ISSN 1397–8640

AALBORG UNIVERSITY
INSTITUTE FOR COMPUTER SCIENCE

Fredrik Bajers Vej 7E, 9220 Aalborg Øst, Denmark

Abstraction-Based Verification of Distributed Systems

PhD thesis

by

Henrik Ejersbo Jensen

Supervisor: Kim G. Larsen

June 1999

Pages: 182
Number of Copies: 100

Abstract

This thesis presents abstraction-based proof methods and practical abstraction strategies to support the integration of theorem proving and model checking methods in verification of distributed systems. The thesis is in two parts. In the first part we present abstraction frameworks for untimed systems described as I/O automata and for real-time systems described as timed automata. The frameworks provide general conditions for preservation of properties from concrete systems to abstract ones. For the I/O automaton model we present preservation conditions for safety and liveness properties stated over actions as well as over states. The preservation conditions are based on simulation relations. The abstraction theory is formalized using the Larch theorem prover and a scheme for translating I/O automata in to the SPIN model checker is examined. For the timed automaton model we provide preservation conditions based on requirements stated as automaton specifications with a satisfaction relation in the form a timed ready simulation relation. Our preservation conditions are based on an action parameterized variant of this simulation relation. The timed abstraction framework is stated in the input language of the UPPAAL model checker for real-time systems providing a close link to automatic verification. In the second part of this thesis we provide abstraction-based proofs for three nontrivial distributed algorithms all parameterized in the number of processes: Burns' Mutual Exclusion algorithm, The Bounded Concurrent Timestamp System (BCTSS) algorithm, and Fischer's Real-Time Mutual Exclusion algorithm. The proof of Burns' algorithm utilizes an abstraction strategy based on skolemization and the proof is carried out by support from the Larch Prover and the SPIN model checker. The proof of the BCTSS algorithm is the most advanced in this thesis. The BCTSS algorithm is one of the most complicated algorithms in the distributed systems literature and existing proofs are all long and hard to understand. Our abstraction proof exploits a combination of induction and abstraction strategies to delegate major proof tasks to automatic verification in the SPIN model checker. The proof of Fischer's algorithm utilizes a combination of compositionality and abstraction strategies based on network invariants. The UPPAAL model checker is used to verify the constructed abstraction.

Dansk Sammenfatning

Denne afhandling præsenterer abstraktionsbaserede bevismetoder og praktiske abstraktionsteknikker til understøttelse af en integration af deduktive og automatiske metoder til verifikation af distribuerede systemer. Afhandlingen er i to dele. I den første del præsenterer vi abstraktionsmetoder for systemer beskrevet som I/O automater og for realtids systemer beskrevet som tidsautomater. Metoderne giver generelle betingelser for bevarelse af egenskaber fra konkrete tilstandssystemer til abstrakte tilstandssystemer. For I/O automater gives betingelser for bevarelse af safety og liveness egenskaber udtrykt over handlinger såvel som over tilstande. Disse betingelser er baseret på simuleringsrelationer. Vores abstraktionsteori er formaliseret ved brug af Larch theorem prover, og en overordnet metode for oversættelse af I/O automater til SPIN model checkeren undersøges. For tidsautomater giver vi betingelser for bevarelse af egenskaber baseret på krav opstillet som automatspecifikationer og med en tilfredsstillelsesrelation i form af en tidssimuleringsrelation. Vores betingelser er baseret på en handlingsparametriseret variant af tidssimuleringen. Abstraktionsmetoden er givet for input-sproget som bruges af UPPAAL model checkeren for realtids systemer. Dette giver et direkte link til automatisk verifikation. I den anden del af afhandlingen præsenterer vi abstraktionsbaserede beviser for tre ikke-trivielle distribuerede algoritmer som alle er parametriseret i antallet af processer: Burns' Mutual Exclusion algoritme, Bounded Concurrent Timestamp System (BCTSS) algoritmen samt Fischers Mutual Exclusion algoritme. Beviset for Burns algoritme benytter en abstraktionsstrategi baseret på skolemisering og beviset udføres med støtte fra Larch Prover og SPIN model checkeren. Beviset for BCTSS algoritmen er det mest avancerede i denne afhandling. BCTSS algoritmen er en af de mest komplicerede algoritmer i litteraturen om distribuerede systemer og eksisterende beviser er alle lange og svært forståelige. Vores abstraktionsbevis udnytter en kombination af induktion- og abstraktionsteknikker til at delegere store bevisbyrder til automatisk håndtering i SPIN model checkeren. Beviset for Fischers algoritme benytter en kombination af kompositionalitet og abstraktion baseret på anvendelse af netværks-invarianter. UPPAAL model checkeren anvendes til at verificere den konstrurerede abstraktion.

Acknowledgments

First of all, I would like to thank Kim Larsen, my supervisor, and Nancy Lynch, my host during a two year stay at M.I.T., for all their wonderful support, suggestions and comments that made this thesis possible. I am grateful to have enjoyed their constant enthusiasm and encouragements.

The Theory of Distributed Systems group at M.I.T. has been a great place to visit. I spent two years of my Ph.D. studies there and I have greatly enjoyed the pleasant environment for doing research. I owe thanks to everyone in the group as well as to my many roommates over time. My stay has been of great value to me and I hope to visit M.I.T. and the Boston area again in the future.

At Aalborg University I would like to thank everyone in the Distributed Systems and Semantics group for making this a great place to work. Although I have spent most of my time during my Ph.D.-studies away from Aalborg I have always felt welcome here. I look forward to further collaborations in the future.

My family and friends have always been there for encouragement and support. To my wife and best friend, Helle: Thanks for your love and constant support through all this work, for always listening to me and for keeping me sane even when things seemed bleak.

Contents

Chapter 1

Introduction

1.1 Motivation

Embedded Systems. Software systems that form an integral part of the physical environment in which they operate are called *embedded systems*. Such systems arise in a still increasing number of application domains, ranging from telecommunications and aircraft control to consumer electronics in toys and minor appliances. Embedded systems continuously interact with their environment by monitoring events through sensors and reacting accordingly via actuators. Therefore, embedded systems are often said to be *reactive*, in contrast to the traditional view of a program as something that takes an input, produces an output and then terminates. Many embedded systems are designed for implementation on a physical platform consisting of a set of interconnected processors. Such embedded systems are in addition said to be *distributed*.

Common to embedded systems is that they very often are safety critical, in the sense that failures may have catastrophic effects such as loss of human lives and vast economical loses. Hence *correctness* is one of the most important quality factors of such systems. Ensuring correctness has traditionally been performed by means of *testing* techniques where a group of people are set to work with early releases of programs to report on errors when encountered. However, when considering safety critical embedded systems this approach is no longer satisfactory as a single means of system validation. Consider for example an air-traffic control system to be implemented on board an aircraft, with the purpose of alerting the aircrafts pilot of any "too-close" neighboring aircrafts. It will most certainly be difficult to find any group of people that would undertake the jobs as "test-pilots" on beta-releases of such a software system. Testing methods exist that generate sets of test-suites in a structured way trying to explore or *simulate* as many system behaviours as possible. However, *exhaustive* testing is usually infeasible due to the vast (possibly infinite) set of behaviours of an embedded system. Also, by the testing approach errors are not discovered until very late in the systems development process. This also applies to logical design errors which it would be more natural to catch before any implementation takes place.

Formal Methods. One way to complement the testing approach in order to overcome the above problems is to make use of *formal methods*. This term covers all approaches to specification and verification based on mathematical formalisms aiming to establish program correctness by mathematical rigour. Any formal method consists of three basic ingredients: a *modelling language* to describe systems, a *specification language* to state system requirements, and a *verification methodology* to establish a formal correctness relation between a system model and a corresponding specification. The modelling language provides a mathematically precise behavioural model of systems. The language is typically in the form of some kind of *state-transition system* consisting of a set of *states* and a binary relation on this set describing a set of *transitions*. States represent points in a systems behaviour (values of variables and program counter) and transitions describe state changes (execution of statements). For a given system its set of possible behaviours or *executions* are taken to be the set of all the possible transition sequences satisfying some given initial requirement. Transition systems are usually described syntactically through some (first-order) logical language where states and transitions are described using formulas.

Verification Methodologies. Formal verification methods can be classified as either *theorem proving methods* or *model checking methods*. Theorem proving methods are based on proving general mathematical theorems about systems using formal deduction in a *proof system* consisting of a set of axioms and inference rules. The rules of the proof system are used to infer properties of system models based on their syntactical (logical) description. Theorem proving methods require intelligent user interaction in the sense that a thorough understanding of the system to be proved as well as the mathematical proof system is required in order to establish the right theorems and neccessary additional lemmas and to do the proofs. As a result theorem proving methods are mainly manual or computer-assisted in a limited way. Tools supporting theorem proving methods are denoted as *theorem provers*. There is in general no restrictions on the class of models and properties being amenable to theorem proving methods, and the insight gained by the user into the behaviour of the model at hand is as high as possible.

Model checking methods are fully automatic methods that proves or disproves properties of systems. In contrast to theorem proving methods, model checking methods do not work on the syntactical description of a transition system but rather on the transition system itself explicitly encoded as a set of transitions. System specifications are verified by automatically examining all possible transitions in the encoding. Due to this exhaustive examination, model checking methods are sensitive to the so called *state explosion problem* occurring as a consequence of the asynchronous behaviours among processes in a distributed system. The number of possible states in such a system grows exponentially with the number of processes. Therefore, model checking methods are applicable only to systems with a finite, or at least finitely representable, number of states. Automatic model checking tools are usually denoted as *model checkers*.

Abstraction Based Verification. To benefit from both the insight and generality of theorem proving methods and the automation of model checking methods, there has recently been an increasing interest into frameworks aiming for an integration of the two. In this thesis we propose such frameworks for untimed and timed distributed systems based on one of the mainly investigated integration strategies – the use of *abstraction methods*.

The goal of any abstraction method is to replace the problem of verifying a large, possible infinite-state, concrete system to the problem of verifying a smaller, hopefully finite-state, abstract system. The abstract system must be *safe* with respect to the concrete system in the sense that successful verification of the abstract system carries over to the concrete system. Obviously, the goal is for the abstract system to be smaller in size than the concrete one, and hopefully small enough to be directly model checkable. In general, verifying that the abstract system is safe with respect to the concrete system cannot be done automatically and hence this step in an abstraction method is typically performed via theorem proving methods. The non-trivial part of any abstraction method lies in the problem of finding the right abstraction of a given concrete problem. Based on the type of concrete problem at hand different *abstraction strategies* may be useful.

A simple and commonly used example of the abstraction idea is the use of the rule of signs to determine the sign of an arithmetic expression. In order to say whether $-1515 * 17$ is positive or negative, we do not have to perform the multiplication on the "concrete" level of numbers and then look at the sign of the result, but instead we can first "abstract" the individual operands to their signs and then apply $\overline{*}$, the rule of signs for multiplication: $\mathbf{neg} \mathbin{\overline{*}} \mathbf{pos} = \mathbf{neg}$. This rule of signs for the product enjoys the property that its result always correctly describes the result of any concrete multiplication on any operands that it abstracts.

1.2 Scope of Thesis

This thesis deals with the topics introduced above. In this section we will describe in more detail the particular topics and contributions of this thesis.

The overall contribution can be summarized as follows:

Abstraction based verification frameworks and practical abstraction strategies to support the integration of theorem proving and model checking

In the following we first take a look at the formal system models that we use throughout this thesis. Then we discuss different forms of abstraction frameworks and we introduce the frameworks developed in this thesis. Finally, we classify different practical strategies for obtaining safe abstractions and we briefly describe the particular strategies applied for proofs in this thesis.

1.2.1 Formal Models

In this thesis we will consider the development of abstraction based verification frameworks for embedded and distributed systems of two different kinds: *timed* and *untimed* systems. A timed system in our sense is a system whose behaviours are sensitive to the existence of real-time physical clocks. Such systems are also commonly denoted as *real-time systems*. In the following we briefly introduce the formal models that we will use to describe untimed and timed systems.

I/O Automata. For untimed systems we will use *I/O Automata* as the underlying formal model. This is a general labelled transition system model with action labelled transitions distinguished as being either input, output or internal. The model was originally proposed by Lynch and Tuttle [LT87, LT89] and several subsequent developments have taken place [LV95, LV96, Lyn96, LSVW95, GSSL93, SL95] including extensions for modeling of timed, hybrid, and probabilistic systems. The I/O automaton model and its related proof methods have been applied successfully to several non-trivial case studies [DL97, LLSA94, LMWF94, WLL88].

Properties to be proved about I/O automata models are often stated as *trace properties*. The traces of an automaton consists of the set of action sequences obtained by removing states and internal actions from its set of executions. Thus the traces represent the externally observable behaviour of an automaton. A trace property is simply a set of action sequences, and an automaton is said to satisfy a trace property if the set of traces of the automaton is included in the set of action sequences of the property. Trace properties can be used to specify both *safety* and *liveness* properties of systems. Informally, a safety property says that some particular *bad* thing never happens and a liveness property expresses that something *good* will eventually happen. Trace properties can be stated indirectly in terms of automata specifications and methods based on showing the existence of *simulations* between automata are sound with respect to trace inclusion.

A simulation is a formal relationship between the states of two automata requiring that the transitions of one system can in some sense be mimicked by the other.

The I/O automaton model is equipped with a *composition* operator used to describe the parallel composition of *asynchronously* executing systems. The composition operator embodies a *synchronization mechanisms* between composed automata based on joining input–output pairs of identically named transitions in different automata. The I/O automaton model is general enough to describe distributed systems that are based on synchronization via shared memory as well as those based on message-passing.

Systems modelled as a composed automata can often be proved correct in a modular fashion, based on correctness proofs of its components. In the I/O automaton model such *compositional* reasoning is supported by compositional definitions of properties.

Other formal automaton models have been proposed [Kur94, Har87] having many of the same features as described above for the I/O automaton model. Also models based on the *process algebras* such as CCS [Mil89], CSP [Hoa85], and ACP [BHK86] include analogous modelling and proof methodologies as those of the I/O automaton model.

Timed Automata. For the modeling of timed systems we will use a formal *timed transition system model.* A timed transition system, as used in this thesis, is a transition system with two separate types of transitions: ordinary *action transitions* describing discrete state changes and *time transitions* describing the continuous evolution of system states. A behaviour of a timed transition system can be understood as a sequence of discrete action transitions separated by time transitions describing the elapse of time in between actions. This type of behaviour is often denoted as *two-phase behaviour.*

We will use *timed automata* to syntactically describe timed transition systems. A timed automaton is a standard finite automaton extended with a set of real valued clocks used to impose constraints on when transitions may be executed. The particular timed automaton model used in this thesis is a variation of the original model introduced by Alur and Dill [AD94]. Timed automata models have been used in several verification frameworks based on theorem proving methods [AL93, LV96] as well as model checking methods [DOTY96, BLL$^+$95, AHH96], and for both methodologies several case-studies on verification of embedded real-time systems have been carried out. For case-studies using theorem proving see for example [SA93, LLSA94, HL94, Luc95] and for model checking see for example [DY95, Kri98, HWT95]. Also process algebraic languages have been proposed that are capable of modeling real-time systems [Yi91, NSY91, BB89].

The verification methodology for timed systems considered in this thesis is based on a notion of *timed simulations.* A timed simulation is a simulation relation from an implementation to a specification where both discrete action transitions and timed transitions in the implementation can be mimicked by corresponding transitions in the specification.

1.2.2 Theorem Provers and Model Checkers

Theorem provers are based on constructing proofs in a logical deduction system either automatically or user assisted. Examples of widely used theorem proving systems include HOL [GM93], Isabelle [Pau94], PVS [ORR$^+$96], and LP [GG91]. In this thesis we will make use of LP (the Larch Proof assistant) to support abstraction based proofs in our untimed I/O automata abstraction framework. LP is a theorem prover for multisorted first-order logic designed to assist users in employing standard mathematical reasoning. We will formalize parts of our abstraction theory in LSL [GH93] (the Larch Shared Language) which is supported by a tool that automatically provides input for LP.

Model checkers are based on constructing proofs of properties specified as formulae in a logic interpreted over the semantic model (transition system) of systems. Typically the logic is some kind of *temporal logic* like Linear Time Logic (LTL) [MP92], Computation Tree Logic [CES86], μ-calculus [Koz82], or Hennesy Milner Logic (HML) [HM85]. Proving correctness of a system means to checking if a given formula is satisfied by the system model. For finite-state systems this can in principle be done automatically by an exhaustive traversal of the reachable system state space. As we mentioned earlier though systems consisting of large numbers of parallel components may run into the state explosion problem making an exhaustive traversal impossible in practice. In the last decade, several approaches have been proposed to reduce the effects of the state explosion problem [BCM$^+$90, GW94, HP94] resulting in a number of efficient model checking tools for untimed systems [Hol91, HK87, McM93, CPS89] as well as real-time and hybrid systems [DOTY96, BLL$^+$95, AHH96]. In this thesis we will make use the SPIN [Hol91] model checker in our untimed framework by providing a translation scheme for implementing I/O automata in the input language of SPIN. In the timed framework we will make use of the UPPAAL [BLL$^+$95] model checker. In our timed framework we use a formal timed automata model to describe systems which is the input language of UPPAAL and thus no translation is required.

1.2.3 Abstraction Frameworks

The goal of any abstraction method is to reduce the problem of verifying a large, possible infinite-state, concrete system to the problem of verifying a smaller, hopefully finite-state, abstract system. Given a concrete system together with a concrete requirement specification, one must construct an abstract system and an abstract specification such that the abstract system is property preserving. Meaning that, if the abstract system satisfies the abstract specification then this implies that the concrete system satisfies the concrete specification. Abstraction methods can be classified according to two types based on the amount of information they intend to preserve in abstract models. The two types are *weakly preserving* methods and *strongly preserving* methods.

Weakly Preserving Methods. A weakly preserving method only preserves the satisfaction of properties in one direction, namely *if* the abstract model satisfies the abstract requirement specification *then* the concrete model satisfies the corresponding concrete specification. To illustrate this, consider again the example abstraction idea of using the rule of signs to determine the sign of some arithmetic expression. If x is a variable over the natural numbers, we want to examine the sign of the expression x^2+x+1 for all x. It is easy to see, that the sign of this expression is always positive, but this can actually not be proven using the abstraction idea from before. The reason for this is the introduction of the + (addition) operator. The rule of signs for addition is as follows: **pos** $\overline{+}$ **pos** = **pos**, **neg** $\overline{+}$ **neg** = **neg**, **pos** $\overline{+}$ **neg** = **undef**, **neg** $\overline{+}$ **pos** = **undef**, where **undef** is an abstract value describing that nothing is known about the result of the abstract addition, it is either **pos** or **neg** but which is unknown. So when deciding, in the abstract calculus, the sign of the considered expression, we would get (e.g. for $x = -3$): (**neg** $\overline{*}$ **neg**) $\overline{+}$ **neg** $\overline{+}$ **pos**, which can be reduced to **neg** $\overline{+}$ **pos** = **undef**.

Weakly preserving methods have been studied in for example [Dam96, DF95, CGL92, Kur89, LGS+95, MN95]. The methods can be divided into types based on the kind of mathematical relation required to exist between states of the concrete model and those of an abstraction. The typical kinds of relations are: *simulation relations*, *homomorphic functions* and *Galois connections.* These relations all require, that the abstract model can somehow "mimic" the behaviour of the concrete model, but not necessarily the other way round, meaning that the abstraction may have some behaviour which is not mimicking any concrete behaviour. Thus weakly preserving abstractions only preserve truth of properties. However, weakly preserving abstractions can be as "abstract" or "small" as one wants still being weakly preserving, the only problem being that the more "abstract" an abstraction becomes the fewer properties does it enjoy.

Some weakly preserving methods [Dam96, LGS+95] have been proposed, where the abstraction relation is given in the form of a Galois connection. These methods can all be related to a classical framework of Abstract Interpretation initially developed in [CC77, CC79] and presented in overview in [CC92a, CC92b]. Abstraction methods based on Galois connections can be seen as attempts to address the question of which of the many possible correctly mimicking abstract models should be the chosen one. Adding some extra requirement to the structure of the abstract domain, the Galois connection framework allows for an ordering of candidate abstract models with respect to the number of properties they enjoy. The more properties an abstract model enjoys the more *precise* it is. The structure imposed on abstract states actually guarantees the existence of a *most precise* abstract model which is the natural choice among the many possibly candidates.

Strongly Preserving Methods. A strongly preserving abstraction preserves properties *both ways* in the sense that the abstract model satisfies the abstract property *if and only if* the concrete model satisfies the corresponding concrete property. In the example from before, the abstraction to the rule of signs is strongly preserving with respect to arithmetic expressions only including multiplication but only weakly preserving when allowing addition as well. As a result, strongly preserving abstractions preserve *both* truth and falsehood of properties, so both positive and negative results from verification of the abstraction carry over to identical results for the concrete model. However, strongly preserving abstractions are more restricted than weak ones regarding the amount of behaviour that can be abstracted away, which has the result that a sufficiently small (for model checking) abstraction cannot always be constructed.

Strongly preserving abstraction methods have been studied extensively in the framework of real-time systems model checking [BLL$^+$95, DOTY96, AHH96, HHK95]. When modeling a real-time system using a dense time domain its direct semantics (e.g. timed transition system) will have infinitely many states and thus it is not directly amenable to model checking. However, when described as a timed automaton it is possible to partition the set of concrete states based on a certain equivalence relation on the time components of states. The equivalence classes of this partitioning forms a set of abstract states. The equivalence relation is induced by a set of linear inequations on automaton clocks obtained from the clock guards in transitions. It is important to note that only the time component is abstracted in real-time model checkers based on the above approach - no abstraction of the discrete component takes place.

Our Contributions. In this thesis we contribute with weakly preserving abstraction frameworks for untimed as well as timed systems. Our untimed framework is for the general untimed I/O automaton model. Here we provide general conditions for preservation of trace and *path* properties from one automaton to another. A path property is analogous to a trace property except that it describes a property of state sequences rather than action sequences. Our preservation conditions are provided for safety as well as liveness properties. The preservation conditions are based on certain notions of *parameterized* simulations that we introduce as well. We formalize parts of our trace based abstraction theory in the Larch tool set to support in discharging preservation conditions, and we provide a rudimentary scheme for translating finite state I/O automata into the input language for the SPIN model checker. This includes a way of stating trace and path properties via LTL formulas.

Our timed framework is for a timed automaton model which is essentially the one used as input language for the UPPAAL real-time model checker. Thus, we can make use of the time abstraction method implemented in the tool. We provide general conditions stating when an abstract timed simulation problem is safe with respect to a concrete problem. The conditions are based on certain parameterized timed simulations. Checking for the existence of a timed simulation relation between two timed automata is not directly implemented in the UPPAAL tool which is based on performing *reachability analysis*. However, we provide a method for translating the problem of checking for timed simulations into a reachability problem amenable to verification in UPPAAL.

1.2.4 Applied Abstraction Strategies

Practical abstraction strategies are often classified as either *data abstractions* or *control abstractions* depending on the source of complexity in the concrete system of interest. The first class is where the unboundedness (or large size) of the system results from data variables which range over unbounded domains. The second class is where the complexity results from the structure of the system. Systems of the latter kind are typical *parameterized systems* consisting of a parallel composition of subsystems, whose number is a varying parameter.

Most data abstraction strategies are based on the method of *abstract interpretation* originally introduced in [CC77, CC79]. Most of the weakly preserving frameworks cited in the previous section are of this kind. The concrete data domain is abstracted into an abstract domain and all operations of the concrete system are replaced by abstract versions over the abstracted domain. The abstract domain is typically based on a partitioning of the concrete data domain induced by the set of conditions (guards) in the concrete system description and the concrete property. This is in some sense analogous to the time abstraction strategy for real-time systems.

Several control abstraction strategies have been proposed for the verification of parameterized systems. Many of these are based on the use of *network invariants* [KM89, WL89a, HLR92]. A network invariant is an abstract system intended to simulate the composition of concrete subsystems for any number of elements in the composition. Using induction and properties of compositionality the simulation proof task can be reduced to a problem of showing simulation relations between a few unparameterized systems.

Our Contributions. In this thesis we present practically applied abstraction strategies for three nontrivial distributed algorithms. First we consider a proof of Burn's mutual exclusion algorithm parameterized in the number of processes. The control abstraction applied for the proof utilizes a *skolemization* strategy to extract a simple two-process abstraction representing any pair of concrete processes *including* the effects of the environment (other processes). Our proof is within the untimed I/O automaton abstraction framework and it makes use of the LP theorem prover as well as the SPIN model checker.

Second, we provide a proof of one of the most complicated algorithms in the distributed systems literature, the Bounded Concurrent Timestamp System (BCTSS) algorithm. The algorithm is parameterized in the number of processes and in addition it has data variables over unbounded domains. Thus our abstraction makes use of both control and data abstraction strategies. Existing proofs for the algorithm are all long and hard to understand. Our abstraction proof reduces the proof task by automating a substantial proof effort via abstraction. Our proof exploits a combination of induction and abstraction and it is the most advanced proof in this thesis.

Finally, we provide a proof Fischer's mutual exclusion algorithm parameterized in the number of processes. Fischer's algorithm is a real-time algorithm and thus our proof is within the timed abstraction framework. Our proof uses a combination of parameterized network invariants and compositionality.

1.2.5 Thesis Outline

This thesis consists of two parts. Part I (Abstraction Frameworks) presents the untimed and timed abstraction frameworks in chapters 2 and 3, respectively. Part II (Applied Abstraction Strategies) presents our three practical applications of abstraction strategies. Chapter 4 presents the proof of Burn's algorithm, Chapter 5 presents the proof of the BCTSS algorithm, and finally Chapter 6 presents the proof of Fischer's algorithm.

