstraction: the keys to practical formal verification. In L. Brim, J. Gruska, and J. Zlatuska, editors, *Proc. of the 23rd Int. Symposium on Mathematical Foundations of Computer Science*, pages 54–71, 1998.
108. Y. Kesten and A. Pnueli. Control and data abstraction: The cornerstones of practical formal verification. *Int. Journal on Software Tools for Technology Transfer*, 2(4):328–342, 2000.
109. Y. Kesten and A. Pnueli. Verification by augmented finitary abstraction. *Information and Computation*, 163(1):203–243, 2000.
110. Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1998.
111. G. Kildall. A unified approach to global program optimization. In *Proc. of POPL '73*, pages 194–206. ACM, January 1973.
112. D. Kozen. Results on the propositional μ-calculus. *Journal of Theoretical Computer Science*, 27:333–354, 1983.
113. S. Kripke. A semantical analysis of modal logic i: normal modal propositional caculi. In *Zeitschrift fuer Mathematische Logik und Grundlagen der Mathematik*, volume 9, pages 67–96, 1963.
114. R. Kuiper and W. P. de Roever. Fairness assumptions for CSP in a temporal logic framework. In D. Bjorner, editor, *Proc. of the IFIP Working Conference on Formal Description of Programming Concepts–II*, pages 159–170. North-Holland Publishing Company, 1983.
115. O. Kupferman and M. Y. Vardi. Module checking revisited. In O. Grumberg, editor, *CAV '97, Proc. of the 9th Int. Conference on Computer-Aided Verification, Haifa. Israel*, volume 1254 of *Lecture Notes in Computer Science.* Springer, June 1997.
116. O. Kupferman, M. Y. Vardi, and P. Wolper. Module checking. In R. Alur, editor, *Proc. of CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 75–86, 1996.

117. L. Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83*, pages 657–668. Elsevier Science Publishers B.V., 1983.
118. L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions in Computer Systems*, 5(1):1–11, 1987.
119. K. Larsen, P. Peterson, and W. Yi. UPPAAL in the nutshell. *Software Tools for Technology Transfer*, 1(1):134–152, 1997.
120. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Twelfth Annual Symposium on Principles of Programming Languages (POPL) (New Orleans, LA)*, pages 97–107. ACM, 1985.
121. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.
122. D. Long. *Model Checking, Abstraction and Compositional Verification.* PhD thesis, Carnegie Mellon University, 1993.
123. B. D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs i. *Acta Inf.*, 21:125–169, 1984.
124. K. L. MacMillan. *Symbolic model checking:an approach to the state space explosion problem.* PhD thesis, Carnegie Mellon University, 1992.
125. Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety.* Springer-Verlag New York, Inc., 1995.
126. R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular aletrnation-free mu-calculus. In *Proceedings of the 5th International Workshop on Formal Methods for Industrial Critical Systems, FMICS'2000*, 2000.
127. S. Merz. *Model checking: a tutorial overview.* Springer-Verlag New York, Inc., 2001.
128. L. I. Millet and T. Teitelbaum. Slicing Promela and its application to model checking, simulation, and protocol understanding. In E. Najm, A. Serhrouchni, and G. Holzmann, editors, *Electronic Proc. of the Fourth Int. SPIN Workshop, Paris, France*, Nov. 1998.
129. R. D. Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM(JACM)*, 42(2):458–487, 1996.
130. X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Proc. of the Real-Time: Theory in Practice, REX Workshop*, pages 526–548. Springer-Verlag, 1992.
131. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.
132. ObjectGeode 4.0. `http://www.csverilog.com/products/geode.htm`, 2003.
133. A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J. R. W. Smith. *System Engineering Using SDL-92.* Elsevier Science, 1997.
134. S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.
135. D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390. Springer-Verlag, 1994.
136. W. Penczek, M. Szreter, R. Gerth, and R. Kuiper. Improving partial order reductions for universal branching time properties. *Fundamenta Informaticae*, 43(1-4):245–267, 2000.