Part I

Abstraction Frameworks

Chapter 2

Untimed Abstraction Framework

This chapter presents our untimed abstraction framework. We consider untimed systems specified in the general I/O automaton model of Lynch and Tuttle [LT87, LT89] and we assume that properties to be verified over automata are stated as either trace or path properties. A trace property is a property about the actions of an automaton and a path property is a property about states. Our abstraction theory provides safe conditions for replacing one verification problem (the concrete one) (A, P), consisting of an I/O automaton A and a property P, by another problem (the abstract one) (A', P'). That is conditions allowing us to conclude that if A' satisfies P' then also A satisfies P. We provide conditions for preservation of trace properties as well as path properties and for both safety and liveness properties. Our conditions provide basis for a weakly preserving abstraction method integrating theorem proving and model checking techniques in two steps: First, given a concrete problem (A, P), too large to be handled immediately by model checking, an abstract problem (A', P') is constructed and shown to satisfy conditions for *property preservation.* This step in general requires theorem proving methods. Second, the abstract problem is automatically analyzed using model checking, provided it is finite state and small enough.

Our preservation conditions are based on variants of the *forward simulation* preorder which constitutes a corner stone proof method in the general I/O automaton model. We propose two variations of the standard forward simulation useful for providing preservation conditions for trace properties and path properties, respectively.

We formalize parts of our abstraction theory using the Larch tool set [GG91, GH93] to provide support for discharging the proof obligations for property preservation. We use the Larch Shared Language (LSL) [GH93] to formalize the notion of I/O automata and the notions of simulation that we use as conditions for property preservation. LSL is supported by a tool that automatically provides input for the Larch Prover (LP) [GG91]. LP is a theorem prover for multi-sorted first-order logic. Our formalization extends a framework for reasoning about I/O automata in the Larch tool set introduced in [SAGG$^+$93]. We also provide a rudimentary scheme for translating finite state I/O automata models into the input language PROMELA used by the SPIN model checker [Hol91]. SPIN is capable of model checking properties in Linear Time Logic (LTL) and our scheme includes a strategy for specifying trace and path properties using LTL formulas.

Chapter Organization. We begin in Section 2.1 by a few mathematical preliminaries on relations, functions and sequences. Then in Section 2.2 we introduce the underlying formal model of I/O automata as well as the notions of trace and path properties. In Section 2.3 we present our general conditions for property preservation between verification problems based on our variations of the forward simulation preorder. In Section 2.4 we formalize part of our abstraction theory using the Larch tool set and finally in Section 2.5 we examine the translation of I/O Automata and trace/path properties into a representation suitable for the model checker SPIN.

2.1 Preliminaries

The following mathematical preliminaries defines notions of *relations*, *functions*, and *sequences*. The notions introduced here will be used for both the untimed and the timed abstraction frameworks of this thesis.

2.1.1 Relations, Functions, and Sequences

A *relation* over sets X and Y is defined to be any subset of the cartesian product $X \times Y$. If R is a relation over X and Y, then we define the *domain* of R to be $dom(R) = \{x \in X \mid (x, y) \in R \text{ for some } y \in Y\}$, and the range of R to be $ran(R) = \{y \in Y \mid (x, y) \in R \text{ for some } x \in X\}$. If $dom(R) = X$ we say that R is *total* (on X). For $x \in X$, we define $R[x] = \{y \in Y \mid (x, y) \in R\}$. A *function* f from X to Y is a relation with $dom(f) = X$ satisfying that for any x there exists exactly one y such that $(x, y) \in f$. We write $f(x) = y$ to denote $(x, y) \in f$. If f and g are functions with disjoint domains, then $f \cup g$ denotes the function from $dom(f) \cup dom(g)$ to $ran(f) \cup ran(g)$ such that $(f \cup g)(x) = f(x)$ if $x \in dom(f)$ and $(f \cup g)(x) = g(x)$ if $x \in dom(g)$.

Let S be any set. The set of finite and infinite sequences of elements from S is denoted $seq(S)$. The symbol λ denotes the empty sequence and the sequence containing one element $s \in S$ is denoted by s. Concatenation of a finite sequence with a finite or infinite sequence is denoted by juxtaposition. Let Σ and Σ' denote sets of sequences such that all sequences in Σ are finite. The concatenation of sets Σ and Σ' is the set $\Sigma\Sigma'$ of sequences $\sigma\sigma'$ such that $\sigma \in \Sigma$ and $\sigma' \in \Sigma'$. A sequence σ is a *prefix* of a sequence ρ, denoted by $\sigma \leq \rho$, if either $\sigma = \rho$, or σ is finite and $\rho = \sigma\sigma'$ for some sequence σ'. A set Σ of sequences is *prefix closed* if, whenever some sequence is in Σ, all its prefixes are as well. A set Σ of sequences is *limit closed* if, an infinite sequence is in Σ whenever all its finite prefixes are.

A *block* over S is a sequence of identical elements from S. Any sequence σ of elements from S can be viewed as a sequence of blocks over S. For any sequence σ we assume a partial function $\sigma : \mathbf{N} \rightarrow S \times \mathbf{N}$ such that $\sigma(i) = \langle s, n \rangle$ iff the i'th block of σ is the sequence consisting of n s-states. We assume that $\sigma_s(i)$ and $\sigma_n(i)$ returns the first and second component of $\sigma(i)$, respectively. We further assume that $\sigma_s(i) \neq \sigma_s(i+1)$ for every i such that σ is defined for i and $i+1$. Thus, consecutive blocks are of different states. We let $lb(\sigma) = i$ if the i'th block is the last block of σ. If no last block exists we let $lb(\sigma) = \omega$. Let σ and σ' be sequences over S. We say that σ is a *block-prefix* of σ', written $\sigma \trianglelefteq \sigma'$, if $lb(\sigma) = lb(\sigma')$ and for all $1 \leq i \leq lb(\sigma)$, $\sigma_s(i) = \sigma'_s(i)$ and $\sigma_n(i) \leq \sigma'_n(i)$.

If σ is a nonempty sequence then *first*(σ) denotes the first element of σ, and *tail*(σ) denotes the sequence obtained from σ by removing *first*(σ). Also, if σ is finite, *last*(σ) denotes the last element of σ.

If $\sigma \in seq(S)$, and $S' \subseteq S$, then $\sigma|S'$ denotes the restriction of σ to elements in S', i.e. the subsequence of σ consisting of the elements of S'. If $S' = \emptyset$ then $\sigma|S' = \lambda$. If $\Sigma \subseteq seq(S)$, then $\Sigma|S'$ is the set $\{\sigma|S' \mid \sigma \in \Sigma\}$.

Assume $R \subseteq S \times S'$ is a total relation between sets S and S'. If $\sigma = s_0 s_1 s_2 \ldots$ is a nonempty sequence in $seq(S)$ then $R(\sigma)$ is the set of sequences $s'_0 s'_1 s'_2 \ldots$ over $ran(R)$ such that for all i, $s'_i \in R[s_i]$. If $\sigma = \lambda$ then $R(\sigma) = \{\lambda\}$. If $\Sigma \subseteq seq(S)$, then $R(\Sigma) = \bigcup_{\sigma \in \Sigma} R(\sigma)$

2.2 I/O Automata

In this section we present the basic I/O automaton model used to describe untimed distributed systems.

An I/O automaton is an action labelled transition system where actions are classified as either *input*, *output*, or *internal.* The inputs and outputs are used by an automaton to communicate with its environment, while the internal actions are visible only to the automaton itself. An I/O automaton cannot *guard* its input actions. This means, that input actions can arrive from the environment at any time. We say that I/O automata are *input-enabled.* Only the output and internal actions of an automaton can be controlled by the automaton itself.

Definition 2.1 (I/O Automaton) *An I/O automaton, or simply an automaton, A is a tuple consisting of components sig(A), states(A), start(A), trans(A), and tasks(A) where,*

- *$sig(A)$ is a signature, which is a tuple consisting of components $in(A)$, $out(A)$, and $int(A)$, being disjoint sets of input, output and internal actions, respectively. The set $ext(A)$ of external actions of A is $in(A) \cup out(A)$ and the set of locally controlled actions $local(A)$ is the set $out(A) \cup int(A)$. The set $acts(A)$ of actions of A is the set $ext(A) \cup int(A)$.*
- *$states(A)$ is a set of states.*
- *$start(A) \subseteq states(A)$ is a nonempty set of start states.*
- *$trans(A) \subseteq states(A) \times acts(A) \times states(A)$ is a state transition relation.*
- *$tasks(A)$ is a task partition, which is an equivalence relation on the set $local(A)$ having at most countably many equivalence classes.*

We let $s, s', u, u', \ldots$ range over states, and $a, a', b, b' \ldots$ over actions. We call an element (s, a, s') of $trans(A)$ a *transition*, or *step*, of A. We write $s \xrightarrow{a}_A s'$, or just $s \xrightarrow{a} s'$ if A is clear from the context, as a shorthand for $(s, a, s') \in trans(A)$. We write $s \longrightarrow_A s'$, or just $s \longrightarrow s'$ if A is clear from the context, to denote that $s \xrightarrow{a}_A s'$ for some action a.

An action a of an automaton A is said to be *enabled* in a state s if there exists a state s' such that the transition (s, a, s') is an element of $trans(A)$. Every input action is required to be enabled in any state.

Executions, Traces, and Paths

An *execution fragment* $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ of an automaton A is a finite or infinite sequence of alternating states and actions beginning with a state, and if it is finite also ending with a state, such that for all i, (s_i, a_{i+1}, s_{i+1}) is an element of $trans(A)$. An *execution* of A is an execution fragment α where $first(\alpha) \in start(A)$. A state s of A is *reachable* if $s = last(\alpha)$ for some finite execution α of A.

Let $\alpha = s_0 a_1 s_1 a_2 s_2 \ldots$ be an execution fragment. The *length* of α, written $|\alpha|$, is defined as follows. If α is finite, $|\alpha|$ is the number of actions occurring in α. If α is infinite, $|\alpha| = \omega$. We define the ith *suffix* of α, for $0 \leq i \leq |\alpha|$, as $_i|\alpha = s_i a_{i+1} s_{i+1} \ldots$ if $i < |\alpha|$; $s_{|\alpha|}$ if α is finite and $i = |\alpha|$.

The *trace* of an execution fragment α of an automaton A, written as $trace_A(\alpha)$, or just $trace(\alpha)$ when A is clear from context, is the subsequence consisting of all the external actions occurring in α. Let β be a sequence of actions from $acts(A)$. Then, $trace_A(\beta)$, or just $trace(\beta)$ when A is clear from context, denotes the subsequence of consisting of all the external actions occurring in β. We say that β is a trace of an automaton A if there is an execution α of A with $\beta = trace(\alpha)$. We denote the set of traces of A by $traces(A)$.

The *path* of an execution α of an automaton A, written $path_A(\alpha)$, or just $path(\alpha)$ when A is clear from context, is the subsequence consisting of all the states in α. We say that γ is a path of A if there is an execution α of A with $\gamma = path(\alpha)$. We denote the set of paths of A by $paths(A)$.

Fair Executions, Fair Traces, and Fair Paths

The task partition $tasks(A)$ of an automaton A, can be thought of as an abstract description of "tasks" or "threads of control" within A. The partition is used to specify *fairness* conditions on A. Such conditions state that during execution A must give fair turns to each of its tasks. The fairness conditions considered in this section are sometimes denoted as *weak fairness* conditions. We say that a task C is *enabled* in a state s if some action in C is enabled in s.

An execution fragment α of an automaton A is said to be *fair* if the following conditions hold for each task C of $tasks(A)$:

- If α is finite, then C is not enabled in the final state of α.
- If α is infinite, then α contains either infinitely many occurrences of actions from C or infinitely many occurrences of states in which C is not enabled.

We denote the set of fair executions of A by $fairexecs(A)$. We say that β is a *fair trace* of A if β is the trace of a fair execution of A, and we denote the set of fair traces of A by $fairtraces(A)$. We say that γ is a *fair path* of A if γ is the path of a fair execution of A, and we denote the set of fair paths of A by $fairpaths(A)$.

2.2.1 Composition

We can compose individual automata to represent complex systems of interacting components. We impose certain restrictions on the automata that may be composed.

Formally, we define a countable collection $\{S_i\}_{i \in I}$ of signatures to be *compatible* if for all $i, j \in I$, $i \neq j$, all of the following hold: $int(S_i) \cap acts(S_j) = \emptyset$, $out(S_i) \cap out(S_j) = \emptyset$, and no action is contained in infinitely many sets $acts(S_i)$.

Definition 2.2 (Composition of Signatures) *Define the composition* $S = \Pi_{i \in I} S_i$ *of a countable compatible collection of signatures* $\{S_i\}_{i \in I}$ *as the signature with*

- $out(S) = \cup_{i \in I} out(S_i)$
- $int(S) = \cup_{i \in I} int(S_i)$

- $in(S) = \cup_{i \in I} in(S_i) - \cup_{i \in I} out(S_i)$

We say that a collection of automata is *compatible* if their signatures are compatible.

Definition 2.3 (Composition of Automata) *Define the composition* $A = \Pi_{i \in I} A_i$ *of a countable, compatible collection of I/O automata* $\{A_i\}_{i \in I}$ *as the automaton with*

- $sig(A) = \Pi_{i \in I} sig(A_i)$
- $states(A) = \Pi_{i \in I} states(A_i)$
- $start(A) = \Pi_{i \in I} start(A_i)$
- $trans(A)$ *is the set of triples* (s, π, s') *such that, for all* $i \in I$*, if* $\pi \in acts(A_i)$*, then* $(s_i, \pi, s'_i) \in trans(A_i)$*; otherwise* $s_i = s'_i$
- $tasks(A) = \cup_{i \in I} tasks(A_i)$

The Π in the definition of $states(A)$ and $start(A)$ refers to ordinary Cartesian product. Also, s_i in the definition of $trans(A)$ denotes the ith component of state vector s.

Notice, that the task partition of the compositions locally controlled actions is formed by taking the union of the components task partitions; that is, each equivalence class of each component automaton becomes an equivalence class of the composition. This means that the task structure of individual components is preserved when the components are composed. Notice also, that since the automata A_i are input-enabled, so is their composition. It follows that $\Pi_{i \in I} A_i$ is indeed an I/O automaton.

We end this section with a basic result that relates the executions, traces and paths of a composition to those of the component automata. Let $A = \Pi_{i \in I} A_i$ be a composition of automata. Given an execution $\alpha = s_0 a_1 s_2 a_2 s_2 \ldots$ of A, we let $\alpha | A_i$ be the sequence obtained by deleting each pair a_r, s_r for which a_r is not an action of A_i and replacing each remaining s_r by $(s_r)_i$, that is, automaton A_i's piece of the state s_r. Given a trace β of A, we let $\beta | A_i$ be the subsequence of β consisting of all the actions of A_i in β. Also, given a path $\gamma = s_0 s_1 s_2 \ldots$ of A, we let $\gamma | A_i$ be the sequence obtained by replacing each s_r by $(s_r)_i$.

Theorem 2.1 *Let* $\{A\}_{i \in I}$ *be a compatible collection of automata and let* $A = \Pi_{i \in I} A_i$.

1. *If* $\alpha \in execs(A)$*, then for every* $i \in I$, $\alpha | A_i \in execs(A_i)$.
2. *If* $\beta \in traces(A)$*, then for every* $i \in I$, $\beta | A_i \in traces(A_i)$.
3. *If* $\gamma \in paths(A)$*, then for every* $i \in I$*, there exists* $\gamma' \trianglelefteq \gamma | A_i$ *such that* $\gamma' \in paths(A_i)$.

4. *If $\alpha \in \mathit{fairexecs}(A)$, then for every $i \in I$, $\alpha|A_i \in \mathit{fairexecs}(A_i)$.*

5. *If $\beta \in \mathit{fairtraces}(A)$, then for every $i \in I$, $\beta|A_i \in \mathit{fairtraces}(A_i)$.*

6. *If $\gamma \in \mathit{fairpaths}(A)$, then for every $i \in I$, there exists $\gamma' \trianglelefteq \gamma|A_i$ such that $\gamma' \in \mathit{fairpaths}(A_i)$.*

Proof. The statements 1-2 and 4-5 are Theorem 8.1 and Theorem 8.4, respectively, in [Lyn96]. Consider 3. Suppose that $\gamma \in \mathit{paths}(A)$. Then there exists $\alpha \in \mathit{execs}(A)$ such that $\gamma = \mathit{path}(\alpha)$. Thus $\gamma|A_i = \mathit{path}(\alpha)|A_i$. From the definition of block prefix $\trianglelefteq$ (see Section 2.1) we have that, $\mathit{path}(\alpha|A_i) \trianglelefteq \mathit{path}(\alpha)|A_i$ and from 1, $\alpha|A_i \in \mathit{execs}(A_i)$. Let $\gamma' = \mathit{path}(\alpha|A_i)$. Then, $\gamma' \in \mathit{paths}(A_i)$ and $\gamma' \trianglelefteq \gamma|A_i$. Consider 6. Suppose that $\gamma \in \mathit{fairpaths}(A)$. Then there exists $\alpha \in \mathit{fairexecs}(A)$ such that $\gamma = \mathit{path}(\alpha)$. Thus $\gamma|A_i = \mathit{path}(\alpha)|A_i$. From definition, $\mathit{path}(\alpha|A_i) \trianglelefteq \mathit{path}(\alpha)|A_i$ and from 4, $\alpha|A_i \in \mathit{fairexecs}(A_i)$. Let $\gamma' = \mathit{path}(\alpha|A_i)$. Then, $\gamma' \in \mathit{fairpaths}(A_i)$ and $\gamma' \trianglelefteq \gamma|A_i$.
■

2.2.2 Precondition-Effect Language

In this thesis we will describe automata using a *precondition-effect style*, which is the standard description style for I/O automata [Lyn96].

The precondition-effect style basically provides a compact description of the transition relation of an automaton. The style groups together all the transitions that involve each particular type of action into a single piece of code. The code specifies the conditions under which the action is permitted to occur, as a predicate (precondition) on the pre-state. Then it describes the changes (effects) that occur as a result of the action. These changes are described either as an assertion relating pre- and post-state or as a sequence of operations that is applied to the pre-state in order to yield the post-state.

Example 2.1 (Channel I/O Automaton) As an example of an I/O automaton described in the precondition-effect style, consider a communication channel automaton C as shown below. We assume that M is a fixed message alphabet. First the signature, $\mathit{sig}(C)$, of automaton C is given. In this example the signature only contains input and output actions, i.e. the set of internal actions is empty. Next the states, $\mathit{states}(C)$, and the start states, $\mathit{start}(C)$, are given as a list of state variables and their initial values. The transitions of C are described in the precondition-effect style. The *send*(m) action is allowed to occur at any time and has the effect of adding the message m to the end of *queue*. The *receive*(m) action can only occur if m is at the head of *queue*, and the effect of the action simply consists of m being removed from *queue*. Finally, the task partition, $\mathit{tasks}(C)$, groups all the *receive* actions into a single task.

Automaton: C

Signature:
Input:
  $send(m), m \in M$
Output:
  $receive(m), m \in M$

States:
$queue$, a FIFO queue of elements from M, initially empty

Transitions:

input: $send(m)$
  Eff: $enqueue(m, queue)$

output: $receive(m)$
  Pre: $head(queue) = m$
  Eff: $dequeue(queue)$

Tasks:
$\{receive(m) : m \in M\}$

2.2.3 Properties of I/O Automata

In this section we define the types of properties that we will use to reason about automata. We consider properties stated over either the external actions or the states of an automaton, and we denote such properties as *trace properties* and *path properties*, respectively. A trace property for an automaton A is basically a set of sequences of actions from a subset of $ext(A)$, and a path property for A is basically a set of sequences of states from $states(A)$.

Definition 2.4 (Trace Property) *A trace property P is a pair of components $sig(P)$ and $traces(P)$, where*

- *$sig(P)$ is a signature, which is a pair $(in(P), out(P))$ consisting of disjoint sets of input and output actions, respectively. Let $acts(P)$ be the set $in(P) \cup out(P)$.*
- *$traces(P)$ is a set of finite or infinite sequences of actions in $acts(P)$.*

Definition 2.5 (Path Property) *A path property Q is a pair of components $states(Q)$ and $paths(Q)$, where*

- *$states(Q)$ is a set of states.*
- *$paths(Q)$ is a set of finite of infinite sequences of states in $states(Q)$.*

Two important special types of trace and path properties – *safety properties* and *liveness properties* – are defined.

Definition 2.6 (Safety and Liveness Properties) *A trace (path) property P is said to be a trace (path) safety property provided that all of the following holds:*

- *$traces(P)$ ($paths(P)$) is nonempty.*
- *$traces(P)$ ($paths(P)$) is prefix-closed.*
- *$traces(P)$ ($paths(P)$) is limit-closed.*

A trace (path) property P is said to be a trace (path) liveness property provided that:

- *Every finite sequence over $acts(P)$ ($states(P)$) has some extension in $traces(P)$ ($paths(P)$).*

A safety property can informally be interpreted as saying that no "bad thing" ever happens. A liveness property can be interpreted as saying that some "good thing" eventually happens. Thus, the intuition behind the conditions of the above definition can be understood as follows. For the safety conditions, nonemptiness is a reasonable condition since no "bad thing" can ever happen in the empty sequence. Prefix-closure is reasonable since, if nothing bad happens in a sequence (trace or path), then nothing bad happens in any prefix of that sequence. Finally, limit-closure is reasonable since, if something bad happens in a sequence, then it happens at some particular "point" in the sequence, i.e. in some finite prefix of the sequence. The intuition behind the liveness condition is simply that regardless of what has occurred in a sequence up to some point, it is still possible for the "good thing" to occur at some later point in time.

Definition 2.7 (Satisfying Trace Properties) *An automaton A satisfies a trace property P iff the following conditions hold.*

- *$in(P) \subseteq in(A)$ and $out(P) \subseteq out(A)$.*
- *If P is a safety property, then $traces(A)|acts(P) \subseteq traces(P)$.*
- *If P is a liveness property, then $fairtraces(A)|acts(P) \subseteq traces(P)$.*

Intuitively, automaton A satisfies trace property P, if P is stated over a subset of the external actions of A, and the set of traces (fairtraces) of A projected on to the actions of this subset, is a subset of the traces of P. This notion of satisfaction of trace properties is a slight generalization of the standard notion in [Lyn96]. In the standard notion, A satisfies P if P is stated over the full set of external actions of A, and the set of traces (fairtraces) of A is a subset of the traces of P. Note, this is a special case of our definition. The motivation for our definition is related to the ease at which we can show that one verification problem is property-preserving with respect to another problem.

Let A be an automaton and let P be a trace (path) property. We will denote the pair (A, P) a *trace (path) verification problem.* Given two verification problems (A, P) and (B, P'), we say that (B, P') is property-preserving with respect to (A, P) iff B satisfies P' implies A satisfies P. Proving this property-preservation partly relies on showing that the behavior of A with respect to actions of P can be *simulated* by B. Now, most likely P does not impose restrictions on all the external actions of A. However, the standard interpretation of A satisfying P does not allow for explicit mentioning of the interesting subset of external actions. Thus, automaton B need to have "dummy" actions to "match" (in the simulation) the external actions of A that are outside the interesting subset. Otherwise, we cannot show that B simulates A with respect to the actions of P. Our interpretation of A satisfying P allows for an explicit mentioning of the interesting subset of external actions of A. Thus, using our interpretation, automaton B need not to have any dummy actions to match uninteresting external actions of A. Such uninteresting actions can simply be matched by B doing nothing.

In Section 2.3 we formally define the notion of "trace" simulation intuitively introduced in the above. We also formalize the precise conditions for an abstract trace problem to be property-preserving with respect to a concrete problem.

Definition 2.8 (Satisfying Path Properties) *An automaton A satisfies a path property Q iff the following conditions hold.*

- *$states(Q) = states(A)$.*
- *If Q is a safety property, then $paths(A) \subseteq paths(Q)$.*
- *If P is a liveness property, then $fairpaths(A) \subseteq paths(Q)$.*

Note, that the above definition is not quite "symmetric" to the one for trace properties. We can make the definition symmetric by first imposing a structure on to the states of an automaton. We can e.g. define a state as a tuple of elements from a list of component sets. This will allow us to state a path property over a subset of the component sets. An automaton A then satisfies a path property Q if both of the following conditions are satisfied. First, $states(Q)$ equals the set $states(A)$ restricted to a certain subset of component sets in $states(A)$. Second, any path of A in which each state is restricted to the components of Q is included in the paths of Q.

We use the less general (but simpler) definition for satisfaction of path properties, since the more general definition will not, as was the case in the trace setting, simplify the proof obligations for property-preservation in the path setting. We explain this intuitively as follows.

Let A be an automaton and let Q be a path property. We denote the pair (A, Q) a *path verification problem.* The notion of property-preservation between path verification problems is analogous to the one for trace verification problems. The condition for property-preservation however, is slightly different in the path setting. Suppose we want to show that a path problem

(B, Q') is property-preserving with respect to a path problem (A, Q). We need to show that any transition of A can be "matched" by a transition of B, where "matched" now refers to states. I.e. the start states of each transition must "match" and so must the end states. (whether the actions match is irrelevant). Even transitions of A that do not change the state with respect to components of Q need to be matched by B. This is because Q may be sensitive to the number of certain states successively occurring in a path. In Section 2.3 we formally define the notion of "path" simulation intuitively introduced in the above.

Compositional Reasoning

We can sometimes state a property (trace or path) of a composition of automata as a *composition of properties*, one for each component automaton. We show that under the right interpretation of property composition, we can infer that a composition of automata satisfies a composition of properties from the fact that each component automata satisfies a corresponding component property.

We define a composition operation for trace properties as follows. We say that a countable collection $\{P_i\}_{i \in I}$ of trace properties is *compatible* if their signatures are compatible.