137. A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.
138. A. Pnueli, J. Xu, and L. Zuck. Liveness with $(0, 1, \infty)$-counter abstraction. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification : 14th Int. Conference, CAV 2002, Copenhagen, Denmark, July 27-31, 2002. Proc.*, volume 2404 of *Lecture Notes in Computer Science*, pages 107 – 122. Springer, 2002.
139. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proc. of the 5th Int. Symposium on Programming 1981*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1982.
140. Specification and Description Language SDL. CCITT, 1993.
141. SDL combined with UML. ITU-T, 1999.
142. Specification and Description Language SDL. ITU-T, 1999.
143. SDL formal definition: Static Semantics. ITU-T, 1993.
144. SDL formal definition: Static Semantics. ITU-T, 2000.
145. SDL formal definition: Dynamic Semantics. ITU-T, 1993.
146. SDL formal definition: Dynamic Semantics. ITU-T, 2000.
147. F. Regensburger and A. Barnard. Formal verification of SDL systems at the Siemens mobile phone department. In B. Steffen, editor, *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 439–455. Springer, 1998.
148. W. P. d. Roever, H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Difference, Proceedings of the International Symposium COMPOS '97, Malente, Germany, September 7–12, 1997*, volume 1536 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.
149. J. Rushby. Theorem proving for verification. In F. Cassez, C. Jard, B. Rozoy, and M. D. Ryan, editors, *Modelling and Verification of Parallel Processes: MOVEP 2000*, number 2067 in Lecture Notes in Computer Science, pages 39–57, Nantes, France, June 2000. springer Verlag.
150. S. Schneider. *Concurrent and Real-Time Systems: The CSP Approach.* Wiley, 2000.
151. N. Sidorova and M. Steffen. Embedding chaos. In P. Cousot, editor, *Proc. of the 8th Static Analysis Symposium (SAS'01)*, volume 2126 of *Lecture Notes in Computer Science*, pages 319–334. Springer-Verlag, 2001.
152. N. Sidorova and M. Steffen. Verifying large SDL-specifications using model checking. In R. Reed and J. Reed, editors, *Proc. of 10th Int. SDL-Forum, Copenhagen, Denmark*, volume 2078 of *Lecture Notes in Computer Science*, pages 399–416. Springer, June 2001.
153. N. Sidorova and M. Steffen. Synchronous closing of timed SDL systems for model checking. In A. Cortesi, editor, *Proc. of the Third Int. Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI) 2002*, volume 2294 of *Lecture Notes in Computer Science*, pages 79–93. Springer-Verlag, 2002.
154. Spin. `http://www.spinroot.com`.
155. A. S. Tanenbaum. *Computer Networks.* Prentice Hall International, Inc., 1981.
156. Telelogic Malmö AB. *SDT 3.1 User Guide, SDT 3.1 Reference Manual.* Telelogic, 1997.
157. Telelogic TAU SDL Suite. http://www.telelogic.com/products/sdl/, 2003.
158. W. Thomas. Automata on infinite words. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 133–191. Elsevier, 1990.

159. F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.
160. H. Tuominen. Embedding a dialect of SDL in Promela. In *Proc. of 6th Int. SPIN Workshop*, volume 1680 of *Lecture Notes in Computer Science*. Springer, 1999.
161. K. J. Turner. *Using Formal Description Techniques: An Introduction to Estelle, Lotos, and SDL*. John Wiley & Sons, Inc., 1993.
162. Y. S. Usenko. *Linearization in μCRL*. PhD thesis, Technische Unversiteit Eindhoven, 2002.
163. A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design*, 1992. Earlier version in the proceeding of CAV '90 Lecture Notes in Computer Science 531, Springer-Verlag 1991, pp. 156–165 and in Computer-Aided Verification '90, DIMACS Series in Discrete Mathematics and Theoretical Computer Science Vol. 3, AMS & ACM 1991, pp. 25–41.
164. J. C. van de Pol and M. Valero Espada. Formal specification of JavaSpaces architecture using μCRL. In F. Arbab and C. L. Talcott, editors, *Proc. of 5th Conference on Coordination Languages and Models (COORDINATION'2002)*, volume 2315 of *Lecture Notes in Computer Science*, pages 274–290. Springer-Verlag, 2002.
165. M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 15 1994.
166. Verilog. *ObjectGEODE SDL Simulator - Reference Manual*, 1996.
167. Verifying industial reactive systems (VIRES), Esprit long-term research project LTR-23498. `http://radon.ics.ele.tue.nl/~vires/`, 1998-2000.
168. W. Visser and H. Barringer. Practical CTL * model checking: Should SPIN be extended? *International Journal on Software Tools for Technology Transfer*, 2(4):350–365, 2000.
169. A wireless ATM network demonstrator (WAND), ACTS project AC085. `http://www.tik.ee.ethz.ch/~wand/`, 1998.
170. Y. Wang. Real-time behaviour of asynchronous agents. In J. C. M. Baeten and J. W. Klop, editors, *Theories of concurrency: unification and extension(CONCUR'90)*, volume 458 of *Lecture Notes in Computer Science*, 1990.
171. P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. of 13th ACM Symp. on Principles of Programming Languages*, pages 184–192, St. Petersburgh, January 1986.
172. ITU-T Recommendation X.291-ISO/IEC 9646-2, Information Technology- Open Systems Interconnection- Conformance Testing Methodology and Framework-Part 2: Abstract Test Specifications.

Summary

In this thesis, we present a number of techniques facilitating the verification of reactive systems. A well-established formal technique for the verification of reactive systems is *model checking*, which is recognized both by industry and by the academic community. As model checking is based on state space exploration, the stumbling block limiting the applicability of model checking is the state explosion problem. Techniques presented in this work were developed to alleviate this problem; they combine abstraction, static analysis and program transformation.

Often, reactive systems are timed systems, and timers are used to express timed constraints imposed on a system. Interpretations of time and time constraints in specification languages used by industry are mainly implementation-oriented and unsuitable for verification purposes. SDL is a vivid representative of the class of implementation-oriented languages, where time is modelled by infinitely an growing variable and timeouts are treated as messages.

We propose a transformation that substitutes traditional SDL timers by timer variables. The transformation allows to avoid unbounded time settings and to optimize the size of the system by modelling timeouts as timeout guards. We justify that for verification purposes timers as variables can be used instead of traditional timers. We prove that original and transformed systems are related by path equivalence up to stuttering, which guarantees the preservation of both positive and negative verification results for properties that are expressible by formulas of the temporal logic *LTL–X*.

The concept of timers as variables can be successfully used for timed verification using untimed verification frameworks that do not take time into account. As an example, we take the μCRL framework that provides a language for the specification of reactive systems and a toolset for the generation and optimisation of state spaces. This framework is especially developed to take data into account. We propose a specification discipline that allows to use the untimed toolset and the untimed specification language for timed verification without introducing new constructs into the language and without modifying the toolset. We also introduce an *LTL*-like action-based timed temporal logic. The formulas of this logic can be translated into the regular alternation-free μ-calculus and model checked with the CADP toolset. Thus we obtain a powerful framework for timed verification that covers both time and data aspects of reactive systems.

In a number of practical examples, properties of reactive systems are expected to hold for all settings of timers satisfying a certain condition. A typical example of such a condition is “for all settings of a timer larger than or equal to some k”. Checking whether a property holds for all systems where settings of a timer satisfy this condition would require an infinite number of iterations, and

thus solving this problem by model checking directly is impossible. We propose a timer abstraction that allows to represent an infinite family of finite state systems satisfying this condition by a single finite state system. We also show that properties that can be expressed by formulas of the universal fragment of the μ-calculus are preserved in the direction from the abstract system to the original one. This timer abstraction appears to be useful for the verification of a wide range of properties. However, it can give rise to false negatives when liveness properties are verified. The problem can be resolved by imposing a strong fairness condition on the abstract system. Imposing this strong fairness condition leads to a substantial growth of the state space. For the timer abstraction, we prove that the strong fairness condition can be brought down to a weak fairness condition. We demonstrate that the weak fairness condition can be built into the model checking algorithm implemented in *Spin*. Using the built-in weak fairness is much more efficient than using the strong fairness condition.