Definition 2.9 (Composition of Trace Properties) *Define the composition $P = \Pi_{i \in I} P_i$ of a countable, compatible collection of trace properties $\{P_i\}_{i \in I}$ as the trace property such that:*

- $sig(P) = \Pi_{i \in I} sig(P_i)$
- *$traces(P)$ is the set of sequences β of actions of P such that for all $i \in I$, $\beta | acts(P_i) \in traces(P_i)$*

Analogously, we define a composition operation for path properties as follows.

Definition 2.10 (Composition of Path Properties) *Define the composition $P = \Pi_{i \in I} P_i$ of a countable collection of path properties $\{P_i\}_{i \in I}$ as the path property such that:*

- $states(P) = \Pi_{i \in I} states(P_i)$
- *$paths(P)$ is the set of sequences γ of states of P such that for all $i \in I$, there exists $\gamma' \trianglelefteq \gamma | states(P_i)$ such that $\gamma' \in paths(P_i)$*

Theorem 2.2 (Compositionality of Trace Properties) *Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \Pi_{i \in I} A_i$. Also, let $\{P_i\}_{i \in I}$ be a compatible collection of trace properties and let $P = \Pi_{i \in I} P_i$. Furthermore, assume that $in(P) \cap out(A) = \emptyset$.*

1. *If $in(P_i) \subseteq in(A_i)$, $out(P_i) \subseteq out(A_i)$, and $traces(A_i)|acts(P_i) \subseteq traces(P_i)$ for every i, then $in(P) \subseteq in(A)$, $out(P) \subseteq out(A)$, and $traces(A)|acts(P) \subseteq traces(P)$.*

2. *If $in(P_i) \subseteq in(A_i)$, $out(P_i) \subseteq out(A_i)$, and $fairtraces(A_i)|acts(P_i) \subseteq traces(P_i)$ for every i, then $in(P) \subseteq in(A)$, $out(P) \subseteq out(A)$, and $fairtraces(A)|acts(P) \subseteq traces(P)$.*

Proof. Consider 1. First we show that $in(P) \subseteq in(A)$ and $out(P) \subseteq out(A)$. From the hypothesis, $in(P_i) \subseteq in(A_i)$ and $out(P_i) \subseteq out(A_i)$ for every i. Since $out(P) = \cup_{i \in I} out(P_i)$ and $out(A) = \cup_{i \in I} out(A_i)$, we immediately have that $out(P) \subseteq out(A)$. By definition, $in(P) = \cup_{i \in I} in(P_i) - \cup_{i \in I} out(P_i)$ and $in(A) = \cup_{i \in I} in(A_i) - \cup_{i \in I} out(A_i)$. From assumption we have that $in(P) \cap out(A) = \emptyset$. Hence, $in(P) \subseteq in(A)$.

We now show that $traces(A)|acts(P) \subseteq traces(P)$. Suppose that $\beta \in traces(A)$. From Theorem 2.1(2) we have that for every i, $\beta|A_i \in traces(A_i)$ and by hypothesis we have that $(\beta|A_i)|acts(P_i) \in traces(P_i)$. Since $acts(P_i) \subseteq ext(A_i)$, $(\beta|A_i)|acts(P_i) = \beta|acts(P_i)$. Thus, $\beta|acts(P_i) \in traces(P_i)$ for every i. Since $acts(P_i) \subseteq acts(P)$ we have that $\beta|acts(P_i) = (\beta|acts(P))|acts(P_i)$. Thus, from the definition of P, $\beta|acts(P) \in traces(P)$.

The proof of 2 is analogous to the above, using Theorem 2.1(5). ■

Theorem 2.3 (Compositionality of Path Properties) *Let $\{A_i\}_{i \in I}$ be a compatible collection of automata and let $A = \Pi_{i \in I} A_i$. Also, let $\{P_i\}_{i \in I}$ be a collection of path properties and let $P = \Pi_{i \in I} P_i$.*

1. *If $states(P_i) = states(A_i)$ and $paths(A_i) \subseteq paths(P_i)$ for every i, then $states(P) = states(A)$ and $paths(A) \subseteq paths(P)$.*

2. *If $states(P_i) = states(A_i)$ and $fairpaths(A_i) \subseteq paths(P_i)$ for every i, then $states(P) = states(A)$ and $fairpaths(A) \subseteq paths(P)$.*

Proof. Consider 1. Since $states(P) = \Pi_{i \in I} states(P_i)$ and $states(A) = \Pi_{i \in I} states(A_i)$ we have from the hypothesis that $states(P) = states(A)$. Suppose that $\gamma \in paths(A)$. From Theorem 2.1(3) we have that for every i, there exists $\gamma' \trianglelefteq \gamma|A_i$ such that $\gamma' \in paths(A_i)$ and thus, by hypothesis, $\gamma' \in paths(P_i)$. Now, directly from the definition of P, $\gamma \in paths(P)$.

The proof of 2 is analogous to the above, using Theorem 2.1(6). ■

2.3 Abstraction Theory

In this section we formalize the conditions required in order for one (the abstract) verification problem (B, P') to be property-preserving with respect to another (the concrete) problem (A, P). That is we provide conditions for the following to hold:

$$B \textit{ satisfies } P' \text{ implies } A \textit{ satisfies } P$$

Our conditions for property-preservation for trace problems and path problems are the existence of *trace simulations* and *path simulations*, respectively, between the automata in the involved verification problems. These simulations can be seen as generalizations of standard forward simulations [Lyn96]. In Section 2.3.1 we define the notions of trace and path simulations and we show that the simulations are sound with respect to generalized notions of trace and path inclusion. These soundness results provide the basis for our theorems of Section 2.3.2, which provide conditions for property-preservation between verification problems in both the trace and the path setting and with respect to safety as well as liveness properties.

2.3.1 Simulations

We first define the notion of a trace simulation between two automata. The notion is relative to a relation between the external actions of the automata. Intuitively, this relation defines an action abstraction. Let A and B be two automata and let R be a relation from $ext(A)$ to $ext(B)$. In a trace simulation from A to B *parameterized by* R, we require that any transition of A on an action a in the domain of R can be matched in B by an execution fragment with a single external action b in the range of R, such that $(a, b) \in R$. We will see that the existence of a trace simulation from A to B parameterized by R is a sound condition for trace inclusion, *relative to* R.

Let A and B be two automata and let R be a relation from $ext(A)$ to $ext(B)$. We write, $u \overset{b}{\Longrightarrow}_B u'$, or simply $u \overset{b}{\Longrightarrow} u'$ when B is clear from the context, to denote that B has a finite execution fragment α with $first(\alpha) = u$, $last(\alpha) = u'$ and $trace(\alpha)|ran(R) = b$.

Definition 2.11 (Trace Simulation) *Let A and B be two I/O automata and let R be a relation from $ext(A)$ to $ext(B)$. A relation S from $states(A)$ to $states(B)$ is a trace simulation from A to B parameterized by R provided,*

1. *If $s \in start(A)$ then $S[s] \cap start(B) \neq \emptyset$.*
2. *If $s \xrightarrow{a} s'$, $(s, u) \in S$, and s and u are reachable states of A and B respectively, then*
   - (a) *If $a \in dom(R)$, then $\exists b, u'$ such that $u \overset{b}{\Longrightarrow} u'$, $(a, b) \in R$ and $(s', u') \in S$.*
   - (b) *If $a \notin dom(R)$, then $\exists u'$ such that $u \overset{\lambda}{\Longrightarrow} u'$ and $(s', u') \in S$.*

We write $A \leq^{\mathrm{t}}_R B$, if there exists a trace simulation from A to B parameterized by R. We write $A \leq^{\mathrm{t}}_R B$ via S, if S is a trace simulation from A to B parameterized by R.

We now define the notion of a path simulation between two automata. Our notion of path simulation can be seen as a generalization of the notion of *homomorphic abstraction functions* used in for example [CGL92]. We will show that the existence of a path simulation from an automaton A to an automaton B *via* a state relation S, will be a sound condition for path inclusion, relative to S.

Definition 2.12 (Path Simulation) *Let A and B be two I/O automata. A relation S from $states(A)$ to $states(B)$ is a path simulation relation from A to B provided, $dom(S) = states(A)$ and $ran(S) = states(B)$ and,*

1. *If $s \in start(A)$ then $S[s] \cap start(B) \neq \emptyset$.*
2. *If $s \longrightarrow s'$, $(s,u) \in S$, and s and u are reachable states of A and B respectively, then $\exists u'$ such that $u \longrightarrow u'$ and $(s',u') \in S$.*

We write $A \leq^{\mathrm{p}} B$, if there is a path simulation from A to B. We write $A \leq^{\mathrm{p}} B$ via S, if S is a path simulation from A to B.

Soundness of Simulations

In the following we first introduce an *Execution Correspondence Theorem.* This theorem states that if any of the simulations defined in the previous subsection has been proven from a concrete automaton to an abstract automaton, then for any execution of the concrete automaton, there is a "corresponding" execution of the abstract automaton. Our theorem is a minor variation of one in [GSSL93]. In order to formalize our notion of correspondence, the notions of (S,R)*-relation,* S*-relation*, and *index mapping* are introduced.

Definition 2.13 ((S,R)-relation and index mappings) *Let A and B be automata, and let S be a relation from $states(A)$ to $states(B)$ and R a relation from $ext(A)$ to $ext(B)$. Furthermore, let α and α' be executions of A and B, respectively, such that $\alpha = s_0a_1s_1a_2s_2\ldots$ and $\alpha' = u_0b_1u_1b_2u_2\ldots$. We say that α and α' are (S,R)-related, written $(\alpha,\alpha') \in (S,R)$, if there exists a total, nondecreasing mapping $m : \{0,1,\ldots,|\alpha|\} \mapsto \{0,1,\ldots,|\alpha'|\}$ such that*

1. *$m(0) = 0$,*
2. *$(s_i, u_{m(i)}) \in S$ for all i, $0 \leq i \leq |\alpha|$,*
3. *$trace(a_i)|dom(R) \in R^{-1}(trace(b_{m(i-1)+1}\ldots b_{m(i)})|ran(R))$ for all i, $0 < i \leq |\alpha|$, and*
4. *for all j, $0 \leq j \leq |\alpha'|$, there exists an i, $0 \leq i \leq |\alpha|$, such that $m(i) \geq j$.*

The mapping m is referred to as an index mapping from α to α' with respect to (S, R). We write $(A, B) \in (S, R)$ if for every execution α of A, there exists an execution α' of B such that $(\alpha, \alpha') \in (S, R)$.

If $R = \emptyset$ and m is the identity, we say that α and α' are S-corresponding, written $(\alpha, \alpha') \in S$, and we refer to m as an index mapping from α to α' with respect to S. We write $(A, B) \in S$ if for every execution α of A, there exists an execution α' of B such that $(\alpha, \alpha') \in S$.

Given the above notions of (S, R)-relation and S-relation we can now state the Execution Correspondence Theorem as follows.

Theorem 2.4 (Execution Correspondence Theorem) *Let A and B be automata. Assume that $A \leq^{\mathrm{t}}_{R} B$ via S (or $A \leq^{\mathrm{p}} B$ via S). Then $(A, B) \in (S, R)$ $((A, B) \in S)$.*

Proof. Analogous to the proof of Theorem 6.11 in [GSSL93]. ■

The Execution Correspondence Theorem can be used to prove the following soundness results of the trace and path simulations with respect to *relativized* trace and path inclusion. The soundness results relies on the following lemma.

Lemma 2.1 *Let A and B be automata and let S be a relation from states (A) to states (B) and R a relation from ext (A) to ext (B). Assume that $(\alpha, \alpha') \in (S, R)$ (or $(\alpha, \alpha') \in S$ and ran (S) = states (B)) and let m be an index mapping from α to α' with respect to (S, R) (with respect to S). Then, for all $0 \leq i \leq |\alpha|$,*

$$\mathit{trace}\,({}_i|\alpha)|\mathit{dom}\,(R) \in R^{-1}(\mathit{trace}\,({}_{m(i)}|\alpha')|\mathit{ran}\,(R))$$

$$(\mathit{path}\,({}_i|\alpha) \in S^{-1}(\mathit{path}\,({}_{m(i)}|\alpha'))\,)$$

Proof. Analogous to the proof Lemma 6.14 in [GSSL93]. ■

We can now prove the following soundness results for trace and path simulations.

Theorem 2.5 (Soundness of Trace Simulations) *Let A and B be automata.*

$$\text{If } A \leq^{\mathrm{t}}_{R} B \text{ via } S \text{ then, } \mathit{traces}(A)|\mathit{dom}\,(R) \subseteq R^{-1}(\mathit{traces}(B)|\mathit{ran}\,(R))$$

Proof. Let $\beta \in \mathit{traces}(A)|\mathit{dom}\,(R)$ and let α be an execution of A such that $\beta = \mathit{trace}\,(\alpha)|\mathit{dom}\,(R)$. Then, by Theorem 2.4, there exists an execution α' of B such that $(\alpha, \alpha') \in (S, R)$. For any execution α'' we have that ${}_0|\alpha'' = \alpha''$, and for any index mapping m, $m(0) = 0$. Thus, from Lemma 2.1, $\mathit{trace}\,(\alpha)|\mathit{dom}\,(R) \in R^{-1}(\mathit{trace}\,(\alpha')|\mathit{ran}\,(R))$. We thus have that $\beta \in R^{-1}(\mathit{traces}(B)|\mathit{ran}\,(R))$ as required. ■

Theorem 2.6 (Soundness of Path Simulations) *Let A and B be automata.*

$$\text{If } A \leq^{\mathrm{p}} B \text{ via } S \text{ then, } paths\,(A) \subseteq S^{-1}(paths\,(A'))$$

Proof. Analogous to proof of Theorem 2.5. ■

2.3.2 Preservation Conditions

We end this section by stating our conditions for property preservation in the following four theorems. The theorems provide preservation conditions for properties of type: trace safety, path safety, trace liveness, and path liveness.

Theorem 2.7 (Trace Safety Preservation) *Let (A, P) and (B, P') be two trace safety verification problems. Let R be a relation with $dom\,(R)$ $= acts(P)$ and $ran\,(R) = acts(P')$, such that $R^{-1}(traces(P')) \subseteq traces(P)$. If,*

1. *$A \leq^{\mathrm{t}}_{R} B$, and*
2. *B satisfies P'*

then, A satisfies P.

Proof. From *1* we have that, $traces(A)|dom\,(R) \subseteq R^{-1}(traces(B)|ran\,(R))$. By assumption $dom\,(R) = acts(P)$ and $ran\,(R) = acts(P')$. Thus, we have that $traces(A)|acts(P) \subseteq R^{-1}(traces(B)|acts(P'))$. From *2* we have that $traces(B)|acts(P') \subseteq traces(P')$ and since $ran\,(R) = acts(P')$ this implies, $R^{-1}(traces(B)|acts(P')) \subseteq R^{-1}(traces(P'))$. By assumption we have that $R^{-1}(traces(P')) \subseteq traces(P)$. Thus, by transitivity of $\subseteq$, we conclude that $traces(A)|acts(P) \subseteq traces(P)$. ■

Theorem 2.8 (Path Safety Preservation) *Let (A, Q) and (B, Q') be two path safety verification problems. Let S be a relation with $dom\,(S) = states\,(Q)$ and $ran\,(S) = states\,(Q')$ such that $S^{-1}(paths\,(Q')) \subseteq paths\,(Q)$. If,*

1. *$A \leq^{\mathrm{p}} B$ via S, and*
2. *B satisfies Q'*

then, A satisfies Q.

Proof. Analogous to proof of Theorem 2.7. ■

Theorem 2.9 (Trace Liveness Preservation) *Let (A, P) and (B, P') be two trace liveness verification problems. Let R be a relation with $dom\,(R) =$ $acts(P)$ and $ran\,(R) = acts(P')$, such that $R^{-1}(traces(P')) \subseteq traces(P)$. If,*

1. *$A \leq^{\mathrm{t}}_{R} B$ via S,*
2. *for all $(\alpha, \alpha') \in (S, R)$: $\alpha \in \mathit{fairexecs}(A) \Rightarrow \alpha' \in \mathit{fairexecs}(B)$, and*
3. *B satisfies P'*

then, A satisfies P.

Proof. First prove $\mathit{fairtraces}(A)|\mathit{dom}(R) \subseteq R^{-1}(\mathit{fairtraces}(B)|\mathit{ran}(R))$. Let $\beta \in \mathit{fairtraces}(A)|\mathit{dom}(R)$. Then there is an execution α of A such that $\alpha \in \mathit{fairexecs}(A)$ and $\beta = \mathit{trace}(\alpha)|\mathit{dom}(R)$. From *1* and Theorem 2.4 (ECT), there exists an execution α' of B such that $(\alpha, \alpha') \in (S, R)$. From *2*, $\alpha' \in \mathit{fairexecs}(B)$ and from Lemma 2.1 we get that, $\mathit{trace}(\alpha)|\mathit{dom}(R)$ $\in R^{-1}(\mathit{trace}(\alpha')|\mathit{ran}(R))$. This implies that $\beta \in R^{-1}(\mathit{trace}(\alpha')|\mathit{ran}(R))$ and since $\alpha' \in \mathit{fairexecs}(B)$, we have that $\beta \in R^{-1}(\mathit{fairtraces}(B)|\mathit{ran}(R))$. Thus, $\mathit{fairtraces}(A)|\mathit{dom}(R) \subseteq R^{-1}(\mathit{fairtraces}(B)|\mathit{ran}(R))$. Since $\mathit{dom}(R)$ $= \mathit{acts}(P)$ and $\mathit{ran}(R) = \mathit{acts}(P')$ we have that $\mathit{fairtraces}(A)|\mathit{acts}(P) \subseteq R^{-1}(\mathit{fairtraces}(B)|\mathit{acts}(P'))$. From *3*, $\mathit{fairtraces}(B)|\mathit{acts}(P') \subseteq \mathit{traces}(P')$ and since $\mathit{ran}(R) = \mathit{acts}(P')$ this implies, $R^{-1}(\mathit{fairtraces}(B)|\mathit{acts}(P')) \subseteq R^{-1}(\mathit{traces}(P'))$. By assumption we have that, $R^{-1}(\mathit{traces}(P')) \subseteq \mathit{traces}(P)$ and by transitivity of $\subseteq$, we conclude that $\mathit{fairtraces}(A)|\mathit{dom}(R) \subseteq \mathit{traces}(P)$. ■

Theorem 2.10 (Path Liveness Preservation) *Let (A, Q) and (B, Q') be two path liveness verification problems. Let S be a relation with $\mathit{dom}(S) = \mathit{states}(Q)$ and $\mathit{ran}(S) = \mathit{states}(Q')$ such that $S^{-1}(\mathit{paths}(Q')) \subseteq \mathit{paths}(Q)$. If,*

1. *$A \leq^{\mathrm{p}} B$ via S,*
2. *for all $(\alpha, \alpha') \in S$: $\alpha \in \mathit{fairexecs}(A) \Rightarrow \alpha' \in \mathit{fairexecs}(B)$, and*
3. *B satisfies Q'*

then, A satisfies Q.

Proof. Analogous to proof of Theorem 2.9. ■

2.4 Abstraction Theory in Larch

In this section we present a formalization using the Larch tool set of the safety part of the trace abstraction framework introduced in the preceding section. In [SAGG$^+$93] a framework is introduced for specifying and reasoning about I/O Automata using the Larch tool set. We extend this framework to include a formalization of our trace abstraction theory. The theory is formalized in the Larch Shared Language (LSL) [GH93] which is supported by a tool that produces input for LP, the Larch Prover. LP is a theorem prover for multi-sorted first-order logic designed to assist users who employ standard proof techniques such as proofs by cases, induction, and contradiction.

2.4.1 I/O Automata in LSL

LSL specifications define two kinds symbols, *operators* and *sorts*. Operators name total functions from tuples of values to values. Sorts name disjoint non-empty sets of values indicating the domains and ranges of operators. Operators and sorts are introduced in *trait*s. A trait is the basic unit of specification in LSL and it can be seen as somewhat similar to the definition of an abstract data type in many algebraic specification languages. However, traits need not fully define a type.

Figure 2.1 shows the trait specifying an I/O automaton `A`. The trait begins with the `introduces` clause, declaring a set of operators by providing a *signature* for each. Signatures implicitly defines sorts for domains and ranges of operators and they are used to sort-check terms. For example, the signature for operator `start` implicitly defines the sort `States[A]` of states of `A`. The *body* of a trait follows the reserved word `asserts`. In the `asserts` clause the introduced operators are constrained by equations. For example, the execution fragments of automaton `A` are defined to be those elements of sort `StepSeq[A]` that satisfy the predicate `execFrag`, which is defined inductively in the `asserts` clause.

Each trait defines a *theory* (a set of sentences closed under logical consequences) in multisorted first-order logic with equality. Each theory contains the trait's assertions, the conventional axioms of first-order logic, everything that follows from them, and nothing else. The basic theory associated with `Automaton (A)` consists of the set of sentences that can be obtained from the assertions by equational rewriting. The equational theory is further strengthened by the `generated by` clause, that asserts that operators `empty` and `^` constitute a complete set of *generators* for sort `Traces[A]`. This justifies a *generator induction scheme* for proving things about the sort.

2.4.2 Trace Simulations in LSL

In order to complete our formalization of the trace based abstraction theory, we introduce two traits specifying the required theory of action parameterized simulation relations.

In Figure 2.2 the trait `ActRel` formalizing the notion of a relation between the external actions of automata and related operators is presented. The trait begins by an `includes` clause which is a means of combining theories. The theory associated with `ActRel` is the theory associated with the union of theories from the included traits, `Automaton(A)` and `Automaton(B)`,

```
Automaton (A): trait
  introduces
    start       : States[A]                         -> Bool
    enabled     : States[A], Actions[A]             -> Bool
    effect      : States[A], Actions[A], States[A]  -> Bool
    isExternal  : Actions[A]                        -> Bool
    isInternal  : Actions[A]                        -> Bool
    isStep      : States[A], Actions[A], States[A]  -> Bool
    null        : States[A]                         -> StepSeq[A]
    __{__,__}   : StepSeq[A], Actions[A], States[A] -> StepSeq[A]
    execFrag    : StepSeq[A]                        -> Bool
    first, last : StepSeq[A]                        -> States[A]
    empty       :                                   -> Traces[A]
    __ ^ __     : Traces[A], Actions[A]             -> Traces[A]
    trace       : Actions[A]                        -> Traces[A]
    trace       : StepSeq[A]                        -> Traces[A]
    inv         : States[A]                         -> Bool

  asserts
   sort Traces[A] generated by empty, ^
    \forall s, s': States[A], a, a': Actions[A], ss: StepSeq[A]
      isInternal(a) <=> ~isExternal(a);
      isStep(s, a, s') <=> enabled(s, a) /\ effect(s, a, s');
      execFrag(null(s));
      execFrag(null(s){a,s'}) <=> isStep(s, a, s');
      execFrag((ss{a,s}){a',s'}) <=> execFrag(ss{a,s}) /\ isStep(s, a', s');
      first(null(s)) = s;
      last(null(s)) = s;
      first(ss{a,s}) = first(ss);
      last(ss{a,s}) = s;
      trace(null(s)) = empty;
      trace(ss{a,s}) = (if isExternal(a) then trace(ss) ^ a else trace(ss));
      trace(a) = (if isExternal(a) then empty ^ a else empty)
```

Figure 2.1: Automaton.lsl

```
ActRel(R,A,B): trait

  includes Automaton(A), Automaton(B)

  introduces
    R      : Actions[A], Actions[B]  -> Bool
    proR   : Traces[B]               -> Traces[B]
    inR    : Actions[A]              -> Bool

  asserts
    with a: Actions[A], a': Actions[B]
      inR(a) <=> (\E a' (R(a,a')));

    with tr: Traces[B], a': Actions[B], a: Actions[A]
      proR(empty) = empty;
      proR(empty^a') = (if \E a R(a,a') then empty^a' else empty);
      proR(tr^a') = (if \E a R(a,a') then proR(tr)^a' else proR(tr));
```

Figure 2.2: ActRel.lsl

and the assertions of `ActRel` itself. Note that `Automaton(B)` defines a theory disjoint (disjoint sorts with new operators) from that of `Automaton(A)` by renaming the parameter `A` by `B`. The trait introduces a relation `R` as a relation from the actions of automaton `A` to those of automaton `B`. It further defines operators `proR` on traces of `B` and `inR` on actions of `A`. Operator `proR` *projects* a trace of `B` on to the actions in the range of relation R. Operator `inR` is a predicate telling whether or not an action of `A` is in the domain of relation `R`.

In Figure 2.3 we show the trait `Simulation` specifying the theory of trace simulation relations. The trait `assumes` (similar to `includes`) the theory of two automata `A` and `B` as well as a relation `R` from the actions of `A` to those of `B`. The trait further introduces an operator `S` to denote a relation among states of `A` and `B`. The assertions of the trait are simply a formalization of the conditions in the definition of a trace simulation relation. The theory can be used to assist in proving that a given state relation `S` between two automata `A` and `B` is a trace simulation parameterized by relation `R` from `A` to `B`. In Chapter 4, in the second part of this thesis, we demonstrate the use of LP to discharge preservation conditions in an abstraction proof for Burns' mutual exclusion algorithm.

2.5 Input/Output Automata in SPIN

An abstract verification problem which is finite-state is directly amenable to automatic verification. We end this chapter on our untimed abstraction

```
Simulation(A,B,R,S): trait

  assumes Automaton(A), Automaton(B), ActRel(R,A,B)

  introduces
    S      : States[A], States[B]    -> Bool

  asserts
    with s, s' : States[A], u: States[B], a: Actions[A], a',a'': Actions[B],
         alpha: StepSeq[B]
    start(s) => \E u (start(u) /\ S(s, u));

    S(s, u) /\ isStep(s, a, s') /\ inR(a) =>
        (\E alpha \E a' (execFrag(alpha) /\ first(alpha) = u /\
                        S(s', last(alpha)) /\ proR(trace(alpha)) = empty^a'
                        /\ R(a,a')));

    S(s, u) /\ inv(s) /\ isStep(s, a, s') /\ ~inR(a)  =>
        (\E alpha (execFrag(alpha) /\ first(alpha) = u /\ S(s', last(alpha))
                  /\ proR(trace(alpha)) = empty))
```

Figure 2.3: Simulation.lsl

framework by examining a translation scheme for representing finite-state verification problems from the I/O automata framework in the model checker SPIN [Hol91]. The SPIN model checker supports automatic verification of next-time-free LTL properties over finite-state transition systems described in the PROMELA language. We begin in Section 2.5.1 by examining a method for specifying trace and path properties using LTL. Then in Section 2.5.2 we present a rudimentary scheme for translating finite-state I/O automata into the PROMELA input language for SPIN.

2.5.1 Temporal Logic

In this section we present a method for specifying trace and path properties using Linear Time Temporal Logic (LTL). We will assume, that any I/O automaton A has a set of *state variables* $\mathcal{V}_A$ over some domain. We will interpret a state of A as a mapping s from $\mathcal{V}_A$ to its domain. We write $s(v)$ to denote the value of v in state s. We will further assume a reserved *action variable* v_{act} ranging over $acts(A) \cup \{\epsilon\}$, where ϵ is a distinct "no-action" symbol. We assume a basic assertion language over variables $\mathcal{V}_A \cup \{v_{act}\}$. Assertions are constructed from basic expressions $v = x$ (x in the domain of v) using only standard boolean connectives.

For an assertion p and a function s that interprets all free variables in p, we write $s \models p$ to denote that s *satisfies* p. Meaning, that the formula obtained by substituting in p all variables v by $s(v)$, is true.

Definition 2.14 (Temporal Formulae) *Let A be an automaton. A temporal formula of A is a formula constructed from assertions of A to which we apply the boolean connectives $\neg$ and $\vee$ and the basic temporal operator $\mathcal{U}$ denoted (Strong) Until.*

For an I/O automaton A, we let $seq(A)$ denote the set of all finite and infinite alternating sequences of states and actions in A. Any sequence must begin with a state and if it is finite it must also end with a state. Note, that the set of executions $execs(A)$ is a subset of the set $seq(A)$. The *trace* of a sequence $\sigma \in seq(A)$ consists of the subsequence of external actions in σ.

Definition 2.15 (Stutter Extending Sequences) *Let σ be a finite sequence in $seq(A)$ such that $\sigma = s_0 a_1 s_1 \ldots a_k s_k$. We define the stutter extension of σ as the infinite sequence,*

$$\sigma_{st} = s_0 a_1 s_1 \ldots a_k s_k \epsilon s_k \epsilon s_k \ldots$$

obtained by concatenating σ with the sequence consisting of an infinite alternation of the "no-action" ϵ and the last state s_k of σ. For an infinite sequence σ we define $\sigma_{st} = \sigma$.

Definition 2.16 (Encoding Actions in State Sequences) *Let σ be a sequence in $seq(A)$ such that $\sigma = s_0 a_1 s_1 a_2 s_2 \ldots$. Then $\tilde{\sigma}$ denotes the sequence obtained from σ by encoding in any state the information of the immediate preceding action in the reserved action variable v_{act}. Formally,*

$$\tilde{\sigma} = (s_0 \cup [v_{act} \mapsto \epsilon])(s_1 \cup [v_{act} \mapsto a_1])(s_2 \cup [v_{act} \mapsto a_2]) \ldots$$

where ϵ denotes a distinct "no-action" not in $acts(A)$.

Definition 2.17 (Sequences Satisfying Formulae ($\models$)) *Let A be an automaton and let p be a temporal formula of A. Furthermore, let $\sigma \in seq(A)$. We define inductively the notion of p holding at a position $j \geq 0$ in $\tilde{\sigma}_{st}$, denoted by $(\tilde{\sigma}_{st}, j) \models p$, as follows.*

If p assertion:

$$(\tilde{\sigma}_{st}, j) \models p \iff s_j \Vdash p$$

Otherwise:

$$\begin{array}{lcl}
(\tilde{\sigma}_{st}, j) \models \neg p & \iff & (\tilde{\sigma}_{st}, j) \not\models p \\
(\tilde{\sigma}_{st}, j) \models p \vee q & \iff & (\tilde{\sigma}_{st}, j) \models p \text{ or } (\tilde{\sigma}_{st}, j) \models q \\
(\tilde{\sigma}_{st}, j) \models p \,\mathcal{U}\, q & \iff & \text{there exists } k \geq j, (\tilde{\sigma}, k) \models q \text{ and} \\
 & & \text{for all } i, j \leq i < k, (\tilde{\sigma}, i) \models p
\end{array}$$

We say that σ satisfies p iff $(\tilde{\sigma}_{st}, 0) \models p$. We will write $\sigma \models p$ in this case.

We can introduce additional derived operators. Boolean operators $\wedge$ and $\rightarrow$ may be defined in the usual way, using the basic $\vee$ and $\neg$ operators. Additional temporal operators are defined by:

$$\begin{array}{lcll} \Diamond p & = & \mathit{true}\,\mathcal{U}\,p & \text{– } \textit{Eventually } p \\ \Box p & = & \neg\Diamond\neg p & \text{– } \textit{Always } p \\ p\,\mathcal{W}\,q & = & \Box p \vee p\,\mathcal{U}\,q & \text{– } p \textit{ Waiting-for (Unless) } q \end{array}$$

Definition 2.18 (Induced Trace and Path Properties) *Let A be an automaton. Let p be an arbitrary temporal formula of A. Define,*

1. $\mathit{traces}_A(p) = \{\mathit{trace}\,(\sigma) \mid \sigma \in \mathit{seq}\,(A) \wedge \sigma \models p\}$
2. $\mathit{paths}_A\,(p) = \{\mathit{path}\,(\sigma) \mid \sigma \in \mathit{seq}\,(A) \wedge \sigma \models p\}$

The trace property induced by p, written $\mathcal{T}_A(p)$, is the trace property with signature $\mathit{sig}\,(\mathcal{T}_A(p)) = (\mathit{in}\,(A), \mathit{out}\,(A))$ and set of traces $\mathit{traces}(\mathcal{T}_A(p)) = \mathit{traces}_A(p)$. The path property induced by p, written $\mathcal{P}_A(p)$, is the path property with signature $\mathit{sig}\,(\mathcal{P}_A(p)) = \mathit{states}\,(A)$ and set of paths $\mathit{paths}\,(\mathcal{P}_A(p)) = \mathit{paths}_A\,(p)$.

We say that a temporal formula p of A is a *trace* (*path*) *safety formula* if $\mathcal{T}_A(p)$ ($\mathcal{P}_A(p)$) is a trace (path) safety property. Analogously, p is said to be a *trace* (*path*) *liveness* formula if $\mathcal{T}_A(p)$ ($\mathcal{P}_A(p)$) is a trace (path) liveness property. In [Pnu86] is a characterization of temporal logic properties according to notions of safety and liveness. Our notion of liveness corresponds to temporal properties commonly called *pure liveness properties*.

Usually, a temporal formula intended to specify a trace property of an automaton A, will only contain assertions over the action variable v_{act}. Analogously, a temporal formula intended to specify a path property, will usually contain only assertions over the state variables $\mathcal{V}_A$ of A.

Definition 2.19 (Fairness Condition in Temporal Logic) *Let A be an automaton. The fairness property for A is the property $\mathcal{F}_A$ defined as,*

$$\mathcal{F}_A = \bigwedge_{C \in \mathit{tasks}(A)} (\Diamond\Box\mathcal{E}_A(C) \rightarrow \Box\Diamond(v_{act} \in C))$$

where $\mathcal{E}_A(C)$ is a predicate describing states of A in which some action from C is enabled.

Lemma 2.2 *Let A be an automaton and let α be any execution of A. Then,*

$$\alpha \models \mathcal{F}_A \iff \alpha \in \mathit{fairexecs}\,(A)$$

Proof. $\Longrightarrow$: Suppose α is infinite and $\alpha \notin \mathit{fairexecs}(A)$. Then α contains only finitely many occurrences of actions from C and only finitely many occurrences of states in which C is not enabled. Thus α satisfies the hypothesis of $\mathcal{F}_A$ but not the conclusion. Hence, $\alpha \not\models \mathcal{F}_A$. Suppose α is finite. Assume for the sake of contradiction that an action $a \in C$ (for some task C) is enabled in the last state s_k of α. Then $(\tilde{\alpha}_{st}, 0) \models \Diamond\Box\mathcal{E}_A(C)$ since s_k is repeated infinitely in α_{st}. Therefore, since $\alpha \models \mathcal{F}_A$ $((\tilde{\alpha}_{st}, 0) \models \mathcal{F}_A)$, it must be that infinitely often in $\tilde{\alpha}_{st}$ there exists a state in which the $v_{act} \in C$. This, however contradicts the fact that by construction $\tilde{\alpha}_{st}$ has an infinite suffix of states in which $v_{act} = \epsilon$. Thus, no C is enabled in s_k.

$\Longleftarrow$: Suppose α is infinite and $\alpha \in \mathit{fairexecs}(A)$. Suppose α contains infinitely many occurrences of action from some task C. Then the conclusion of $\mathcal{F}_A$ holds and thus $\alpha \models \mathcal{F}_A$. Suppose α contains infinitely many occurrences of states in which C is not enabled. Then the hypothesis of $\mathcal{F}_A$ is false and $\alpha \models \mathcal{F}_A$ vacuously. Suppose α is finite. Then no task C is enabled in the last state s_k of α. Since s_k is repeated infinitely in α_{st}, $(\tilde{\alpha}_{st}) \not\models \Diamond\Box\mathcal{E}_A(C)$ for any C. Hence, $\alpha \models \mathcal{F}_A$ vacuously. ■

We now formally define the notion of an I/O automaton A satisfying a temporal formula as follows.

Definition 2.20 (Automata Satisfying Formulae) *Let A be an automaton and let p be any safety formula and q any liveness formula. Then,*

$$\begin{aligned} A \models p &\iff \forall \alpha \in \mathit{execs}(A).\ \alpha \models p \\ A \models q &\iff \forall \alpha \in \mathit{execs}(A).\ \alpha \models \mathcal{F}_A \rightarrow q \end{aligned}$$

It follows from the definition that, if $A \models p$ then A satisfies $\mathcal{T}_A(p)$ and $\mathcal{P}_A(p)$.

2.5.2 Translating Automata

Our translation scheme is based on the representation of I/O automata in the precondition-effect language described in Section 2.2.2. We will assume that the state types, predicates, and operations used to describe states, preconditions and effects of I/O automata are all implementable in the PROMELA language. Thus, our translation scheme is relative to a correct translation of the above elements. In theory, only a fragment of the very general meta I/O automata language is implementable in the PROMELA language. In practice however, many nontrivial algorithms are describable in a fragment of the I/O automata language that allows for translation.

We consider the general translation of a composition $A = \Pi_{i \in I} A_i$ of automata. Figure 2.4 presents the generic code for any automaton A_i in the composition. We assume that pre(A_i, a) defines the precondition for action a of automaton A_i. Analogously, we assume that eff(A_i, a) defines the effect of action a.

Automaton: A_i

Signature
Inputs:
  $a_{i1}, a_{i2}, \ldots$
Internals:
  $b_{i1}, b_{i2}, \ldots$
Outputs:
  $c_{i1}, c_{i2}, \ldots$

States:
$x_{i1} : T_{i1}, x_{i2} : T_{i2}, \ldots$

Transitions:

```
...

input: $a_{ij}$
    Eff: eff($A_i, a_{ij}$)

...

internal: $b_{ik}$
    Pre: pre($A_i, b_{ik}$)
    Eff: eff($A_i, b_{ik}$)

...

output: $c_{il}$
    Pre: pre($A_i, c_{il}$)
    Eff: eff($A_i, c_{il}$)

...
```

Tasks:
$\{C_{i1}, C_{i2}, \ldots\}$

Figure 2.4: I/O Automaton A_i

```
proctype $A$ (){
 mtype $=\{in(A), int(A), out(A)\}$;
 mtype $v_{act}$;
 $T_{11}\ x_{11}; \ldots ; T_{nm}\ x_{nm}$
 do
 ...
 :: atomic $\{true -> v_{act} = a_{ij}; \text{eff}(A_i, a_{ij})\}$
 ...
 :: atomic $\{\text{pre}(A_i, b_{ik}) -> v_{act} = b_{ik}; \text{eff}(A_i, b_{ik})\}$
 ...
 :: atomic $\{\text{pre}(A_i, c_{il}) -> v_{act} = c_{il}; \text{eff}(A_1, c_{il}); \ldots ; \text{eff}(A_n, c_{il})\}$
 ...
 :: else $->$ $v_{act} = \epsilon$ ; break
 od}
```

Figure 2.5: PROMELA code for automaton A

We translate the composition A into a single PROMELA proctype declaration as shown in Figure 2.5. We use `typewriter` font for PROMELA syntax. We use the syntax of the precondition-effect code to describe the PROMELA implementations of state and action variables of A. We also use the notions $\text{pre}(A_i, a)$ and $\text{eff}(A_i, a)$ to denote the PROMELA implementations of the identically named predicates and operations of the precondition-effect code. Recall, that we assume that a correct translation of these entities exists.

The PROMELA code starts by declaring a process A with no parameters. The process declaration is divided into two parts. The first part consists of the first three lines inside the `proctype` declaration and it declares the variables to be used in process A. In the first line we declare a message type `mtype` to be a list of all the actions of the composition A, and in the next line we declare the variable v_{act} to be of this type. The variable v_{act} implements the action variable described in Section 2.5.1 and is used to keep track of the latest action performed by A. Recall from Section 2.5.1 that we use v_{act} to describe trace properties by LTL properties. The third line ends the declaration part of the translation. Here we simply declare all the state variables of the composition A.

The transitions of automaton A are described in a single `do::od` loop construction. The loop has an entry for any action in the composition A. Consider the actions of automaton A_i and let us see how these are represented in the PROMELA description of the composition A.

Consider an input action a_{ij} of A_i. If no other process A_k contains a_{ij} as an output action, a_{ij} becomes an input in the composition A. In this case the effect of a_{ij} in A becomes the effect of a_{ij} in A_i. This situation is the one described for action a_{ij} in Figure 2.5. The precondition and effect of the action is encapsulated in an `atomic` sequence which guarantees the execution of precondition and effect as a single indivisible statement. The `atomic` sequence is divided into two parts, a *guard* and an *effect*. The guard in this case in the statement *true*, describing the input-enabledness of action a_{ij}. The effect consists of assigning to v_{act} the name of the action performed, as well as performing the effect of a_{ij} in A_i.

If some process A_k contains a_{ij} as output action, a_{ij} becomes an output in A. This situation is described in Figure 2.5 for an output action c_{ij} of automaton A_i. Since A_i is the only automaton controlling c_{ij} in A, the precondition in A becomes the precondition from A_i. The effect however becomes the sequence of effects that action c_{ij} may have by virtue of being an input action in other automata than A_i, as well as the effect of c_{ij} in A_i. As before, the effect also contains the assignment to action variable v_{act} of the name of the action c_{ij} performed.

Finally, consider an internal action b_{ik} of A_i. This action becomes an internal action of A with preconditions and effects identical to those of A_i.

The PROMELA code for A contains a final entry in the loop construction. This entry, guarded by the statement `else`, is executable exactly if none of the other entries are. The effect of the entry simply consists of assigning the "no-action" ϵ to the action variable v_{act} and then breaking out of the loop construct. We use this entry to guarantee the correct stutter semantics of automata as it is described in Section 2.5.1. When verifying LTL properties, SPIN automatically stutters the last state in any finite execution to obtain an infinite execution. The entry described above guarantees that $v_{act} = \epsilon$ in any stutter extension of a finite execution.

The tasks of automaton A are not directly code into the PROMELA code for A. Rather, we code the fairness conditions induced by the task partition into liveness properties as described in Section 2.5.1.

Chapter 3

Timed Abstraction Framework

In this chapter we present our timed abstraction framework. Analogous to the untimed framework, presented in the previous chapter, the timed framework provides general conditions for one verification problem to be property preserving with respect to another problem. However, the systems that we consider in this framework are timed systems, and so the properties that interests us include timing information as well.

The timed framework developed in this thesis is motivated by practical experience with the UPPAAL real-time model checker [BLL$^+$95]. UPPAAL can efficiently deal with verification of real-time systems (over a dense time domain) specified as networks of timed automata. A timed automata is basically an extension of a classical automaton with real-valued clocks. The timed automaton model constrains the allowed conditions on clock variables in a way that makes it possible to obtain a finite abstract semantic model. The abstract model preserves (strongly) enough information to allow it to be used for verification about properties of the concrete system. Thus, the tool performs an automatic abstraction of the timing component of any properly described real-time system.

We provide an abstraction framework that in addition allows for the abstraction of untimed information like control and data. Thus our framework provides a link to the UPPAAL tool providing support for verification of e.g. parameterized real-time systems consisting of a number of composed processes, the particular value of the number being the parameter, or processes with unbounded number of actions or unbounded data domains.

The properties that can be directly verified in UPPAAL are simple *reachability* properties. However, based on the use of *test automata* verification of properties other than plain reachability ones may be carried out as well. In [ABL98] the authors describe the testing approach for properties expressed in a dense-time temporal logic suitable for specifying safety and bounded liveness properties. Given a property ϕ to model check, the user provides a test automaton T_ϕ for it. The test automaton must be such that the original system S has the property expressed by ϕ precisely when no *bad states* of T_ϕ can be reached in the composition of S and T_ϕ. In [ABL98] it is also presented how the logical property language of that paper can be used to provide *characteristic properties* [IS94] for timed automata with respect to a timed version of the ready simulation preorder [LS91, BIM95]. This provides for an indirect verification of timed ready simulation using the testing approach.

Our abstraction framework is based on a variant of the timed ready simulation preorder. We consider verification problems consisting of a pair of timed systems, an implementation and a specification, and our goal is to verify whether the implementation is timed ready simulated by the specification. Given two verification problems, a concrete one and an abstract one, we provide conditions for the abstract problem to be property preserving with respect to the concrete problem. Meaning that under these conditions, if the abstract implementation is timed ready simulated by the abstract specification then the concrete implementation is timed ready simulated by the concrete specification. The condition for property preservation is based on an action parameterized version of the timed ready simulation. We will see that this simulation has nice properties like e.g. preservation under system composition which supports hierarchical verification.

We furthermore provide a method for translating the problem of checking for the existence of timed ready simulations into a reachability question amenable to verification by UPPAAL. Given two systems S_1, S_2 we write $S_1 \preceq S_2$ if S_1 is timed ready simulated by S_2. By our testing approach, we construct a test automaton T_{S_2} for S_2 such that no bad states of the composition of S_1 and T_{S_2} can be reached precisely when $S_1 \preceq S_2$. Our approach is more direct than the one presented in [ABL98] in which a characteristic property of system S_2 must first be constructed and then in a second step a test automaton for this property must be constructed.

Chapter Organization. We begin in Section 3.1 by presenting the underlying formal model used to describe timed systems. We use a notion of timed labelled transition systems commonly used to provide semantics for timed automata. The timed automata language is presented in this section as well. In Section 3.2 we present the general conditions for property preservation between verification problems based on the notion of parameterized timed ready simulation. We also prove properties like compositionality and transitivity of parameterized timed ready simulations. Section 3.3 presents a method for constructing abstract verification problems from concrete problems such that the required conditions for property preservation are guaranteed to hold. Finally, in Section 3.4 we present the method of translating checks for the existence of timed ready simulations into a reachability problem.

3.1 Timed Labelled Transition Systems

In this section we present the basic model of timed labelled transition systems as well as the timed automaton language used to describe these transition systems syntactically.

A timed labelled transition systems has two types of labels: atomic actions and delays, representing discrete and continuous changes of real-time systems. We will assume that $\mathcal{A}_u$ and $\mathcal{A}_l$ are universal and disjoint sets of *urgent actions* and *lazy actions*, respectively. Urgent actions are used to enforce immediate synchronization among transition systems, in the sense that no delay can occur beyond a point in time at which an urgent synchronization becomes enabled. Lazy actions are simply all non-urgent actions. We let $\mathcal{A} = \mathcal{A}_u \cup \mathcal{A}_l$.

We will assume the existence of a special internal action τ distinct from any action in $\mathcal{A}$ and we define $\mathcal{A}_\tau = \mathcal{A} \cup \{\tau\}$, $\mathcal{A}_{u,\tau} = \mathcal{A}_u \cup \{\tau\}$ and $\mathcal{A}_{l,\tau} = \mathcal{A}_l \cup \{\tau\}$. We use $\mathcal{D}$ to denote the set of delay actions $\{\epsilon(d) \mid d \in \mathbf{R}_{\geq 0}\}$ where $\mathbf{R}_{\geq 0}$ denotes the set of non-negative real numbers. We denote the 0-delay action $\epsilon(0)$ by $\mathbf{0}$ and we define $\mathcal{A}_{\tau,\mathbf{0}} = \mathcal{A}_\tau \cup \{\mathbf{0}\}$, $\mathcal{A}_{u,\mathbf{0}} = \mathcal{A}_u \cup \{\mathbf{0}\}$ and $\mathcal{A}_{l,\mathbf{0}} = \mathcal{A}_l \cup \{\mathbf{0}\}$. We use a, b to range over $\mathcal{A}_{\tau,\mathbf{0}}$.

Definition 3.1 (Timed Labelled Transition System) *A timed labelled transition system is a tuple $\mathcal{T} = \langle S, s_0, \longrightarrow\rangle$ where S is a set of states, $s_0 \in S$ is the initial state, and $\longrightarrow \subseteq S \times \mathcal{A}_\tau \cup \mathcal{D} \times S$ is a transition relation satisfying the following properties:*

- (time determinism) *For every $s, s', s'' \in S$, if $s \xrightarrow{\epsilon(d)} s'$ and $s \xrightarrow{\epsilon(d)} s''$, then $s' = s''$.*
- (time additivity) *For every $s, s'' \in S$, $s \xrightarrow{\epsilon(d_1+d_2)} s''$ iff $s \xrightarrow{\epsilon(d_1)} s' \xrightarrow{\epsilon(d_2)} s''$, for some $s' \in S$.*
- (0-delay) *For every $s, s' \in S$, $s \xrightarrow{\mathbf{0}} s'$ iff $s = s'$*

We say that a timed labelled transition system is deterministic if, in addition, the transition relation satisfies the following property:

- (determinism) *For every $s, s', s'' \in S$ and $a \in \mathcal{A}_\tau$, if $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$, then $s' = s''$.*

Following [ABL98, Yi90], we now define versions of the transition relations that abstract away from the internal evolution of states. Let $\xrightarrow{\tau}^*$ denote the reflexive and transitive closure of $\xrightarrow{\tau}$. For any states s and s'

of a timed labelled transition system and for any action $a \in \mathcal{A}_\tau$, we write $s \xrightarrow{a}$ iff there exists s' such that $s \xrightarrow{a} s'$, and we write $s \overset{a}{\Longrightarrow} s'$ iff there exists s'', s''' such that $s \xrightarrow{\tau}^* s'' \xrightarrow{a} s''' \xrightarrow{\tau}^* s'$. Analogously, we write $s \xrightarrow{\epsilon(d)}$ iff there exists s' such that $s \xrightarrow{\epsilon(d)} s'$, and we write $s \overset{\epsilon(d)}{\Longrightarrow} s'$ iff there exists a finite transition sequence $s = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} s_n = s'$ such that for all $i \in \{1, \ldots, n\}$, $\alpha_i = \tau$ or $\alpha_i \in \mathcal{D}$, and $d = \sum\{d_i \mid \alpha_i = \epsilon(d_i)\}$. By convention, if the set $\{d_i \mid \alpha_i = \epsilon(d_i)\}$ is empty, then $\sum\{d_i \mid \alpha_i = \epsilon(d_i)\}$ is 0. With this convention, the relation $\overset{\mathbf{0}}{\Longrightarrow}$ coincides with $\xrightarrow{\tau}^*$. We say that a state s of $\mathcal{T}$ is *reachable* if there exists a finite transition sequence $s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \ldots \xrightarrow{\alpha_n} s_n$ of $\mathcal{T}$ such that for all $i \in \{1, \ldots, n\}$, $\alpha_i \in \mathcal{A}_\tau \cup \mathcal{D}$, s_0 is the initial state of $\mathcal{T}$, and $s_n = s$. By $acts(\mathcal{T})$ we denote the set $\{a \in \mathcal{A}_{\tau,\mathbf{0}} \mid \exists s, s' : s \xrightarrow{a} s'\}$. Note that $\mathbf{0}$ is an element of $acts(\mathcal{T})$.

3.1.1 Composition

To describe concurrency and synchronization between timed labelled transition systems we use synchronization functions. Assume a special "undefined"-symbol $- \notin \mathcal{A}_\tau \cup \mathcal{D}$.