Compositional verification is one of the approaches used to cope with state explosion. A system is decomposed into components that can be checked separately. Since model checkers usually do not work with open systems, the components should be closed prior to model checking. Manual closing is error-prone and time-consuming. We provide automatic closing of open systems with the most general, chaotic, environment. Closing involves static analysis, abstraction and program transformation. We propose a combination of may- and must-analyses that marks variables at each location of a system specification as definitely influenced by the environment, or as definitely not influenced by the environment, or as "don't know" variables whose values at a location depend on a run. The data coming from the environment are abstracted into a single abstract value. For timers, we use a more complex three-valued abstraction. A program transformation, which follows the combined may/must analysis, removes the manipulations on data that are definitely influenced by the environment. The manipulations on data that are definitely not influenced by the environment are left unmodified. The manipulations on "don't know" data are treated dynamically in the transformed system. Abstracting from data coming from the environment eliminates one factor causing state explosion. Another factor leading to state explosion is asynchronous communication with the environment. The transformation removes it by embedding the environment into the system. We show that there is path inclusion up to stuttering between the closed and the original open system. This guarantees the transfer of positive verification results from the closed system to the original open one for all properties that can be expressed by *LTL–X* formulas mentioning only variables not influenced by the environment.

All techniques presented in this thesis have been implemented. For each of the developed approaches, we have performed a number of experiments confirming their usefulness.

Samenvatting

In dit proefschrift hebben wij een aantal technieken gepresenteerd die helpen bij de verificatie van reactieve systemen. Een gerenommeerde formele techniek voor het verifiëren van reactieve systemen is model checking, gebruikt door zowel bedrijven als de academische wereld. Aangezien model checking gebaseerd is op het onderzoeken van toestandsruimtes, is het struikelblok voor de toepasbaarheid van model checking het probleem van explosie van toestandsruimtes. De technieken gepresenteerd in dit proefschrift werden ontwikkeld om dit probleem te verlichten. Zij combineren abstractie, statische analyse en programma-transformatie.

Vaak zijn reactieve systemen ook systemen met tijd, en worden timers gebruikt om de tijdsbeperkingen uit te drukken die het systeem worden opgelegd. Interpretaties van tijd en tijdsbeperkingen in specificatietalen gebruikt bij bedrijven zijn voornamelijk georiënteerd op implementatie en ongeschikt voor verificatiedoeleinden. SDL is een duidelijke vertegenwoordiger van de klasse van implementatiegeoriënteerde talen waarin de tijd gemodelleerd wordt met een oneindig groeiende variabele en waarin timeouts als berichten worden behandeld.

Wij hebben een transformatie voorgesteld die traditionele SDL-timers vervangt door timervariabelen. De transformatie maakt het mogelijk om onbegrensde tijdswaarden te vermijden en de grootte van het systeem te optimaliseren door timeouts als timeout condities te modelleren. Wij hebben gerechtvaardigd dat voor verificatiedoeleinden timers als variabelen gebruikt kunnen worden in plaats van traditionele timers. Wij hebben bewezen dat de originele en getransformeerde systemen verbonden zijn door "path equivalence up to stuttering". Dit garandeert dat zowel positieve als negatieve verificatieresultaten behouden blijven voor eigenschappen die uitgedrukt kunnen worden door formules in de temporele logica *LTL–X*.

Het concept van timers als variabelen kan met succes worden benut voor verificatie met tijd, door gebruik te maken van raamwerk voor verificatie die geen rekening houden met tijd. Als voorbeeld hebben we het μCRL raamwerk genomen, dat een taal voor de specificatie van reactieve systemen en tools voor optimalisatie van toestandsruimtes biedt. Dit raamwerk is in het bijzonder ontwikkeld om met data te kunnen omgaan. Wij hebben een specificatiemethode beschreven die het mogelijk maakt om de toolset zonder tijd en de specificatietaal zonder tijd te gebruiken voor verificatie met tijd, zonder nieuwe constructies in de taal te introduceren en zonder de toolset aan te passen. Wij hebben eveneens een *LTL*-achtige temporele logica geïntroduceerd die op acties gebaseerd is en rekening houdt met tijd. De formules van deze logica kunnen vertaald worden in de reguliere alternatie-vrije μ-calculus en met de CADP

toolset worden gecontroleerd. Zo hebben wij een krachtig raamwerk verkregen dat om kan gaan met zowel tijd als data in reactieve systemen.