Definition 3.2 (Synchronization Function) *A synchronization function f is a function from $(\mathcal{A}_{\tau,\mathbf{0}} \cup \{-\}) \times (\mathcal{A}_{\tau,\mathbf{0}} \cup \{-\})$ to $\mathcal{A}_{\tau,\mathbf{0}} \cup \{-\}$ satisfying the following conditions:*

1. *If $a, b \in \mathcal{A}_{u,\mathbf{0}}$ and $a \neq \mathbf{0}$ or $b \neq \mathbf{0}$, then $f(a,b) \in \mathcal{A}_{u,\tau} \cup \{-\}$.*
2. *If $a, b \in \mathcal{A}_{l,\mathbf{0}}$ and $a \neq \mathbf{0}$ or $b \neq \mathbf{0}$, then $f(a,b) \in \mathcal{A}_{l,\tau} \cup \{-\}$.*
3. *If $a \in \mathcal{A}_u$ and $b \in \mathcal{A}_l$ then $f(a,b) = -$.*
4. *$f(\mathbf{0},\mathbf{0}) = \mathbf{0}$*
5. *$f(\tau,\mathbf{0}) = f(\mathbf{0},\tau) = \tau$.*
6. *If $a \in \mathcal{A}_\tau$, then $f(a,\tau) = f(\tau,a) = -$.*
7. *If $a \in \mathcal{A}_{\tau,\mathbf{0}}$, then $f(a,-) = f(-,a) = f(-,-) = -$*

Condition 1 states that the synchronization of two urgent actions, or a single urgent action combined with the $\mathbf{0}$-action, must result in either an urgent action, the internal action τ, or the "undefined" symbol $-$. Condition 2 states the analogous preservation condition for lazy actions, and condition 3 prohibits joint synchronization of urgent and lazy actions. Condition 4 implies that the 0-delay property of timed labelled transition systems is preserved by parallel composition. Condition 5 states that τ-actions can occur asynchronously in a composition, and condition 6 says that τ-actions can not synchronize with any other actions than the $\mathbf{0}$-action. Finally, condition 7 prohibits any synchronization with the undefined symbol $-$.

Definition 3.3 (Composition) *Let $\mathcal{T}_i = \langle S_i, s_{0,i}, \longrightarrow_i \rangle$, $i = 1, 2$ be two timed labelled transition systems and let f be a synchronization function. The parallel composition $\mathcal{T}_1 \otimes_f \mathcal{T}_2$ is the timed labelled transition system $\langle S, s_0, \longrightarrow \rangle$ where $S = S_1 \times S_2$, $s_0 = \langle s_{1,0}, s_{2,0} \rangle$, and $\longrightarrow$ is defined as follows for all $a \in \mathcal{A}_{\tau,\mathbf{0}}$ and $d > 0$:*

- $\langle s_1, s_2 \rangle \xrightarrow{a} \langle s_1', s_2' \rangle$ *iff* $s_1 \xrightarrow{a_1}_1 s_1'$, $s_2 \xrightarrow{a_2}_2 s_2'$, *and* $f(a_1, a_2) = a$
- $\langle s_1, s_2 \rangle \xrightarrow{\epsilon(d)} \langle s_1', s_2' \rangle$ *iff* $s_1 \xrightarrow{\epsilon(d)}_1 s_1'$, $s_2 \xrightarrow{\epsilon(d)}_2 s_2'$, *and*
  $\forall t \in [0, d[,\ a_1, a_2 \in \mathcal{A}_u,\ s_1'', s_2'',$
  $\neg(s_1 \xrightarrow{\epsilon(t)}_1 s_1'' \xrightarrow{a_1}_1 \ \wedge\ s_2 \xrightarrow{\epsilon(t)}_2 s_2'' \xrightarrow{a_2}_2 \ \wedge$
  $f(a_1, a_2) \neq -)$

Note that the 0-delay property is preserved by the first of the two rules since by Definition 3.2, $f(\mathbf{0}, \mathbf{0}) = \mathbf{0}$. The definition forces the composed transition systems to synchronize on actions that correspond via f and on delays, but with the restriction that delaying is only possible when no synchronization on urgent actions is. Thus synchronizations on urgent actions must happen immediately.

3.1.2 Timed Automata

Timed labelled transition systems are described syntactically by timed automata. The timed automata model considered is the one used by the UPPAAL tool and described e.g. in [BLL+95, Kri98]. A timed automaton is a standard automaton extended with finite collections of real-valued clocks and integer-valued data variables. We consider general automata where actions are taken from the infinite set $\mathcal{A}_\tau$ and data variables are over the unbounded set of integers. However, when providing input for UPPAAL we are restricted to finite sets of actions and integers in order to obtain decidability. Assume C is a finite set of clocks and V is a finite set of data variables. We use $G(C, V)$ to stand for the set of *guards* g generated as logical combinations of constraints p on the form: $x \sim n$ or $i \sim n$ for $x \in C$, $i \in V$, $\sim \in \{<, >, =\}$, and n being a natural number. A guard g can be divided into two parts: a conjunction g_c of constraints of the form $x \sim n$ over clock variables x, and a conjunction g_v of constraints of the form $i \sim n$ over data variables i. To manipulate clock and data variables we use *reset set* of the form: $\overline{w} := \overline{e}$, which is a set of assignment operations of the form $w := e$ for w a clock or data variable and e an expression. We use R to denote the set of all possible reset operations. A reset operation on a clock variable x must be of the form $x := n$, n being a natural number, and a reset operation on a data variable i must be of the form $i := c * i + c'$, where c and c' are integer constants. For a reset set $r \in 2^R$ we let r_v be the subset of r consisting of reset operations on data variables and r_c the subset consisting of reset operations on clock variables.

Definition 3.4 (Timed Automaton) *A timed automaton A is a tuple $\langle N, l_0, C, V, E\rangle$ where N is a finite set of locations, $l_0 \in N$ is the initial location, C is a finite set of clocks, V is a finite set of data variables, and $E \subseteq N \times G(C,V) \times \mathcal{A}_\tau \times 2^R \times N$ is a set of edges.*

For a timed automaton A, we sometimes write $l \xrightarrow{g,a,r}_A l'$, or simply $l \xrightarrow{g,a,r} l'$ when A is clear from the context, to denote that $\langle l, g, a, r, l'\rangle$ is an edge of A.

Suppose that C and V are sets of clock variables and data variables, respectively. A *variable assignment* is a mapping from C to $\mathbf{R}_{\geq 0}$ and from V to integers. For a variable assignment v and a delay $d \in \mathbf{R}_{\geq 0}$, $v+d$ denotes the variable assignment such that $(v+d)(x) = v(x) + d$ for any $x \in C$, and $(v+d)(i) = v(i)$ for any integer variable i. For a reset set r, we use $r(v)$ to denote the variable assignment v' with $v'(w) = val(e,v)$ whenever $w := e \in r$ and $v'(w) = v(w)$ otherwise, where $val(e,v)$ denotes the values of e in v. Given a guard $g \in G(C,V)$ and a variable assignment v, $g(v)$ is a boolean value describing whether or not g is satisfied by v.

A state of an automaton A is a pair $\langle l, v\rangle$ where l is a node of A and v is a variable assignment. The initial state of A is $\langle l_0, v_0\rangle$, where l_0 is the initial node of A and v_0 is the initial assignment that maps all variables to 0.

Definition 3.5 (Timed Automaton Semantics) *The operational semantics of a timed automaton A is given by the timed labelled transition system, $\mathcal{T}_A = \langle S, s_0, \longrightarrow\rangle$, where S is the set of states of A, s_0 is the initial state of A, and $\longrightarrow$ is the transition relation defined as follows:*

- $\langle l, v\rangle \xrightarrow{a} \langle l', v'\rangle$ *iff* $\exists r, g.\ l \xrightarrow{g,a,r} l' \wedge g(v) \wedge v' = r(v)$
- $\langle l, v\rangle \xrightarrow{\epsilon(d)} \langle l', v'\rangle$ *iff* $l = l' \wedge v' = v + d$

where $a \in \mathcal{A}_\tau$ and $\epsilon(d) \in \mathcal{D}$.

For any timed automata A and B, we define $A \otimes_f B$ iff $\mathcal{T}_A \otimes_f \mathcal{T}_B$, and way say that B is deterministic iff $\mathcal{T}_B$ is deterministic.

3.1.3 Properties of Timed Labelled Transition Systems

When reasoning about a timed labelled transition system $\mathcal{T}$, we will specify the requirements to $\mathcal{T}$ using a specification $\mathcal{T}'$, and we will say that $\mathcal{T}$ satisfies $\mathcal{T}'$ provided there exists a *timed ready simulation relation* from $\mathcal{T}$ to $\mathcal{T}'$. The relation is a special case of a more general *parameterized* timed ready simulation, where the parameter is a certain *action relation*. This parameterized timed ready simulation will play a central role in our later conditions for property preservation. The general simulation relation will be introduced in the following section.

3.2 Timed Abstraction Theory

In this section we present our abstraction theory for timed labelled transition systems. We will denote a pair $(\mathcal{T}_1, \mathcal{T}_2)$ of timed labelled transition systems with $acts(\mathcal{T}_1) = acts(\mathcal{T}_2)$ as a *verification problem* and it is our intention to verify whether $\mathcal{T}_1$ is timed ready simulated by $\mathcal{T}_2$, to be written $\mathcal{T}_1 \preceq \mathcal{T}_2$. Given another (abstract) verification problem $(\mathcal{T}_1', \mathcal{T}_2')$ we will provide conditions for the following to hold:

$$\mathcal{T}_1' \preceq \mathcal{T}_2' \text{ implies } \mathcal{T}_1 \preceq \mathcal{T}_2$$

We begin in Section 3.2.1 by introducing a notion of parameterized timed ready simulation, where the parameter is a certain action relation. This is quite analogous to the parameterized simulations of the untimed framework. The above notion $\preceq$ of unparameterized timed ready simulation will be a special case of the parameterized relation, where the parameter is the identity relation. We will prove several nice properties of the parameterized simulation relation such as e.g. preservation under composition of timed labelled transition systems. Then in Section 3.2.2 we present the conditions for property preservation between verification problems.

3.2.1 Timed Simulations

The parameter of a parameterized timed ready simulation will be a relation on the actions of the involved transition systems. The intention is, as in the untimed framework, to allow for abstraction of a large set of concrete actions by a smaller set of abstract actions. Formally, an *action relation* is defined as follows.

Definition 3.6 *An action relation R is a relation over $\mathcal{A}^2_{\tau,\mathbf{0}}$ such that for all $(a, b) \in R$, $a, b \in \mathcal{A}_u$, $a, b \in \mathcal{A}_l$, or $a, b \in \{\mathbf{0}, \tau\}$ and $a = b$.*

We require that urgency and laziness of actions is preserved by abstraction. Note that if R is an action relation then so is R^{-1}.

Definition 3.7 *Let $\mathcal{T}_1$, $\mathcal{T}_2$ be two timed labelled transition systems and let R be an action relation. We say that R is total on $\mathcal{T}_1$ and $\mathcal{T}_2$ provided,*

- *for all $a \in acts(\mathcal{T}_1)$ there exists $b \in acts(\mathcal{T}_2)$ such that $(a, b) \in R$*
- *for all $b \in acts(\mathcal{T}_2)$ there exists $a \in acts(\mathcal{T}_1)$ such that $(a, b) \in R$*

We now define the notion of a parameterized simulation relation as follows. The definition is variant of the one in [ABL98] parameterized with an action relation.

Definition 3.8 *Let $\mathcal{T}_i = \langle S_i, s_{0,i}, \longrightarrow_i \rangle$, $i = 1, 2$ be two timed labelled transition systems. Let R be an action relation total on $\mathcal{T}_1$ and $\mathcal{T}_2$, and let Q be a relation from S_1 to S_2. We say that Q is a timed ready simulation from $\mathcal{T}_1$ to $\mathcal{T}_2$ parameterized by R provided,*

1. $(s_{0,1}, s_{0,2}) \in Q$
2. *whenever s_1, s_2 are reachable states of $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively, and $(s_1, s_2) \in Q$:*
   (a) *if $s_1 \overset{a}{\Longrightarrow}_1 s_1'$ for some $a \in \mathcal{A}$, then for all $b \in acts(\mathcal{T}_2) \cap R[a]$ there exists s_2' such that $s_2 \overset{b}{\Longrightarrow}_2 s_2'$ and $(s_1', s_2') \in Q$*
   (b) *if $s_1 \overset{\epsilon(d)}{\Longrightarrow}_1 s_1'$ for some $d \geq 0$, then $s_2 \overset{\epsilon(d)}{\Longrightarrow}_2 s_2'$ for some s_2' such that $(s_1', s_2') \in Q$*
   (c) *if $s_2 \xrightarrow{b}_2 s_2'$ for some $b \in \mathcal{A}_u$, then for all $a \in acts(\mathcal{T}_1) \cap R^{-1}[b]$ there exists s_1' such that $s_1 \xrightarrow{a}_1 s_1'$*

We write $\mathcal{T}_1 \preceq^R \mathcal{T}_2$ if there exists a timed ready simulation from $\mathcal{T}_1$ to $\mathcal{T}_2$ parameterized by R.

Conditions 2(a) and 2(b) state that concrete action and delay moves must be matched by abstract moves with related actions and identical delays, respectively. Condition 2(c) is required in order to obtain preservation of the simulation under system composition.

Consider two timed labelled transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$ with $acts(\mathcal{T}_1) = acts(\mathcal{T}_2)$. The identity relation id is trivially an action relation total on $\mathcal{T}_1$ and $\mathcal{T}_2$. We write $\mathcal{T}_1 \preceq \mathcal{T}_2$ in case $\mathcal{T}_1 \preceq^{id} \mathcal{T}_2$. Thus we have identified the notion of unparameterized timed ready simulation as a special case of the parameterized one.

In the following we state and prove several results about the parameterized timed ready simulation. Some of these results will form the basis for the preservation condition in Section 2.3.2.

The following proposition states an alternative characterization of a timed ready simulation, which is useful when proving results about it. For any $a \in \mathcal{A}_{\tau,\mathbf{0}}$, let $\hat{a} = \mathbf{0}$ if $a = \tau$ and $\hat{a} = a$ otherwise.

Proposition 3.1 *If $\mathcal{T}_i = \langle S_i, s_{0,i}, \longrightarrow_i \rangle$, $i = 1, 2$ are two timed labelled transition systems, R is an action relation total on $\mathcal{T}_1$ and $\mathcal{T}_2$, and Q is a relation from S_1 to S_2, then Q is a timed ready simulation from $\mathcal{T}_1$ to $\mathcal{T}_2$ parameterized by R iff,*

1. $(s_{0,1}, s_{0,2}) \in Q$
2. *whenever s_1, s_2 are reachable states of $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively, and $(s_1, s_2) \in Q$:*

(a) *if* $s_1 \xrightarrow{a}_1 s_1'$ *for some* $a \in \mathcal{A}_\tau$, *then for all* $b \in acts(\mathcal{T}_2) \cap R[a]$ *there exists* s_2' *such that* $s_2 \overset{\hat{b}}{\Longrightarrow}_2 s_2'$ *and* $(s_1', s_2') \in Q$

(b) *if* $s_1 \xrightarrow{\epsilon(d)}_1 s_1'$ *for some* $d > 0$, *then* $s_2 \overset{\epsilon(d)}{\Longrightarrow}_2 s_2'$ *for some* s_2' *such that* $(s_1', s_2') \in Q$

(c) *if* $s_2 \xrightarrow{b}_2 s_2'$ *for some* $b \in \mathcal{A}_u$, *then for all* $a \in acts(\mathcal{T}_1) \cap R^{-1}[b]$ *there exists* s_1' *such that* $s_1 \xrightarrow{a}_1 s_1'$

Proof. In the following $1'$ and $2'$ refers to 1 and 2 of Proposition 3.1, and 1 and 2 refers to 1 and 2 of Definition 3.8.

Assume that Q is a timed ready simulation from $\mathcal{T}_1$ to $\mathcal{T}_2$ parameterized by R. That is 1 and 2 hold. We show that $1'$ and $2'$ hold. Trivially $1'$ holds since it is identical to 1. Now, assume s_1 and s_2 are reachable states of $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively, and $(s_1, s_2) \in Q$. Suppose $s_1 \xrightarrow{a}_1 s_1'$ for some $a \in \mathcal{A}_\tau$. Then $s_1 \overset{\hat{a}}{\Longrightarrow}_1 s_1'$ and from $2(a)$ (case $a \neq \tau$) and $2(b)$ (case $a = \tau$) we have that for any $b \in acts(\mathcal{T}_2) \cap R[a]$ there exists s_2' such that $s_2 \overset{\hat{b}}{\Longrightarrow}_2 s_2'$ and $(s_1', s_2') \in Q$. Thus $2'$(a) holds. Suppose $s_1 \xrightarrow{\epsilon(d)}_1 s_1'$. Then $s_1 \overset{\epsilon(d)}{\Longrightarrow}_1 s_1'$ and from $2(b)$ there exists s_2' such that $s_2 \overset{\epsilon(d)}{\Longrightarrow}_2 s_2'$ and $(s_1', s_2') \in Q$. Thus $2'$(b) holds. Finally, $2'(c)$ holds trivially since it is identical to $2(c)$.

Now, assume that Q and R are relations satisfying $1'$ and $2'$. We show that 1 and 2 hold. Trivially 1 holds since it is identical to $1'$. Now, assume s_1 and s_2 are reachable states of $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively, and $(s_1, s_2) \in Q$. It can trivially be shown that if $s_1 \xrightarrow{\tau}_1^* s_1'$ then $s_2 \xrightarrow{\tau}_2^* s_2'$ for some s_2' such that $(s_1', s_2') \in Q$. Let $(*)$ denote this fact.

Now, suppose $s_1 \overset{a}{\Longrightarrow}_1 s_1'$ for some $a \in \mathcal{A}$. Then $s_1 \xrightarrow{\tau}_1^* s_1'' \xrightarrow{a}_1 s_1''' \xrightarrow{\tau}_1^* s_1'$ for some s_1'', s_1'''. From $(*)$ and $2'(a)$ we have that for any $b \in acts(\mathcal{T}_2) \cap R[a]$, there exists s_2' such that $s_2 \overset{b}{\Longrightarrow}_2 s_2'$ and $(s_1', s_2') \in Q$. Note that $b \neq \tau$ so $\hat{b} = b$. Thus 2(a) holds. Suppose $s_1 \overset{\epsilon(d)}{\Longrightarrow}_1 s_1'$. If $d = 0$ then $s_1 \xrightarrow{\tau}_1^* s_1'$. From $(*)$, $s_2 \xrightarrow{\tau}_2^* s_2'$ for some s_2' such that $(s_1', s_2') \in Q$. Since $\xrightarrow{\tau}^* = \overset{\mathbf{0}}{\Longrightarrow}$ this proves $2(b)$ in the case $d = 0$. If $d > 0$ then $s_1 = s_{1,0} \xrightarrow{\tau}_1^* s_{1,0}' \xrightarrow{\epsilon(d_1)}_1 s_{1,1} \xrightarrow{\tau}_1^* s_{1,1}' \xrightarrow{\epsilon(d_2)}_1 s_{1,2} \xrightarrow{\tau}_1^* \cdots \xrightarrow{\tau}_1^* s_{1,n-1}' \xrightarrow{\epsilon(d_n)}_1 s_{1,n} = s_1'$ $(n \geq 0)$ such that $\sum\{d_i \mid 1 \leq i \leq n\} = d$. From $(*)$ and $2'(b)$ we have that there exists s_2' such that $s_2 \overset{\epsilon(d)}{\Longrightarrow}_2 s_2'$ and $(s_1', s_2') \in Q$. Thus 2(b) holds. Finally, 2(c) holds trivially since it is identical to $2'$(c). ■

Let σ denote the synchronization function such that $\sigma(a, a) = a$ for all $a \in \mathcal{A}$. For any two timed labelled transition systems $\mathcal{T}_1$ and $\mathcal{T}_2$, we denote the parallel composition $\mathcal{T}_1 \otimes_\sigma \mathcal{T}_2$ *the synchronous composition* of $\mathcal{T}_1$ and $\mathcal{T}_2$. Note, that this composition is synchronous only with respect to non-τ actions. It still allows for asynchronous execution of τ-transitions from $\mathcal{T}_1$ and $\mathcal{T}_2$. This follows directly from condition 5 in Definition 3.2.

Theorem 3.1 (Idempotency of Synchronous Composition) *If $\mathcal{T}$ has no urgent actions, then $\mathcal{T} \preceq \mathcal{T} \otimes_\sigma \mathcal{T}$ and $\mathcal{T} \otimes_\sigma \mathcal{T} \preceq \mathcal{T}$.*

Proof. We first show that $\mathcal{T} \preceq \mathcal{T} \otimes_\sigma \mathcal{T}$. Let Q be the relation from states of $\mathcal{T}$ to states of $\mathcal{T} \otimes_\sigma \mathcal{T}$ such that $(s_1, \langle s_2, s_3\rangle) \in Q$ iff $s_1 = s_2 = s_3$. We show that Q is a timed ready simulation parameterized with the identity action relation. Let s_0 be the initial state of $\mathcal{T}$. Then $\langle s_0, s_0\rangle$ is the initial state of $\mathcal{T} \otimes_\sigma \mathcal{T}$ and trivially $(s_0, \langle s_0, s_0\rangle) \in Q$.

Consider reachable states s and $\langle s, s\rangle$ of $\mathcal{T}$ and $\mathcal{T} \otimes_\sigma \mathcal{T}$, respectively. Then $(s, \langle s, s\rangle) \in Q$. Suppose $s \xrightarrow{a} s'$ for some $a \in \mathcal{A}_\tau$. Consider first the case that $a = \tau$. From condition 5 in Definition 3.2 we have that $\sigma(\tau, \mathbf{0}) = \sigma(\mathbf{0}, \tau) = \tau$. Thus, $\langle s, s\rangle \xrightarrow{\tau} \langle s', s\rangle \xrightarrow{\tau} \langle s', s'\rangle$. Consider now $a \neq \tau$. Since $\sigma(a, a) = a$, we have that $\langle s, s\rangle \xrightarrow{a} \langle s', s'\rangle$. Suppose that $s \xrightarrow{\epsilon(d)} s'$. Since $\mathcal{T}$ has no urgent actions, $\langle s, s\rangle \xrightarrow{\epsilon(d)} \langle s', s'\rangle$. This concludes the first part of the proof.

We now show that $\mathcal{T} \otimes_\sigma \mathcal{T} \preceq \mathcal{T}$. Let Q be the relation from states of $\mathcal{T} \otimes_\sigma \mathcal{T}$ to states of $\mathcal{T}$ such that $(\langle s_1, s_2\rangle, s_3) \in Q$ iff $s_1 = s_3$ or $s_2 = s_3$. The initial condition holds by identical argument to the one above.

Consider reachable states $\langle s_1, s_2\rangle$ and s_3 of $\mathcal{T} \otimes_\sigma \mathcal{T}$ and $\mathcal{T}$, respectively. Assume without loss of generality that $s_1 = s_3$. Then $(\langle s_1, s_2\rangle, s_1) \in Q$. Suppose $\langle s_1, s_2\rangle \xrightarrow{a} \langle s_1', s_2'\rangle$ for some $a \in \mathcal{A}_\tau$. Then $s_1 \xrightarrow{a_1} s_1'$ and $s_2 \xrightarrow{a_2} s_2'$ for some a_1, a_2 such that $\sigma(a_1, a_2) = a$. Three cases exist. Case 1: $a_1 = a_2 = a$. Trivially, $s_1 \xrightarrow{a} s_1'$. Case 2: $a_1 = \mathbf{0}, a_2 = a = \tau$. Trivial since $s_1' = s_1$. Case 3: $a_1 = a = \tau, a_2 = \mathbf{0}$. Trivially, $s_1 \xrightarrow{\tau} s_1'$. Now, suppose that $\langle s_1, s_2\rangle \xrightarrow{\epsilon(d)} \langle s_1', s_2'\rangle$. Then $s_1 \xrightarrow{\epsilon(d)} s_1'$ and $s_2 \xrightarrow{\epsilon(d)} s_2'$. This concludes the proof. ■

An important property of parameterized timed ready simulation is preservation under composition of timed labelled transition systems. In the verification of realistic distributed systems, it is often useful to replace the individual components of the system under verification with more abstract versions before building the model of the complete system. The following compositionality theorem supports this type of hierarchical approach to verification. In the second part of this thesis we present an application of this compositionality principle in an abstraction based verification of the parameterized Fischer distributed mutual exclusion algorithm.

Definition 3.9 *Let R be an action relation and f a synchronization function. We say that R is closed with respect to f provided that, if $f(a_1, a_2) = a$, $f(b_1, b_2) = b$, and $(a, b) \in R$ then $(a_1, b_1) \in R$ and $(a_2, b_2) \in R$.*

Theorem 3.2 (Compositionality) *Let R be an action relation closed with respect to f and total on $\mathcal{T}_1 \otimes_f \mathcal{T}_3$ and $\mathcal{T}_2 \otimes_f \mathcal{T}_4$. If $\mathcal{T}_1 \preceq^R \mathcal{T}_2$ and $\mathcal{T}_3 \preceq^R \mathcal{T}_4$, and $\mathcal{T}_2, \mathcal{T}_4$ are τ-free, then $\mathcal{T}_1 \otimes_f \mathcal{T}_3 \preceq^R \mathcal{T}_2 \otimes_f \mathcal{T}_4$*

Proof. Assume that Q_1 and Q_2 are timed ready simulations from $\mathcal{T}_1$ to $\mathcal{T}_2$ and from $\mathcal{T}_3$ to $\mathcal{T}_4$, respectively, both parameterized with action relation R. Define Q to be the relation from states of $\mathcal{T}_1 \otimes_f \mathcal{T}_3$ to states of $\mathcal{T}_2 \otimes_f \mathcal{T}_4$ such that $(\langle s_1, s_3\rangle, \langle s_2, s_4\rangle) \in Q$ iff $(s_1, s_2) \in Q_1$ and $(s_3, s_4) \in Q_2$. We will show that Q is a timed ready simulation parameterized with R.

The initial state $(\langle s_{0,1}, s_{0,3}\rangle, \langle s_{0,2}, s_{0,4}\rangle) \in Q$ since $s_{0,i}$ is the initial state of $\mathcal{T}_i$ for any i, and by assumption $(s_{0,1}, s_{0,2}) \in Q_1$ and $(s_{0,3}, s_{0,4}) \in Q_2$.

Now assume that $(\langle s_1, s_3\rangle, \langle s_2, s_4\rangle) \in Q$ and that $\langle s_1, s_3\rangle$ and $\langle s_2, s_4\rangle$ are reachable states of $\mathcal{T}_1 \otimes_f \mathcal{T}_3$ and $\mathcal{T}_2 \otimes_f \mathcal{T}_4$, respectively.

Suppose $\langle s_1, s_3\rangle \xrightarrow{a} \langle s_1', s_3'\rangle$ for some $a \in \mathcal{A}_\tau$. Then $s_1 \xrightarrow{a_1} s_1'$ and $s_3 \xrightarrow{a_2} s_3'$ for some $a_1, a_2 \in \mathcal{A}_{\tau,\mathbf{0}}$ such that $f(a_1, a_2) = a$. Let b be any action in $acts(\mathcal{T}_2 \otimes_f \mathcal{T}_4) \cap R[a]$. Note that such an action exists since R is total on $\mathcal{T}_1 \otimes_f \mathcal{T}_3$ and $\mathcal{T}_2 \otimes_f \mathcal{T}_4$. By definition there exists $b_1 \in acts(\mathcal{T}_2)$, $b_2 \in acts(\mathcal{T}_4)$ such that $f(b_1, b_2) = b$ and since R is closed wrt. f, $(a_1, b_1) \in R$ and $(a_2, b_2) \in R$. Thus by simulation def. we have that $s_2 \xRightarrow{\hat{b_1}} s_2'$ and $s_4 \xRightarrow{\hat{b_2}} s_4'$ for some s_2', s_4' such that $(s_1', s_2') \in Q_1$ and $(s_3', s_4') \in Q_2$. Thus $\langle s_2, s_4\rangle \xRightarrow{\hat{b}} \langle s_2', s_4'\rangle$ and $(\langle s_1', s_3'\rangle, \langle s_2', s_4'\rangle) \in Q$.

Suppose $\langle s_1, s_3\rangle \xrightarrow{\epsilon(d)} \langle s_1', s_3'\rangle$. Then $s_1 \xrightarrow{\epsilon(d)} s_1'$, $s_3 \xrightarrow{\epsilon(d)} s_3'$, and for all $t \in [0, d[$, $a_1, a_2 \in \mathcal{A}_u$, and s_1'', s_3'': $\neg(s_1 \xrightarrow{\epsilon(t)} s_1'' \xrightarrow{a_1} \wedge\ s_3 \xrightarrow{\epsilon(t)} s_3'' \xrightarrow{a_2} \wedge\ f(a_1, a_2) \neq -)$ $(*)$. Since $(s_1, s_2) \in Q_1$, $(s_3, s_4) \in Q_2$, and $\mathcal{T}_2$ and $\mathcal{T}_4$ are τ-free, we have that $s_2 \xrightarrow{\epsilon(d)} s_2'$ for some s_2' such that $(s_1', s_2') \in Q_1$ and $s_4 \xrightarrow{\epsilon(d)} s_4'$ for some s_4' such that $(s_3', s_4') \in Q_2$. Now, assume for the sake of contradiction that there exists $t \in [0, d[$, $b_1, b_2 \in \mathcal{A}_u$, and s_2'', s_4'' such that $s_2 \xrightarrow{\epsilon(t)} s_2'' \xrightarrow{b_1}$, $s_4 \xrightarrow{\epsilon(t)} s_4'' \xrightarrow{b_2}$, and $f(b_1, b_2) \neq -$. From time additivity there exists s_1'', s_3'' such that $s_1 \xrightarrow{\epsilon(t)} s_1''$ and $s_3 \xrightarrow{\epsilon(t)} s_3''$, and due to time determinism $(s_1'', s_2'') \in Q_1$ and $(s_3'', s_4'') \in Q_2$. Suppose $f(b_1, b_2) = b$ and let a be any action in $acts(\mathcal{T}_1 \otimes_f \mathcal{T}_3) \cap R^{-1}[b]$. Note that such an action exists since R is total on $\mathcal{T}_1 \otimes_f \mathcal{T}_3$ and $\mathcal{T}_2 \otimes_f \mathcal{T}_4$. By def. there exists $a_1 \in acts(\mathcal{T}_1)$ and $a_2 \in acts(\mathcal{T}_3)$ such that $f(a_1, a_2) = a$, and since R is closed wrt. f, $(a_1, b_1) \in R$, $(a_2, b_2) \in R$, and $a_1, a_2 \in \mathcal{A}_u$. Thus from simulation def. $s_1'' \xrightarrow{a_1}$ and $s_3'' \xrightarrow{a_2}$ contradicting $(*)$.

Suppose $\langle s_2, s_4\rangle \xrightarrow{b} \langle s_2', s_4'\rangle$ for some $b \in \mathcal{A}_u$. Then $s_2 \xrightarrow{b_1} s_2'$ and $s_4 \xrightarrow{b_2} s_4'$ for some $b_1, b_2 \in \mathcal{A}_u \cup \{\mathbf{0}\}$ such that $f(b_1, b_2) = b$. Let a be any action in $acts(\mathcal{T}_1 \otimes_f \mathcal{T}_3) \cap R^{-1}[b]$. By def. there exists $a_1 \in acts(\mathcal{T}_1)$, $a_2 \in acts(\mathcal{T}_3)$ such that $f(a_1, a_2) = a$, and since R is closed wrt. f, $(a_1, b_1) \in R$ and $(a_2, b_2) \in R$. Thus $s_1 \xrightarrow{a_1} s_1'$ and $s_3 \xrightarrow{a_2} s_3'$ for some s_1', s_3' and hence $\langle s_1, s_3\rangle \xrightarrow{a} \langle s_1', s_3'\rangle$ and $(\langle s_1', s_3'\rangle, \langle s_2', s_4'\rangle) \in Q$. ■

Theorem 3.3 (Transitivity) *Suppose* $\mathcal{T}_1 \preceq^{R_1} \mathcal{T}_2$ *and* $\mathcal{T}_2 \preceq^{R_2} \mathcal{T}_3$. *Then* $\mathcal{T}_1 \preceq^{R_1 R_2} \mathcal{T}_3$

Proof. Note that by definition R_1R_2 is total on $\mathcal{T}_1$ and $\mathcal{T}_2$. Assume that Q_1 is a timed ready simulation from $\mathcal{T}_1$ to $\mathcal{T}_2$ parameterized by R_1 and Q_2 is a timed ready simulation from $\mathcal{T}_2$ to $\mathcal{T}_3$ parameterized by R_2. Let $Q = Q_1Q_2$. We show that Q is a timed ready simulation from $\mathcal{T}_1$ to $\mathcal{T}_3$ parameterized by R_1R_2. We consider each of the conditions in Definition 3.8 seperately.

By assumption $(s_{0,1}, s_{0,2}) \in Q_1$ and $(s_{0,2}, s_{0,3}) \in Q_2$ for $s_{0,1}, s_{0,2}$ the initial states of $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. Thus $(s_{0,1}, s_{0,3}) \in Q$ and condition 1 is satisfied. Now, assume that $(s_1, s_2) \in Q_1$ and $(s_2, s_3) \in Q_2$ and s_1, s_2 and s_3 are reachable states of $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$, respectively.

Suppose $s_1 \overset{a}{\Longrightarrow} s_1'$ for some s_1' and $a \in \mathcal{A}$. Consider any $b \in \mathit{acts}(\mathcal{T}_3) \cap R_1R_2[a]$. Then for some $c \in \mathit{acts}(\mathcal{T}_2)$, $(a, c) \in R_1$ and $(c, b) \in R_2$. Since $(s_1, s_2) \in Q_1$, $s_2 \overset{c}{\Longrightarrow} s_2'$ for some s_2' such that $(s_1', s_2') \in Q_1$, and since $(s_2, s_3) \in Q_2$, $s_3 \overset{b}{\Longrightarrow} s_3'$ for some s_3' such that $(s_2', s_3') \in Q_2$. Hence $(s_1', s_3') \in Q$ and 2(a) holds.

Suppose $s_1 \overset{\epsilon(d)}{\Longrightarrow} s_1'$ for some s_1' and $\epsilon(d) \in \mathcal{D}$. Since $(s_1, s_2) \in Q_1$, $s_2 \overset{\epsilon(d)}{\Longrightarrow} s_2'$ for some s_2' such that $(s_1', s_2') \in Q_1$, and since $(s_2, s_3) \in Q_2$, $s_3 \overset{\epsilon(d)}{\Longrightarrow} s_3'$ for some s_3' such that $(s_2', s_3') \in Q_2$. Hence $(s_1', s_3') \in Q$ and 2(b) holds.

Finally, suppose $s_3 \xrightarrow{b} s_3'$ for some s_3' and $b \in \mathcal{A}_u$. Consider any $a \in \mathit{acts}(\mathcal{T}_1) \cap R_1R_2^{-1}[b]$. Then for some $c \in \mathcal{A}_u$, $(a, c) \in R_1$ and $(c, b) \in R_2$. Since $(s_2, s_3) \in R_2$, $s_2 \xrightarrow{c} s_2'$ for some s_2', and since $(s_1, s_2) \in R_1$, $s_1 \xrightarrow{a} s_1'$ for some s_1'. Hence 2(c) holds. ■

Theorem 3.4 (Subset Closure) *If $\mathcal{T}_1 \preceq^R \mathcal{T}_2$ then for all action relations $R' \subseteq R$ such that R' is total on $\mathcal{T}_1$ and $\mathcal{T}_2$, $\mathcal{T}_1 \preceq^{R'} \mathcal{T}_2$*

Proof. Assume Q is a timed ready simulation from $\mathcal{T}_1$ to $\mathcal{T}_2$ parameterized by R. Let R' be any action relation from $\mathcal{T}_1$ to $\mathcal{T}_2$ such that $R' \subseteq R$ and R' is total on $\mathcal{T}_1$ and $\mathcal{T}_2$. We show that Q is a timed ready simulation from $\mathcal{T}_1$ to $\mathcal{T}_2$ parameterized by R'. We consider each of the conditions in Definition 3.8. Condition 1 holds trivially. Asumme that s_1 and s_2 are reachable states of $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively, and $(s_1, s_2) \in Q$.

Suppose $s_1 \overset{a}{\Longrightarrow} s_1'$ for some s_1' and $a \in \mathcal{A}$. Consider any $b \in \mathit{acts}(\mathcal{T}_2) \cap R'[a]$. Then $b \in \mathit{acts}(\mathcal{T}_2) \cap R[a]$ and since $(s_1, s_2) \in Q$, $s_2 \overset{b}{\Longrightarrow} s_2'$ for some s_2' such that $(s_1', s_2') \in Q$. Hence 2(a) holds.

Condition 2(b) holds directly by Definition 3.8 since $(s_1, s_2) \in Q$.

Suppose $s_2 \xrightarrow{b} s_2'$ for some s_2' and $b \in \mathcal{A}_u$. Consider any $a \in \mathit{acts}(\mathcal{T}_1) \cap (R')^{-1}[b]$. Then $a \in \mathit{acts}(\mathcal{T}_1) \cap R^{-1}[b]$ and since $(s_1, s_2) \in Q$, $s_1 \xrightarrow{a} s_1'$ for some s_1', so 2(c) holds. ■

We now proceed to provide sufficient conditions for an abstract verification problem to be property preserving with respect to a concrete problem.

3.2.2 Preservation Conditions

For any two verification problems $(\mathcal{T}_1, \mathcal{T}_2)$ and $(\mathcal{T}'_1, \mathcal{T}'_2)$, we will state and prove a sufficient condition for the following to hold:

$$\mathcal{T}'_1 \preceq \mathcal{T}'_2 \text{ implies } \mathcal{T}_1 \preceq \mathcal{T}_2$$

The condition is based the existence of parametarized timed simulation relations from $\mathcal{T}_1$ to $\mathcal{T}'_1$ and from $\mathcal{T}'_2$ to $\mathcal{T}_2$, respectively.

Theorem 3.5 *Let $(\mathcal{T}_1, \mathcal{T}_2)$ and $(\mathcal{T}'_1, \mathcal{T}'_2)$ be two verification problems. Also, let R be an action relation total on $\mathcal{T}_1$ and $\mathcal{T}'_1$ (R^{-1} is an action relation total on $\mathcal{T}'_2$ and $\mathcal{T}_2$). If,*

$$\mathcal{T}_1 \preceq^R \mathcal{T}'_1 \quad \textit{and} \quad \mathcal{T}'_2 \preceq^{R^{-1}} \mathcal{T}_2$$

then

$$\mathcal{T}'_1 \preceq \mathcal{T}'_2 \quad \textit{implies} \quad \mathcal{T}_1 \preceq \mathcal{T}_2$$

Proof. Assume that $\mathcal{T}_1 \preceq^R \mathcal{T}'_1$, $\mathcal{T}'_2 \preceq^{R^{-1}} \mathcal{T}'_1$, and $\mathcal{T}'_1 \preceq \mathcal{T}'_2$. By transitivity, Theorem 3.3, $\mathcal{T}_1 \preceq^{RR^{-1}} \mathcal{T}_2$ and since $id \subseteq RR^{-1}$ and since id is an action relation total on $\mathcal{T}_1$ and $\mathcal{T}_2$, subset closure, Theorem 3.4, implies that $\mathcal{T}_1 \preceq \mathcal{T}_2$. ■

3.3 Constructing Abstract Timed Automata

In this section we consider the problem of constructing a special and useful abstract timed automaton directly from the description of a concrete automaton together with a set of abstract actions and an abstract data variable domain. The abstraction is constructed such that it timed simulates, up to the action relation, the concrete automaton, and the concrete automaton timed simulates, up to the inverse action relation, the abstract automaton. The construction here will be used in the abstraction proof of Fischer's mutual exclusion algorithm presented in Chapter 6 of the second part of this thesis.

Let $A = \langle N, l_0, C, V, E \rangle$ be a timed automaton and let $h : \mathbf{Z} \mapsto \mathbf{Z}$ be a function. For any guard $g \in G(C, V)$ we denote by g_h the guard obtained from g by replacing all occurrences of natural numbers n in g_v by $h(n)$. Similarly, for any reset set $r \in 2^R$ we denote by r_h the set obtained from r by replacing all occurrences of integers c in reset operations of r_v by $h(c)$. For any variable assignment v we let $h(v)$ denote the assignment such that $h(v)(i) = h(v(i))$ for any data variable i and $h(v)(x) = v(x)$ for any clock variable x. A guard g is said to be *preserved by h* iff for any assignment v,

$$g(v) \text{ iff } g_h(h(v))$$

A reset set r is said to be *preserved by* h iff for any assignment v,

$$h(r(v)) = r_h(h(v))$$

An *abstract data domain* for A is a pair (Δ, h), where $\Delta \subseteq \mathbf{Z}$ such that $0 \in \Delta$, and where h is a function from $\mathbf{Z}$ to Δ such that h is preserving any guard and any reset operation of A and $h(0) = 0$. An *abstract action domain* for A is a pair (Σ, R), where R is an action relation total on $acts(\mathcal{T}_A)$ and Σ. An *abstract domain* for A is a pair consisting of a data abstraction for A an action abstraction for A. Given an automaton A and and abstract domain F for A we now define an abstract automaton A_F which we can show will timed ready simulate automaton A. The abstract automaton is obtained simply by replacing transition labels of A (guards, actions, resets) with their abstracted counterparts.

Definition 3.10 *Let A be a timed automaton and $F = ((\Delta, h), (\Sigma, R))$ an abstract domain for A. The abstraction of A with respect to F is the timed automaton A_F over nodes and variables of A and with transition relation such that:*

$$l \xrightarrow{g,b,r}_{A_F} l' \quad \textit{iff} \quad l \xrightarrow{g',a,r'}_{A} l'$$

for some g', r', a such that $g'_h = g, r'_h = r$, and $(a, b) \in R$.

We say that A is *closed* under R iff $l \xrightarrow{g,a,r}_A l'$ and $(a, b) \in R$ implies that for all a' such that $(a', b) \in R$, $l \xrightarrow{g,a',r}_A l'$. If we restrict a, a' to be urgent actions, we say that A is *urgent closed* under R. For timed automata A,B we define $A \preceq^R B$ iff $\mathcal{T}_A \preceq^R \mathcal{T}_B$.

Theorem 3.6 *If A is a timed automaton and F is an abstract domain for A with action relation R, and if A is urgent closed under R. Then $A \preceq^R A_F$.*

Proof. Let $F = ((\Delta, h), (\Sigma, R))$. Let Q be the relation from states of $\mathcal{T}_A$ to states of $\mathcal{T}_{A_F}$ such that $(\langle l, v\rangle, \langle l', v'\rangle) \in Q$ iff $l' = l$ and $v' = h(v)$. We show that Q is a timed ready simulation from $\mathcal{T}_A$ to $\mathcal{T}_{A_F}$ parameterized by R. We consider each of the conditions in Definition 3.8.

Let $\langle l_0, v_0\rangle$ be the initial state of $\mathcal{T}_A$. Directly from the definition of A_F we have that $\langle l_0, v_0\rangle$ is the initial state of $\mathcal{T}_{A_F}$ and since $h(0) = 0$ we have that $h(v_0) = v_0$ and hence $(\langle l_0, v_0\rangle, \langle l_0, v_0\rangle) \in Q$. Now, assume that $(\langle l, v\rangle, \langle l, h(v)\rangle) \in Q$ and that $\langle l, v\rangle$ and $\langle l, h(v)\rangle$ are reachable states of $\mathcal{T}_A$ and $\mathcal{T}_{A_F}$, respectively.

Suppose that $\langle l, v\rangle \xrightarrow{a} \langle l', v'\rangle$ for some $a \in \mathcal{A}_\tau$. Then there exists g, r such that $l \xrightarrow{g,a,r}_A l'$, $g(v)$, and $r(v) = v'$. From definition of A_F we have that for any b such that $(a, b) \in R$, $l \xrightarrow{g_h,b,r_h}_{A_F} l'$. Thus, since $g(v)$ implies $g_h(h(v))$ and $r_h(h(v)) = h(r(v)) = h(v')$, we have that $\langle l, h(v)\rangle \xrightarrow{b} \langle l', h(v')\rangle$ and $(\langle l', v'\rangle, \langle l', h(v')\rangle) \in Q$.

Suppose that $\langle l, v\rangle \xrightarrow{\epsilon(d)} \langle l', v'\rangle$ for some $\epsilon(d) \in \mathcal{D}$. Then $l' = l$ and $v' = v + d$. Since $h(v) + d = h(v + d) = h(v')$ we have directly from the operational semantics of a timed automaton that $\langle l, h(v)\rangle \xrightarrow{\epsilon(d)} \langle l, h(v')\rangle$ and $(\langle l', v'\rangle, \langle l', h(v')\rangle) \in Q$.

Suppose that $\langle l, h(v)\rangle \xrightarrow{b} \langle l', v'\rangle$ and $b \in \mathcal{A}_u$. Then there exists g, r such that $l \xrightarrow{g,b,r}_{A_F} l'$, $g(h(v))$, and $r(h(v)) = v'$. From the definition of A_F and since A is urgent closed under R there exists g', r' such that $g'_h = g$ and $r'_h = r$ and for any $a \in \mathcal{A}_u$ such that $(a, b) \in R$, $l \xrightarrow{g',a,r'}_A l'$. Now, since $g(h(v))$ implies $g'(v)$ and $r(v') = r(h(v)) = h(r'(v))$ we have that $\langle l, v\rangle \xrightarrow{a} \langle l', r'(v)\rangle$ and $(\langle l', r'(v)\rangle, \langle l', h(r'(v))\rangle \in Q$. This ends the proof that $A \preceq^R A_F$. ■

Theorem 3.7 *If A is a timed automaton and F is an abstract domain for A with action relation R, and if A closed under R. Then $A_F \preceq^{R^{-1}} A$.*

Proof. Let $F = ((\Delta, h), (\Sigma, R))$. Let Q be the relation from states of $\mathcal{T}_{A_F}$ to states of $\mathcal{T}_A$ such that $(\langle l, v\rangle, \langle l', v'\rangle) \in Q$ iff $l' = l$ and $v = h(v')$. We show that Q is a timed ready simulation from $\mathcal{T}_A$ to $\mathcal{T}_{A_F}$ parameterized by R. We consider each of the conditions in Definition 3.8.

Let $\langle l_0, v_0\rangle$ be the initial state of $\mathcal{T}_{A_F}$. Directly from the definition of A_F we have that $\langle l_0, v_0\rangle$ is the initial state of $\mathcal{T}_A$ and since $h(0) = 0$ we have that $h(v_0) = v_0$ and hence $(\langle l_0, v_0\rangle, \langle l_0, v_0\rangle) \in Q$. Now, assume that $(\langle l, h(v)\rangle, \langle l, v\rangle) \in Q$ and that $\langle l, h(v)\rangle$ and $\langle l, v\rangle$ are reachable states of $\mathcal{T}_{A_F}$ and $\mathcal{T}_A$, respectively.

Suppose that $\langle l, h(v)\rangle \xrightarrow{a} \langle l', v'\rangle$ for some $a \in \mathcal{A}_\tau$. Then there exists g, r such that $l \xrightarrow{g,a,r}_{A_F} l'$, $g(h(v))$, and $r(h(v)) = v'$. From definition of A_F and since A is closed under R there exists g', r' such that $g'_h = g$ and $r'_h = r$ and for any b such that $(a, b) \in R^{-1}$, $l \xrightarrow{g',b,r'}_A l'$. Thus, since $g(h(v))$ implies $g'(v)$ and $v' = r(h(v)) = h(r'(v))$, we have that $\langle l, v\rangle \xrightarrow{b} \langle l', r'(v)\rangle$ and $(\langle l', h(r'(v))\rangle, \langle l', r'(v)\rangle) \in Q$.

Suppose that $\langle l, h(v)\rangle \xrightarrow{\epsilon(d)} \langle l', v'\rangle$ for some $\epsilon(d) \in \mathcal{D}$. Then $l' = l$ and $v' = h(v) + d$. Since $h(v) + d = h(v + d)$ we have directly from the operational semantics of a timed automaton that $\langle l, v\rangle \xrightarrow{\epsilon(d)} \langle l, v + d\rangle$ and $(\langle l', h(v + d)\rangle, \langle l', v + d\rangle) \in Q$.

Suppose that $\langle l, v\rangle \xrightarrow{b} \langle l', v'\rangle$ and $b \in \mathcal{A}_u$. Then there exists g, r such that $l \xrightarrow{g,b,r}_A l'$, $g(v)$, and $r(v) = v'$. Directly from the definition of A_F, for any $a \in \mathcal{A}_u$ such that $(a, b) \in R^{-1}$, $l \xrightarrow{g_h,a,r_h}_{A_F} l'$. Now, since $g(v)$ implies $g_h(h(v))$ and $h(v') = h(r(v)) = r_h(h(v))$ we have that $\langle l, h(v)\rangle \xrightarrow{a}_{\langle} l', h(v')\rangle$ and $(\langle l', h(v')\rangle, \langle l', v'\rangle \in Q$. This ends the proof that $A_F \preceq^{R^{-1}} A$. ■

The following corollary follows directly from Theorems 3.5, 3.6, and 3.7.

Corollary 3.8 *Let (A, B) be a verification problem and F an abstract domain for A and B with action relation R.*

1. *If A is urgent closed and B is closed, under R, then*

$$A_F \preceq B_F \text{ implies } A \preceq B$$

2. *If A and B are closed under R, then*

$$A_F \preceq B_F \text{ iff } A \preceq B$$

3.4 Test Automata for Timed Ready Simulation

We end this chapter by considering the problem of *testing* for the existence of a timed ready simulation between two timed automata. For any deterministic and τ-free timed automaton B we will define the notion of a *test automaton* T_B for B. Test automaton T_B will have the property that it can be used to determine whether B timed ready simulates any timed automaton A ($A \preceq B$), by performing reachability analysis in the composition of A and T_B. In the following we will assume that $\mathcal{A}$ is equipped with a mapping $\bar{\cdot} : \mathcal{A} \mapsto \mathcal{A}$ such that $\overline{\overline{a}} = a$ for every $a \in \mathcal{A}$. Let $A = \langle N, l_0, C, V, E \rangle$ be a timed automaton. We let $G_{E,l,a}$ denote the set of guards $\{g \mid \exists r, l'.\ l \xrightarrow{g,a,r} l' \in E\}$. We let $U_{E,l}$ denote the set $\{a \in \mathcal{A}_u \mid \exists g, r, l'.\ l \xrightarrow{g,a,r} l' \in E\}$. For any node $l \in N$ and urgent action $a \in U_{E,l}$ we assume the existence of distinguished nodes l_a^1 and l_a^2 such that $l_a^1, l_a^2 \notin N$. We let $\mathcal{N}_E$ denote the set $\bigcup_{l \in N} \bigcup_{a \in U_{E,l}} \{l_a^1, l_a^2\}$. For any $l \in N$, we also assume the existence of a distinguished nodes $l_r, l_\tau \notin N \cup \mathcal{N}_E$, $l_r \neq l_\tau$. We let $\mathcal{N}_{r,\tau}$ denote the set $\bigcup_{l \in N} \{l_r, l_\tau\}$. Finally, for any urgent action $a \in \bigcup_{l \in N} U_{E,l}$ we assume the existence of a distinguished clock $x_a \notin C$. We let $\mathcal{C}_E$ denote the set $\bigcup_{l \in N} \bigcup_{a \in U_{E,l}} \{x_a\}$.