In een aantal praktische voorbeelden worden de eigenschappen van reactieve systemen geacht geldig te blijven voor alle instellingen van timers die aan een bepaalde conditie voldoen. Een typisch voorbeeld van zo'n conditie is "voor alle instellingen van een timer die groter dan of gelijk aan een bepaalde k zijn". Om te controleren of een eigenschap geldig blijft voor alle systemen waar de instellingen van een timer aan deze conditie voldoen is een oneindig aantal iteraties nodig, en dus is het onmogelijk dit probleem direct op te lossen met model checking. Wij hebben een timerabstractie gegeven die het mogelijk maakt een oneindige familie van finite-state systemen die aan deze conditie voldoen te representeren met een enkel finite-state systeem. Wij hebben ook aangetoond dat eigenschappen die uitgedrukt kunnen worden door formules uit het universele fragment van de μ-calculus behouden worden in de richting van het abstracte systeem naar het originele.

De timerabstractie is nuttig gebleken voor de verificatie van een groot scala van eigenschappen. Het kan echter onterechte foutmeldingen veroorzaken wanneer liveness-eigenschappen geverifieerd worden. Het probleem kan worden opgelost door een "strong fairness" conditie aan het abstracte systeem op te leggen. Het opleggen van deze "strong fairness" conditie leidt tot een aanzienlijke groei van de toestandsruimte. Voor de timerabstractie hebben we aangetoond dat de "strong fairness" conditie teruggebracht kan worden tot een "weak fairness" conditie. Wij hebben aangetoond dat de "weak fairness" conditie ingebouwd kan worden in het model checking algoritme dat verwezenlijkt is in Spin. Het gebruiken van de ingebouwde "weak fairness" is veel efficiënter dan het gebruiken van "strong fairness".

Compositionele verificatie is een van de aanpakken die gebruikt worden voor het omgaan met explosie van toestandsruimtes. Een systeem wordt ontleed in componenten die los van elkaar gecontroleerd kunnen worden. Aangezien model checkers gewoonlijk niet met open systemen werken, moeten de componenten afgesloten worden voorafgaand aan model checking. Handmatig afsluiten is ontvankelijk voor fouten en tijdverslindend. Wij hebben een automatische afsluiting gegeven van open systemen met de meest algemene, de chaotische, omgeving. De afsluiting omvat statische analyse, abstractie en programma transformatie. Wij hebben een combinatie voorgesteld van may- en must-analyse die variabelen op iedere locatie van een systeemspecificatie markeert als zeker beïnvloed door de omgeving, als zeker niet beïnvloed door de omgeving of als "weet niet"-variabelen waarvan de waarde op een locatie afhangt van een executie. De gegevens die van de omgeving komen worden geabstraheerd in een enkele abstracte waarde. Voor timers gebruiken we een complexere, driewaardige abstractie.

Een programma-transformatie, die de gecombineerde may- and must-analyse volgt, verwijdert de bewerkingen op data die zeker beïnvloed zijn door de omgeving. De bewerkingen op data die zeker niet beïnvloed zijn door de omgeving worden onveranderd gelaten. De bewerkingen op de "weet niet"-data worden dynamisch behandeld in het getransformeerde systeem. Het abstraheren van data uit de omgeving elimineert een factor die explosie van toestandsruimtes veroorzaakt. Een andere factor die leidt tot explosie van toestandsruimtes is asynchrone communicatie met de omgeving. De transformatie verwijdert dit door de omgeving in het systeem vast te leggen. We hebben aangetoond dat er path inclusion up to stuttering bestaat tussen afgesloten en originele open systemen. Dit garandeert de overdracht van positieve verificatieresultaten van afgesloten naar open systemen voor alle eigenschappen die uitgedrukt kunnen worden in *LTL–X* formules met alleen variabelen die niet beïnvloed zijn door de omgeving.

Alle technieken gepresenteerd in dit proefschrift zijn geïmplementeerd. Voor elk van de ontwikkelde aanpakken hebben we een aantal experimenten uitgevoerd die hun nut bevestigen.