Definition 3.11 *Let $A = \langle N, l_0, C, V, E \rangle$ be a τ-free deterministic timed automaton. The test automaton for A is the timed automaton $T_A = \langle N^T, l_0^T, C^T, V^T, E^T \rangle$ where $N^T = N \cup \mathcal{N}_E \cup \mathcal{N}_{r,\tau}$, $l_0^T = \overline{l_0}$, $C^T = C \cup \mathcal{C}_E$, $V^T = V$, and where $l_T \xrightarrow{g_T, a_T, r_T} l'_T \in E^T$ iff one of the following holds:*

1. $l_T = l \ \wedge\ l'_T = l_\tau \ \wedge\ g_T = true \ \wedge\ a_T = \tau \ \wedge\ r_T = \emptyset$
2. $l_T = l_\tau \ \wedge\ l'_T = l' \ \wedge\ g_T = g \ \wedge\ a_T = \overline{a} \ \wedge\ r_T = r \ \wedge\ l \xrightarrow{g,a,r} l' \in E$
3. $l_T = l_\tau \ \wedge\ l'_T = l_r \ \wedge\ g_T = \bigwedge_{g \in G_{E,l,a}} \neg g \ \wedge a_T = \overline{a} \ \wedge\ r_T = \emptyset$
4. $l_T = l \ \wedge\ l'_T = l_a^1 \ \wedge\ g_T = \bigvee_{g \in G_{E,l,a}} g \ \wedge\ a_T = \tau \ \wedge\ r_T = \{x_a := 0\}$

5. $l_T = l_a^1 \;\wedge\; l'_T = l_a^2 \;\wedge\; g_T = true \;\wedge\; a_T = \overline{a} \;\wedge\; r_T = \emptyset$

6. $l_T = l_a^1 \;\wedge\; l'_T = l_r \;\wedge\; g_T = \{x_a > 0\} \;\wedge\; a_T = \tau \;\wedge r_T = \emptyset$

We will write a variable assignment over $C \cup V \cup \mathcal{C}_E$ as a composition $v \oplus w$ of variable assignments v and w over $C \cup V$ and $\mathcal{C}_E$, respectively, such that $(v \oplus w)(x) = v(x)$ if $x \in C \cup V$ and $(v \oplus w)(x) = w(x)$ if $x \in \mathcal{C}_E$. In Figure 3.1 we have illustrated the general test automaton construction of Definition 3.11. Note that we assume that $l \xrightarrow{false,a,\emptyset}_A l'$ for any l' iff $l \not\xrightarrow{a}_A$.

We now define the notion of testing. First, let s denote the synchronization function defined as follows. For any $a \in \mathcal{A}$, $s(a, \overline{a}) = \tau$, and $s(a, \mathbf{0}) = s(\mathbf{0}, \overline{a}) = -$. In the following we will write $\otimes$ to denote the composition operator $\otimes_s$. This operator will be the one used to combine a test automaton (its semantics) with a given system to be tested.

Definition 3.12 *Let $\mathcal{T}$ be a TLTS and T_A the test automaton for some timed automaton A.*

- *A node l of T_A is reachable from a state $\langle s, t\rangle$ of $\mathcal{T} \otimes \mathcal{T}_{T_A}$ iff there exists a state $\langle s', t'\rangle$ of $\mathcal{T} \otimes \mathcal{T}_{T_A}$ such that $\langle s', t'\rangle$ is reachable from $\langle s, t\rangle$ and $t' = \langle l, v\rangle$ for some variable assignment v.*
- *We say that $\mathcal{T}$ fails the A-test iff a reject node l_r of T_A is reachable from the initial state of $\mathcal{T} \otimes \mathcal{T}_{T_A}$. Otherwise, we say that $\mathcal{T}$ passes the A-test.*

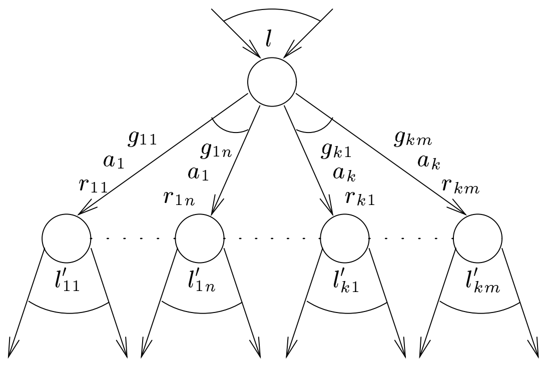
If B and A are timed automata, we say that B passes the A-test iff $\mathcal{T}_B$ passes the A-test.

Theorem 3.9 *If $\mathcal{T}$ is a TLTS and A is a τ-free and deterministic timed automata, then $\mathcal{T}$ passes the A-test iff $\mathcal{T} \preceq \mathcal{T}_A$.*

Proof. We will assume that E denotes the set of edges of automaton A.

Assume that $\mathcal{T}$ passes the A-test. Define R to be the relation from states of $\mathcal{T}$ to states of $\mathcal{T}_A$ such that $(s, \langle l, v\rangle) \in R$ iff there exists w such that $\langle s, \langle l, v \oplus w\rangle\rangle$ is reachable from the initial state of $\mathcal{T} \otimes \mathcal{T}_{T_A}$. We will show that R is a timed ready simulation relation from $\mathcal{T}$ to $\mathcal{T}_A$. We show that R satisfies the conditions of Proposition 3.1. Consider first condition 1. Let s_0 and $\langle l_0, v_0\rangle$ be the initial states of $\mathcal{T}$ and $\mathcal{T}_A$, respectively. From the definition of $\mathcal{T}_{T_A}$, $\langle l_0, v_0 \oplus w_0\rangle$ is the initial state of $\mathcal{T}_{T_A}$, where w_0 is assigning the value 0 to all elements of $\mathcal{C}_E$. Thus, from the definition of $\mathcal{T} \otimes \mathcal{T}_{T_A}$, $\langle s_0, \langle l_0, v_0 \oplus w_0\rangle\rangle$ is the initial state of $\mathcal{T} \otimes \mathcal{T}_{T_A}$ and hence $(s_0, \langle l_0, v_0\rangle) \in R$.

Now assume that $(s, \langle l, v\rangle) \in R$ and s and $\langle l, v\rangle$ are reachable states of $\mathcal{T}$ and $\mathcal{T}_A$, respectively. Then there exists w such that $\langle s, \langle l, v \oplus w\rangle\rangle$ is reachable in $\mathcal{T} \otimes \mathcal{T}_{T_A}$. Fix w. We consider each of the conditions 2(a)-2(d).



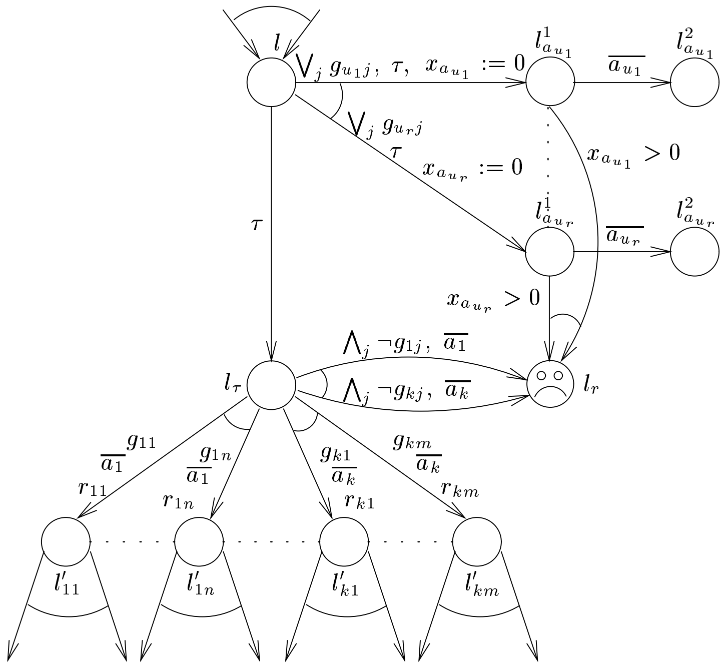

l
$\bigvee_j g_{u_1 j},\ \tau,\ x_{a_{u_1}} := 0$
$l^1_{a_{u_1}}$
$\overline{a_{u_1}}$
$l^2_{a_{u_1}}$
$\bigvee_j g_{u_r j}$ τ $x_{a_{u_r}} := 0$
$x_{a_{u_1}} > 0$
τ
$l^1_{a_{u_r}}$
$\overline{a_{u_r}}$
$l^2_{a_{u_r}}$
$x_{a_{u_r}} > 0$
$\bigwedge_j \neg g_{1j},\ \overline{a_1}$
$\bigwedge_j \neg g_{kj},\ \overline{a_k}$
l_τ
l_r
g_{11} $\overline{a_1}$ r_{11}
g_{1n} $\overline{a_1}$ r_{1n}
g_{k1} $\overline{a_k}$ r_{k1}
g_{km} $\overline{a_k}$ r_{km}
l'_{11}
l'_{1n}
l'_{k1}
l'_{km}

Figure 3.1: Timed Automaton A and Test Automaton T_A

Suppose $s \xrightarrow{a} s'$ for some $a \in \mathcal{A}$. From condition 1 in the definition of T_A, $\langle l, v \oplus w\rangle \xrightarrow{\tau} \langle l_\tau, v \oplus w\rangle$, and from condition 3, $\langle l_\tau, v \oplus w\rangle \xrightarrow{\overline{a}} \langle l_r, v \oplus w\rangle$ if $g_T(v \oplus w)$ where $g_T = \bigwedge_{g \in G_{E,l,a}} \neg g$. Thus, since $\mathcal{T}$ passes the A-test it must be that $\neg g_T(v \oplus w)$ i.e. $\bigvee_{g \in G_{E,l,a}} g(v \oplus w)$. This implies $\bigvee_{g \in G_{E,l,a}} g(v)$. Fix g. Then, there exists r, l' such that $l \xrightarrow{g,a,r} l' \in E$ and $g(v)$ implies that $\langle l, v\rangle \xrightarrow{a} \langle l', v'\rangle$, $v' = r(v)$. From condition 2 in the definition of T_A, $\langle l_\tau, v \oplus w\rangle \xrightarrow{\overline{a}} \langle l', v' \oplus w\rangle$ and thus $\langle s', \langle l', v' \oplus w\rangle\rangle$ is reachable in $\mathcal{T} \otimes \mathcal{T}_{T_A}$ and so $(s', \langle l', v'\rangle) \in R$.

Suppose $s \xrightarrow{\tau} s'$. Then $\langle s, \langle l, v \oplus w\rangle\rangle \xrightarrow{\tau} \langle s', \langle l, v \oplus w\rangle\rangle$ and $\langle s', \langle l, v \oplus w\rangle\rangle$ is reachable in $\mathcal{T} \otimes \mathcal{T}_{T_A}$. Hence $(s', \langle l, v\rangle) \in R$.

Suppose $s \xrightarrow{\epsilon(d)} s'$. We know that $\langle l, v\rangle \xrightarrow{\epsilon(d)} \langle l, v'\rangle$ where $v' = v + d$, and $\langle l, v \oplus w\rangle \xrightarrow{\epsilon(d)} \langle l, v' \oplus w'\rangle$ where $w' = w + d$. Now, assume for the sake of contradiction that $\langle s, \langle l, v \oplus w\rangle\rangle \not\xrightarrow{\epsilon(d)}$. Then from Definition 3.3, there exists $t \in [0, d[$, $a \in \mathcal{A}_u$, such that $\langle l, v \oplus w\rangle \xrightarrow{\epsilon(t)} \langle l, (v \oplus w) + t\rangle \xrightarrow{a}$. However, this contradicts conditions 1 and 4 in the definition of T_A according to which, only τ-actions are possible from states with node component l. Thus, $\langle s, \langle l, v \oplus w\rangle\rangle \xrightarrow{\epsilon(d)} \langle s', \langle l, v' \oplus w'\rangle\rangle$ and so $(s', \langle l, v'\rangle) \in R$.

Suppose $\langle l, v\rangle \xrightarrow{a} \langle l', v'\rangle$ for some $a \in \mathcal{A}_u$. Then there exists g, r such that $l \xrightarrow{g,a,r} l' \in E$ and $g(v)$. Since no variables from $\mathcal{C}_E$ occurs in g, $g(v)$ implies $g(v \oplus w)$. Since $g \in G_{E,l,a}$ we have from condition 4 in the definition of T_A that $\langle l, v \oplus w\rangle \xrightarrow{\tau} \langle l_a^1, v \oplus w'\rangle$ where $w' = w\{x_a := 0\}$. Hence, $\langle s, \langle l, v \oplus w\rangle\rangle \xrightarrow{\tau} \langle s, \langle l_a^1, v \oplus w'\rangle\rangle$. Now, assume for the sake of contradiction that $\langle s, \langle l_a^1, v \oplus w'\rangle\rangle \xrightarrow{\epsilon(d)}$ for some $d > 0$. Then $\langle l_a^1, v \oplus w'\rangle \xrightarrow{\epsilon(d)} \langle l_a^1, (v + d) \oplus (w' + d)\rangle$ and since $(w' + d)(x_a) > 0$ we get from condition 6 in the definition of T_A that $\langle l_a^1, (v+d) \oplus (w'+d)\rangle \xrightarrow{\tau} \langle l_r, (v+d) \oplus (w'+d)\rangle$. Hence $\langle s, \langle l_r, (v + d) \oplus (w' + d)\rangle\rangle$ is reachable in $\mathcal{T} \otimes \mathcal{T}_{T_A}$ which contradicts the fact that $\mathcal{T}$ passes the A-test. Therefore, $\langle s, \langle l_a^1, v \oplus w'\rangle\rangle \not\xrightarrow{\epsilon(d)}$ for any $d > 0$. Hence, there exists $a_1, a_2 \in \mathcal{A}_u$ such that $s \xrightarrow{a_1}$ and $\langle l_a^1, v \oplus w'\rangle \xrightarrow{a_2}$ and $a_1 = \overline{\overline{a_2}}$. From part 5 in the definition of T_A, $\langle l_a^1, v \oplus w'\rangle \xrightarrow{\overline{a}} \langle l_a^2, v \oplus w'\rangle$ is the only urgent transition possible from this state and hence $s \xrightarrow{a}$.

Assume now that $\mathcal{T} \preceq \mathcal{T}_A$. In the following we let N, C, and V denote the node set, clock set, and data variable set, respectively, of A. Let R be a timed ready simulation from $\mathcal{T}$ to $\mathcal{T}_A$. We will prove the following invariants for any reachable state $\langle s, \langle n, u \oplus w\rangle\rangle$ in $\mathcal{T} \otimes \mathcal{T}_{T_A}$:

P1. $n \neq l_r$ for any $l \in N$

P2. if $n = l_a^1$ for some $l \in N$, $a \in U_{E,l}$ then $w(x_a) = 0$

P3. if $n \in \{l_a^1, l, l_\tau\}$ for some $l \in N$, $a \in U_{E,l}$ then $(s, \langle l, u\rangle) \in R$ and s and $\langle l, u\rangle$ are reachable in $\mathcal{T}$ and $\mathcal{T}_A$, respectively

Invariant P1 implies that $\mathcal{T}$ passes the A-test. Invariants P2 and P3 are used to strengthen the inductive hypothesis enough to prove P1. We prove P1, P2, and P3 by induction on the length of a transition sequence in $\mathcal{T} \otimes \mathcal{T}_{T_A}$ starting in the initial state.

Let $\langle s_0, \langle n_0, u_0 \oplus w_0 \rangle\rangle$ be the initial state of $\mathcal{T} \otimes \mathcal{T}_{T_A}$. Then $n_0 = l_0$ where l_0 is the initial node of A. Thus $n_0 \notin \{l_r, l_a^1\}$ for any $l \in N$ and $a \in U_{E,l}$. Hence both P1 and P2 holds for the initial state. Finally, $u_0 = v_0$ where v_0 is the initial assignment over C and V and from the definition of R, $(s_0, \langle l_0, v_0 \rangle) \in R$.

Now, assume that $\langle s, \langle n, u \oplus w \rangle\rangle$ is reachable in $\mathcal{T} \otimes \mathcal{T}_{T_A}$ and that P1, P2, and P3 holds in this state.

Suppose $\langle s, \langle n, u \oplus w \rangle\rangle \xrightarrow{\tau} \langle s', \langle n', u' \oplus w' \rangle\rangle$. Then exactly one of the following three cases hold:

a. $s \xrightarrow{\tau} s'$ and $\langle n', u' \otimes w' \rangle = \langle n, u \otimes w \rangle$

b. $s' = s$ and $\langle n, u \oplus w \rangle \xrightarrow{\tau} \langle n', u' \oplus w' \rangle$

c. $s \xrightarrow{a} s'$ and $\langle n, u \oplus w \rangle \xrightarrow{\overline{a}} \langle n', u' \oplus w' \rangle$ for some $a \in \mathcal{A}$

Consider case a. In this case P1 and P2 holds vacuously in $\langle s', \langle n', u' \oplus w' \rangle\rangle$ since $\langle n', u' \oplus w' \rangle = \langle n, u \oplus w \rangle$ and by hypothesis P1 and P2 holds for $\langle n, u \oplus w \rangle$. Consider P3. Suppose $n \in \{l_a^1, l, l_\tau\}$ for some $l \in N$ and $a \in U_{E,l}$. By hypothesis $(s, \langle l, u \rangle) \in R$. Thus, since A is τ-free, $(s', \langle l, u \rangle) \in R$ and since $n' = n$ and $u' = u$ this proves P3 in the case that $n \in \{l_a^1, l, l_\tau\}$. Suppose $n \notin \{l_a^1, l, l_\tau\}$ for any $l \in N$, $a \in U_{E,l}$. Since $n' = n$, P3 holds vacuously.

Consider case b. Assume for the sake of contradiction that $n' = l_r$ for some $l \in N$. From condition 6 in the definition of T_A we have that the only way a state with node component l_r can be reached via a τ-action is if $n = l_a^1$ for some $a \in U_{E,l}$, and $w(x_a) > 0$. This however contradicts P2 for state $\langle s, \langle n, u \oplus w \rangle\rangle$ and thus $n' \neq l_r$ for any $l \in N$ and hence P1 holds. Now, assume that $n' = l_a^1$ for some $l \in N$, $a \in U_{E,l}$. From condition 4 in the definition of T_A it must be that $w'(x_a) = 0$. Hence P2 holds. Finally, from conditions 1 and 4 in the definition of T_A, we have that $n = l$ and either $n' = l_a^1$ or $n' = l_\tau$ for some $l \in N$, $a \in U_{E,l}$. Furthermore, $u' = u$ and from hypothesis $(s, \langle l, u \rangle) \in R$. Hence P3 holds.

Consider case c. Assume for the sake of contradiction that $n' = l_r$ for some $l \in N$. Then from condition 3 in the definition of T_A, $n = l_\tau$, $g_T(u \oplus w)$ where $g_T = \bigwedge_{g \in G_{E,l,a}} \neg g$, $u' = u$, and $w' = w$. From hypothesis $(s, \langle l, u \rangle) \in R$. Thus, since A is τ-free and deterministic there is a unique state $\langle l', v \rangle$ such that $\langle l, u \rangle \xrightarrow{a} \langle l', v \rangle$. Thus there exists g, r such that $l \xrightarrow{g,a,r} l' \in E$ and $g(u)$. Now, $g(u)$ implies $g(u \oplus w)$ since no variables from $\mathcal{C}_E$ occurs in g. However, $g(u \oplus w)$ contradicts $g_T(u \oplus w)$ since $g \in G_{E,l,a}$. Hence $n' \neq l_r$ and P1 holds. From condition 4 in the definition of T_A we can only reach a state with node component l_a^1 by a τ-transition. Thus $n' \neq l_a^1$ for any $l \in N$, $a \in U_{E,l}$ and hence P2 holds vacuously. Now, suppose that $n' \in \{l_a^1, l, l_\tau\}$ for some $l \in N$, $a \in \mathcal{U}_{E,l}$. From condition 2 in the definition of T_A we then have that $n' = l$ and that there exists l', g, r such that $l' \xrightarrow{g,a,r} l \in E$, $n' = l'_\tau$, $g(u)$, $r(u) = u'$, and $w' = w$. Now, since $(s, \langle l', u\rangle) \in R$ and since A is deterministic and τ-free, $\langle l', u\rangle \xrightarrow{a} \langle l, u'\rangle$ is the only a-transition from this state and hence $(s', \langle l, u'\rangle) \in R$. Thus P3 holds.

Now, suppose $\langle s, \langle n, u \oplus w\rangle\rangle \xrightarrow{\epsilon(d)} \langle s', \langle n, u' \oplus w'\rangle\rangle$ for some $d > 0$. Then P1 holds vacuously in $\langle s', \langle n, u' \oplus w'\rangle\rangle$ since by hypothesis P1, $n \neq l_r$ for any $l \in N$. Consider P2. We know that $s \xrightarrow{\epsilon(d)} s'$ and $\langle n, u \oplus w\rangle \xrightarrow{\epsilon(d)} \langle n, u' \oplus w'\rangle$. Now, assume for the sake of contradiction that $n = l_a^1$ for some $l \in N$, $a \in U_{E,l}$. Then from condition 5 in the definition of T_A, $\langle n, u \oplus w\rangle \xrightarrow{\overline{a}}$. From condition 4 in the same definition, $l \xrightarrow{g_T, \tau, \{x_a := 0\}} l_a^1$ where $g_T = \bigvee_{g \in G_{E,l,a}} g$ is the only edge leading to node l_a^1. Since $\langle s, \langle n, u \oplus w\rangle\rangle$ is reachable and since by hypothesis $w(x_a) = 0$, there exists w'' such that $\langle l, u \oplus w''\rangle \xrightarrow{\tau} \langle n, u \oplus w\rangle$. Thus there exists $g \in G_{E,l,a}$ such that $g(u)$. Thus $\langle l, u\rangle \xrightarrow{a}$ and since $(s, \langle l, u\rangle) \in R$ we have that $s \xrightarrow{a}$. This however contradicts our assumption that $\langle s, \langle n, u \oplus w\rangle\rangle \xrightarrow{\epsilon(d)}$. Hence $n \neq l_a^1$ for any $l \in N$, $a \in U_{E,l}$ and P2 holds. Finally, suppose $n \in \{l_a^1, l, l_\tau\}$ for some $l \in N$, $a \in U_{E,l}$. By hypothesis $(s, \langle l, u\rangle) \in R$. Now, since A is τ-free and due to time determinism $\langle l, u\rangle \xrightarrow{\epsilon(d)} \langle l, u + d\rangle$ and $(s', \langle l, u + d\rangle) \in R$. Now, since $u + d = u'$, this proves P3. ■

Part II

Applied Abstraction Strategies

Chapter 4

Burns' Mutual Exclusion Algorithm

This chapter presents our first application of abstraction techniques to prove correctness of distributed systems. We consider as case study, the Burns distributed mutual exclusion algorithm [Bur78]. We prove that the parameterized algorithm guarantees mutual exclusion between any pair of processes, where the parameter is the number of processes running the algorithm. Our proof exploits abstraction, by using a certain *skolemization strategy* to construct a simple 2-process abstract model preserving the required properties from the concrete n-process model. The abstract model preserves not only the behavior of *any* pair of concrete processes, but also the possible effects on such a pair by processes in its environment. Our proof is within the I/O automaton framework and it uses the theory of Chapter 2 to obtain conditions for property preservation. In particular, our notion of parameterized trace simulations will provide preservation conditions that nicely supports the skolemization abstraction strategy. The conditions for property preservation are easily discharged using support from the LP theorem prover, and the abstract algorithm is automatically verified using the SPIN model checker. The results of this chapter have previously been published in [JL98].

4.1 Background an Contributions

Mutual exclusion algorithms are intended to guarantee a set of concurrent processes mutually exclusive access to a single unshareable resource. The exclusion condition to be satisfied by such algorithms states that for any reachable system state there cannot be two processes both in their critical regions. We can interpret the exclusion condition as a simple logical conjunction over the collection of process index pairs.

The skolemization strategy that we apply in the proof for the Burns algorithm utilizes exactly this conjunctive form of properties. Proving mutual exclusion between *all* pairs of processes can be done by considering an *arbitrary* pair of Skolem processes and our abstract model essentially consists of such an arbitrary pair of processes, with additional information representing the effects on such a pair by other processes. We show that this abstraction does indeed preserve the mutual exclusion property between any pair of concrete processes. The preservation proof consists of showing that the abstract model correctly simulates the behaviors of any two processes in the concrete model. Our notion of parameterized trace simulation nicely supports this kind of proof. We formalize our preservation proof in the LP theorem prover, and the proof is carried out with almost no user assistance. Finally, we verify the abstract model using the SPIN model checker.

In [LSW95] the authors apply a skolemization strategy related to ours to give a proof of a timing based mutual exclusion protocol. In their approach, the protocol model as well as the exclusion specification is described as a conjunction of *aspects* or *projective views*, one for each pair of process indices.

4.1.1 Chapter Organization

This chapter is organized as follows. In section 4.2 we introduce the Burns algorithm as well as the mutual exclusion property to be proved. Section 4.3 presents our skolemization abstraction strategy as well as the abstract algorithm and abstract property resulting from its use. In section 4.4 we present the LP supported proof of property preservation and finally in section 4.5 we describe the automatic verification in SPIN of our abstract algorithm.

4.2 Burns' Algorithm

In this section we present Burns n-process mutual exclusion algorithm. The algorithm runs on a shared memory model consisting of n processes together with n shared variables $\mathit{flag}_1, \dots, \mathit{flag}_n$, each flag_i writable by process i and readable by all other processes. Each process i is acting on behalf of a user process which can be thought of as some application program. The processes competes for mutually exclusive access to a shared resource by reading and writing the shared variables in a way determined by the algorithm.

We model the algorithm formally as an I/O automaton *Burns*, which is the composition of a shared memory automaton M and a set of user automata $U_1, \dots, U_n$. Automaton M models the n processes together with the set of shared variables $\mathit{flag}_1, \dots, \mathit{flag}_n$, and it is modelled as one big I/O automaton, where the process and variable structure is captured by means of some locality restrictions on transitions. Each state in M consists of a state for each process i, plus a value for each shared variable flag_i. We specify the transitions of M by giving preconditions and effects for all the actions. For any $i \in \{1, \dots, n\}$ automaton M has actions as shown in Figure 4.1.

The inputs to M are (for all $1 \leq i \leq n$) actions try_i, which models a request by user i to process i for access to the shared resource, and actions $exit_i$, which models an announcement by user i to process i that it is done with the resource. The outputs of M are $crit_i$, which models the granting from process i of the resource to user i, and rem_i, which models process i telling user i that it can continue with the rest of its work.

Each process i executes three loops. The first two loops involve checking the flags of all processes with smaller indices, i.e. all $flag_j$, $1 \leq j < i$. The first loop is actually not needed for the mutual exclusion condition, but is important to guarantee progress. The two loops are modelled in M by internal actions $test\text{-}sml\text{-}fst(j)_i$ and $test\text{-}sml\text{-}snd(j)_i$, where j is a parameter denoting the index of the flag to be read by process i. In between the first two loops process i sets its own $flag_i$ to 1, modelled in M by internal action $set\text{-}flg\text{-}1_i$. If both loops are successfully passed, meaning all the considered flags have value 0, then i can proceed to the third loop, which involves checking the flags of all processes with larger indices, i.e. $flag_j$, $i < j \leq n$. This is modelled by internal action $test\text{-}lrg(j)_i$. If process i passes all three loops successfully, it proceeds to its critical region. Process i keeps the value of its $flag_i$ to 1 from when it starts testing flags with larger indices and until it leaves its critical region. The state of each process i in M is modelled by two state variables local to M: a program counter pc_i initially having the value rem, and a set S_i of process id's initially empty, used to keep track of the indices of all shared flags that have successfully been checked in one of the three loops.

Each user automaton U_i has as single state variable a program counter pc_i, local to U_i and initially having the value rem, indicating that U_i starts in its remainder region ready to make a request for access to the shared resource. We specify the transitions of U_i by giving preconditions and effects for all the different actions of U_i as shown in Figure 4.2.

4.2.1 The Mutual Exclusion Property

We state the mutual exclusion property for *Burns* as a conjunction of trace properties $P_{\{i,j\}}$, one for each subset $\{i, j\}_{i \neq j}$ in the set of process indices $\{1, \ldots, n\}$. The set $sig(P_{\{i,j\}})$ has as its only actions the set of output actions from *Burns* with indices i and j, and $traces(P_{\{i,j\}})$ is the set of sequences such that no two $crit_i$, $crit_j$ events occur (in that order) without an intervening $exit_i$ event, and similarly for i and j switched.

4.3 The Abstraction

To prove the mutual exclusion property we will construct a single finite-state abstract model which preserves the external behavior of *any* two concrete

```
input: $try_i$
    Eff: $pc_i := set\text{-}flg\text{-}0$

internal: $set\text{-}flg\text{-}0_i$
    Pre: $pc_i = set\text{-}flg\text{-}0$
    Eff: $flag_i := 0$
         if $i = 1$ then
           $pc_i := set\text{-}flg\text{-}1$
         else
           $pc_i := test\text{-}sml\text{-}fst$

internal: $test\text{-}sml\text{-}fst(j)_i$
    Pre: $pc_i = test\text{-}sml\text{-}fst$
         $j \notin S_i$
         $1 \leq j \leq i - 1$
    Eff: if $flag_j = 1$ then
           $S_i := \emptyset$
           $pc_i := set\text{-}flg\text{-}0$
         else
           $S_i := S_i \cup \{j\}$
           if $|S_i| = i - 1$ then
             $S_i := \emptyset$
             $pc_i := set\text{-}flg\text{-}1$

internal: $set\text{-}flg\text{-}1_i$
    Pre: $pc_i = set\text{-}flg\text{-}1$
    Eff: $flag_i := 1$
         if $i = 1$ then
           $pc_i := test\text{-}lrg$
         else
           $pc_i := test\text{-}sml\text{-}snd$

internal: $test\text{-}sml\text{-}snd(j)_i$
    Pre: $pc_i = test\text{-}sml\text{-}snd$
         $j \notin S_i$
         $1 \leq j \leq i - 1$
    Eff: if $flag_j = 1$ then
           $S_i := \emptyset$
           $pc_i := set\text{-}flg\text{-}0$
         else
           $S_i := S_i \cup \{j\}$
           if $|S_i| = i - 1$ then
             $S_i := \emptyset$
             if $i = n$ then
               $pc_i := leave\text{-}try$
             else
               $pc_i := test\text{-}lrg$

internal: $test\text{-}lrg(j)_i$
    Pre: $pc_i = test\text{-}lrg$
         $j \notin S_i$
         $i + 1 \leq j \leq n$
    Eff: if $flag_j = 1$ then
           $S_i := \emptyset$
         else
           $S_i := S_i \cup \{j\}$
           if $|S_i| = n - i$ then
             $pc_i := leave\text{-}try$

output: $crit_i$
    Pre: $pc_i = leave\text{-}try$
    Eff: $pc_i := crit$

input: $exit_i$
    Eff: $pc_i := reset$

internal: $reset_i$
    Pre: $pc_i = reset$
    Eff: $flag_i := 0$
         $S_i := \emptyset$
         $pc_i := leave\text{-}exit$

output: $rem_i$
    Pre: $pc_i = leave\text{-}exit$
    Eff: $pc_i := rem$
```

Figure 4.1: Precondition-Effect code for automaton M

```
output: $try_i$                      output: $exit_i$
    Pre: $pc_i = rem$                    Pre: $pc_i = crit$
    Eff: $pc_i := try$                   Eff: $pc_i = exit$

input: $crit_i$                      input: $rem_i$
    Eff: $pc_i := crit$                  Eff: $pc_i := rem$
```

Figure 4.2: Precondition-Effect code for automaton U_i

processes running in the environment of *all* other processes and users. Formally, we construct an abstract automaton $Burns_\alpha$, which is the composition of a shared memory automaton, M_α, with two user automata. Automaton M_α models two abstract processes 0 and 1 together with two shared variables $flag_0$ and $flag_1$. Processes 0 and 1 are abstract representations of any pair of concrete processes i and j within the environment of all other concrete processes. Assuming that $i < j$, the abstract process 0 represents *the smaller concrete process* i and abstract process 1 represents *the larger concrete process* j, both within the environment of all other processes. We also construct an abstract trace property P_α over the external actions of $Burns_\alpha$. This property is basically obtained from the concrete property $P_{\{i,j\}}$ by a change of action indices, 0 for i and 1 for j. We will define relations, $R_{\{i,j\}}$ from the external actions of $Burns$ to the external actions of $Burns_\alpha$ and $S_{\{i,j\}}$ from the states of $Burns$ to the states of $Burns_\alpha$, and we will prove that for any subset $\{i, j\}$:

$$Burns \leq^{\mathrm{t}}_{R_{\{i,j\}}} Burns_\alpha \text{ via } S_{\{i,j\}}, \quad \text{and} \tag{4.1}$$

$$R^{-1}_{\{i,j\}}(traces(P_\alpha)) \subseteq traces(P_{\{i,j\}}). \tag{4.2}$$

Then from Theorem 2.7 (Trace Safety Preservation) we can conclude that if $Burns_\alpha$ satisfies P_α then $Burns$ satisfies $P_{\{i,j\}}$ for any $\{i, j\}$.

4.3.1 Abstract Actions and State Space

The external actions of automaton M_α are the actions try_k, $crit_k$, $exit_k$, rem_k for any $k \in \{0, 1\}$. This set of actions also forms the interface of the two user automata to be composed with automaton M_α. These two automata are obtained from the concrete user automaton U_i by changing all occurrences of index i in actions and states to values 0 and 1, respectively. The abstract user automata are denoted U_0 and U_1. In the following we let i, j be any pair of indices in $\{1, \ldots, n\}$ such that $i < j$. We define a relation $R_{\{i,j\}}$ from the external actions of $Burns$ to the external actions of $Burns_\alpha$ relating actions in the obvious way.

Definition 4.1 *$R_{\{i,j\}}$ is a relation from $ext(Burns)$ to $ext(ABurns)$ such that:*

$$\begin{aligned} R_{\{i,j\}} \;=\; & \{(try_i, try_0), (try_j, try_1), (crit_i, crit_0), (crit_j, crit_1), \\ & (exit_i, exit_0), (exit_j, exit_1), (rem_i, rem_0), (rem_j, rem_1)\} \end{aligned}$$

Thus, abstract actions with index 0 represent the external actions of the (smaller) concrete process i and actions with index 1 represent the external actions of the (larger) concrete process j.

A state of automaton M_α consists of a state for each of the abstract processes 0 and 1 together with values for each of the shared variables $flag_0$ and $flag_1$. The states of the abstract processes 0 and 1 are modelled by variables pc_k and S_k for $k \in \{0,1\}$. Variable pc_k is a program counter and S_k is a set of elements from $\{0,1\}$. The intended interpretation of the introduced state variables is represented in the following definition of the state abstraction relation $S_{\{i,j\}}$.

Definition 4.2 *$S_{\{i,j\}}$ is a relation from $states(Burns)$ to $states(Burns_\alpha)$ such that $S_{\{i,j\}}(s,u)$ iff :*

- *$u.upc_0 = s.upc_i$ and $u.upc_0 = s.upc_1$*
- *$u.ppc_0 = s.ppc_i$ and $u.upc_1 = s.ppc_j$*
- *$u.flag_0 = s.flag_i$ and $u.flag_1 = s.flag_j$*
- *$u.S_0 = \{1\}$ if $j \in s.S_i$ and $u.S_1 = \{0\}$ if $i \in s.S_j$*

We use notation upc_i to denote the value of program counter pc_i in user automaton U_i, and ppc_i to denote the value of the program counter pc_i in automata M and M_α.

4.3.2 The Abstract Property

The abstract mutual exclusion property for $Burns_\alpha$ is the trace property P_α with $sig(P_\alpha)$ having as its actions the external actions of $Burns_\alpha$ and having $traces(P_\alpha)$ as the set of sequences such that no two $crit_0$ and $crit_1$ events occur (in that order) without an intervening $exit_0$ event, and similarly for 0 and 1 switched. Thus, directly by definition we have the following theorem, proving condition (4.2).

Theorem 4.1 *For all i, j, $i \neq j$, $R^{-1}_{\{i,j\}}(traces(P_\alpha)) \subseteq traces(P_{\{i,j\}})$*

```
input: $try_0$
    Eff: $pc := \textit{set-flg-0}$

internal: $\textit{set-flg-0}_0$
    Pre: $pc = \textit{set-flg-0}$
    Eff: $\mathit{flag}_0 := 0$
         $pc := \textit{test-sml-fst}$

internal: $\textit{set-flg-0-sml}_0$
    Pre: $pc = \textit{set-flg-0}$
    Eff: $\mathit{flag}_0 := 0$
         $pc := \textit{set-flg-1}$

internal: $\textit{test-sml-fail}_0$
    Pre: $pc \in \{\textit{test-sml-fst}, \textit{test-sml-snd}\}$
    Eff: $pc := \textit{set-flg-0}$

internal: $\textit{test-sml-fst-succ}_0$
    Pre: $pc = \textit{test-sml-fst}$
    Eff: $pc := \textit{set-flg-1}$

internal: $\textit{set-flg-1}_0$
    Pre: $pc = \textit{set-flg-1}$
    Eff: $\mathit{flag}_0 := 1$
         $pc := \textit{test-sml-snd}$

internal: $\textit{set-flg-1-sml}_0$
    Pre: $pc = \textit{set-flg-1}$
    Eff: $\mathit{flag}_0 := 1$
         $pc := \textit{test-lrg}$

internal: $\textit{test-sml-snd-succ}_0$
    Pre: $pc = \textit{test-sml-snd}$
    Eff: $pc := \textit{test-lrg}$

internal: $\textit{test-other-flg}_0$
    Pre: $pc = \textit{test-lrg}$
         $S = \emptyset$
    Eff: if $\mathit{flag}_1 = 0$ then
           $S := S \cup \{1\}$

internal: $\textit{test-lrg-fail}_0$
    Pre: $pc = \textit{test-lrg}$
    Eff: $S := \emptyset$

internal: $\textit{test-lrg-succ}_0$
    Pre: $pc = \textit{test-lrg}$
         $S = \{1\}$
    Eff: $pc := \textit{leave-try}$

output: $crit_0$
    Pre: $pc = \textit{leave-try}$
    Eff: $pc := crit$

input: $exit_0$
    Eff: $pc := reset$

internal: $reset_0$
    Pre: $pc = reset$
    Eff: $\mathit{flag}_0 := 0$
         $S := \emptyset$
         $pc := \textit{leave-exit}$

output: $rem_0$
    Pre: $pc = \textit{leave-exit}$
    Eff: $pc := rem$
```

Figure 4.3: Transitions of abstract process 0 in M_α

4.3.3 The Abstract Automaton

We now present the abstract automaton M_α by considering the transitions for each of the abstract processes 0 and 1 modelled by M_α. The transitions of process 0 are shown in Figure 4.3.

One of the consequences of having abstract process 0 represent the behavior of *any* smaller process is that the abstract process has two actions for setting its own flag to 0 (1): *set-flg-0-sml*$_0$ (*set-flg-1-sml*$_0$) and *set-flg-0*$_0$ (*set-flg-1*$_0$). The first representing that the concrete process 1 (the one with smallest index) sets its flag to 0 (1), where after it skips the test of flags with smaller indices, as there are none, and sets it program counter to *set-flg-1* (*test-lrg*). The second representing that any other smaller process sets it flag to 0 (1) and thereafter tests flags with smaller indices, which do exist in this case. The abstract process represents that a smaller process fails or succeeds a test of smaller flags by allowing abstract fail or succeed actions whenever its program counter is *test-sml-fst* or *test-sml-snd*. No further preconditions apply to these actions since all information about the actual values of smaller flags have been abstracted away.

In order for abstract process 0 to succeed its test of flags with larger indices, it must test the flag of abstract process 1 since this process represent some larger concrete process. This test is modelled by the action *test-other-flg*$_0$. Having read this flag successfully (i.e. as 0) abstract process 0 can now enter its critical region. Also, as long as the process has program counter *test-lrg* it can at any time perform an abstract action *test-lrg-fail*.

Abstract process 1 is modelled analogously and its transitions are shown in Figure 4.4. Process 1 differs from process 0 only in the implementation of the *test*-actions. In order for process 1 to perform actions *test-sml-fst-succ*$_1$ and *test-sml-snd-succ*$_1$ it must have seen $\mathit{flag}_0 = 0$ since process 0 represents a concrete process having a smaller index than that of the process represented by 1. In contrast, process 0 can perform actions *test-sml-fst-succ*$_0$ and *test-sml-snd-succ*$_0$ without knowing anything about flag_1 since process 1 represents a concrete process with an index larger than that of the process represented by 0. Analogously, process 1 can perform action *test-lrg-succ*$_1$ without knowing anything about flag_0, whereas process 0 must have seen $\mathit{flag}_1 = 0$ to perform *test-lrg-succ*$_0$.

That our abstract automaton preserves the behaviour of any two concrete processes is stated in the following theorem, which proves condition (4.1).

Theorem 4.2 *For all i, j, $i \neq j$, Burns $\leq^{\mathrm{t}}_{R_{\{i,j\}}}$ Burns$_\alpha$ via $S_{\{i,j\}}$*

The proof of Theorem 4.2 is the topic of the following section.

```
input: $try_1$
    Eff: $pc := set\text{-}flg\text{-}0$

internal: $set\text{-}flg\text{-}0_1$
    Pre: $pc = set\text{-}flg\text{-}0$
    Eff: $flag_1 := 0$
         $pc = test\text{-}sml\text{-}fst$

internal: $test\text{-}other\text{-}flg_1$
    Pre: $pc \in \{test\text{-}sml\text{-}fst, test\text{-}sml\text{-}snd\}$
         $S = \emptyset$
    Eff: if $flag_0 = 0$ then
           $S := S \cup \{0\}$

internal: $test\text{-}sml\text{-}fail_1$
    Pre: $pc \in \{test\text{-}sml\text{-}fst, test\text{-}sml\text{-}snd\}$
    Eff: $S := \emptyset$
         $pc := set\text{-}flg\text{-}0$

internal: $test\text{-}sml\text{-}fst\text{-}succ_1$
    Pre: $pc = test\text{-}sml\text{-}fst$
         $S = \{0\}$
    Eff: $S := \emptyset$
         $pc := set\text{-}flg\text{-}1$

internal: $set\text{-}flg\text{-}1_1$
    Pre: $pc = set\text{-}flg\text{-}1$
    Eff: $flag_1 := 1$
         $pc := test\text{-}sml\text{-}snd$

internal: $test\text{-}sml\text{-}snd\text{-}succ_1$
    Pre: $pc = test\text{-}sml\text{-}snd$
         $S = \{0\}$
    Eff: $pc := test\text{-}lrg$

internal: $test\text{-}sml\text{-}snd\text{-}succ\text{-}lrg_1$
    Pre: $pc = test\text{-}sml\text{-}snd$
         $S = \{0\}$
    Eff: $pc := leave\text{-}try$

internal: $test\text{-}lrg\text{-}fail_1$
    Pre: $pc = test\text{-}lrg$
    Eff: $pc := test\text{-}lrg$

internal: $test\text{-}lrg\text{-}succ_1$
    Pre: $pc = test\text{-}lrg$
    Eff: $pc := leave\text{-}try$

output: $crit_1$
    Pre: $pc = leave\text{-}try$
    Eff: $pc := crit$

input: $exit_1$
    Eff: $pc := reset$

internal: $reset_1$
    Pre: $pc = reset$
    Eff: $flag_1 := 0$
         $S := \emptyset$
         $pc := leave\text{-}exit$

output: $rem_1$
    Pre: $pc = leave\text{-}exit$
    Eff: $pc := rem$
```

Figure 4.4: Transitions of abstract process 1 in M_α

4.4 The Simulation Proof

To prove Theorem 4.2 for *all* i, j we prove it for *any* i, j with i and j treated as Skolem constants. For the remaining part of this section we assume that i and j are Skolem constants with $i < j$. To prove the theorem, we need to check the two conditions in the Trace Simulation definition, Definition 2.11. We use our formalization from Chapter 2 of the trace abstraction theory in Larch to discharge the proof obligations. We begin by describing the formalization of automata *Burns* and $Burns_\alpha$ in the Larch Shared Language.

4.4.1 The Automata in LSL

The formalization of automata *Burns* and $Burns_\alpha$ in LSL is a rather straightforward translation from the precondition-effect descriptions given in the preceding sections. In the following we present in detail the translation of the concrete automaton *Burns*.

Recall that automaton *Burns* is defined as the composition of the shared memory automaton M and the set of user automata $U_1, \ldots, U_n$. We formalize automaton *Burns* as a single trait called `AutomatonBurns` representing the above composition. A part of the trait is shown in Figure 4.5. The trait consists of a declaration part in which sorts and operator signatures are declared, and a body part in which the introduced operators are constrained by equations.

The trait begins with an `includes` clause that includes in the theory of trait `AutomatonBurns` the union of theories from the included traits. `AutomatonBurns` *specializes* the state and action sorts from the general `Automaton` trait introduced in Chapter 2. A state of `AutomatonBurns` is a tuple of integer indexed arrays `upc`, `ppc`, `flag` and `S`. For any integer `i`, the elements `upc[i]`, `ppc[i]`, `flag[i]`, and `S[i]` together describe the combined state for user automaton U_i and process i in automaton M.

Actions of `AutomatonBurns` are of two types depending on the number of parameters. The `test`-actions have two parameters, the index of the process performing the test and the index of the flag to be tested. All other actions have a single parameter, being the index of the performing process.

Besides introducing state and action sorts, trait `AutomatonBurns` also introduces a constant `N` denoting the total number of processes and a few operators implementing predicates on integers.

The body of trait `AutomatonBurns` is contained in the *asserts* clause which constrains the operators of the trait. The clause defines the initial state and the actions of the automaton by constraining predicates `start`, `enabled` and `effect`. Figure 4.5 only shows the equations defining the try_i action. The `enabled` predicate states that action try_i is enabled in state s only if the user automaton U_i has its program counter equal *rem*. The `effect` predicate states that as a result of performing action try_i in state s,

```
AutomatonBurns (B): trait

  includes Automaton(B), Integer1(Int), Array1(PPC, Int, PPCs),
           Array1(UPC, Int, UPCs), Array1(Int, Int, FLGs),
           Set1(Int, UIDSET), Array1(UIDSET, Int, UIDSETs)

  PPC enumeration of rem, setflg0, testsmlfst,
     testsmlsnd, setflg1, testlrg, leavetry, crit, reset, leaveexit
  UPC enumeration of rem, try, crit, exit
  States[B] tuple of upc: UPCs, ppc: PPCs, flag: FLGs, S: UIDSETs
  ActionTypes1[B] enumeration of try, setflg0, setflg1, crit, exit,
                  reset, rem
  ActionTypes2[B] enumeration of testsmlfst, testsmlsnd, testlrg

  introduces
   __[__]      : ActionTypes1[B], Int        ->  Actions[B]
   __[__,__]   : ActionTypes2[B], Int, Int   ->  Actions[B]
   unchangedB  : States[B], States[B], Int   ->  Bool
   N           : -> Int
   isIndxB     : Int -> Bool
   isSmlIndx   : Int, Int -> Bool
   isLrgIndx   : Int, Int -> Bool

  asserts
    sort Actions[B] generated freely by __[__], __[__,__]

    with s: States[B], i: Int
      start(s) <=> \A i (isIndxB(i) =>
                          ((s.upc[i] = rem) /\ (s.ppc[i] = rem) /\
                           (s.flag[i] = 0) /\ (s.S[i] = {})));

    with s,s': States[B], i,j: Int
      unchangedB(s, s', i) <=> \A j ((isIndxB(j) /\ j ~= i) =>
                                      ((s'.upc[j] = s.upc[j]) /\
                                       (s'.ppc[j] = s.ppc[j]) /\
                                       (s'.flag[j] = s.flag[j]) /\
                                       (s'.S[j] = s.S[j])));

      enabled(s, try[i]) <=> isIndxB(i) /\
                             s.upc[i] = rem;
      effect(s, try[i], s') <=> s'.upc[i] = try /\ s'.ppc[i] = setflg0 /\
                                s'.flag[i] = s.flag[i] /\ s'.S[i] = s.S[i]
                                /\ unchangedB(s, s', i);
```

Figure 4.5: Trait AutomatonBurns.lsl

```
BurnsSimulation: trait

  includes AutomatonBurns(B), AutomatonABurns(C), ActRel(R,B,C)

  introduces
   I : -> Int
   J : -> Int
   S : States[B], States[C] -> Bool

  asserts
   with s: States[B], u: States[C]
     S(s, u)  <=>  ((u.upc[0] = s.upc[I] /\ u.upc[1] = s.upc[J]) /\
                    (u.ppc[0] = s.ppc[I] /\ u.ppc[1] = s.ppc[J]) /\
                    (u.flag[0] = s.flag[I] /\ u.flag[1] = s.flag[J]) /\
                    (J \in s.S[I] => u.S[0] = {1}) /\
                    (I \in s.S[J] => u.S[1] = {0}))

   with a: Actions[B], a': Actions[C]
     R(a,a')  <=>
      ((a = try[I] /\ a' = try[0]) \/ (a = try[J] /\ a' = try[1]) \/
       (a = crit[I] /\ a' = crit[0]) \/ (a = crit[J] /\ a' = crit[1]) \/
       (a = exit[I] /\ a' = exit[0]) \/ (a = exit[J] /\ a' = exit[1]) \/
       (a = rem[I] /\ a' = rem[0]) \/ (a = rem[J] /\ a' = rem[1]));

   with i: Int   isLrgIndx(J,I);

  implies
    Simulation(B,C,R,S)
```

Figure 4.6: Trait BurnsSimulation.lsl

the user U_i changes its program counter to *try* and process i in M changes its program counter to *set-flg-0*. All other state components are unchanged by the action. The parts of the *asserts* clause not shown in Figure 4.5 consists of a direct translation of remaining transitions in automaton *Burns*.

In a manner completely analogous to the above, we formalize automaton $Burns_\alpha$ into an LSL trait `AutomatonABurns`. Having defined the two automata *Burns* and $Burns_\alpha$ in LSL we now proceed to define the LSL version of relations $R_{\{i,j\}}$ and $S_{\{i,j\}}$ from Definitions 4.1 and 4.2, respectively.

4.4.2 The Simulation Relation in LSL

The LSL formalization of relations $R_{\{i,j\}}$ and $S_{\{i,j\}}$ takes place in the trait `BurnsSimulation` shown in Figure 4.6. The trait fixes the Skolem constants i and j as the integers `I` and `J`, respectively. The LSL definitions of the relations follows directly from Definitions 4.1 and 4.2.

Trait `BurnsSimulation` ends with an `implies` clause, where we state a claim that the theory of trait `BurnsSimulation` logically implies the theory of trait `Simulation`. Trait `Simulation` states the requirements for a trace simulation relation. The trait is defined is defined in Figure 2.3 of Chapter 2 and it implements the requirements from the Trace Simulation definition, Definition 2.11. In the next section we describe how we prove the theory containment claim using the Larch Prover.

4.4.3 The LP Simulation Proof

LSL is supported by a tool, the LSL Checker, which can be used to syntax-check and type-check LSL traits and to extract proof required to check the semantic claims from traits. When running the LSL Checker with input trait `BurnsSimulation`, an input for LP is generated that initiates a proof of the claim from the trait. The first proof obligation to be discharged is the start condition from the Trace Simulation definition, Definition 2.11.

The start condition is trivial, because the initial states of *Burns* and Burns_α have the value of *pc* set to *rem* for all processes and users, and they have all flags set to 0 and all sets of indices empty.

The second proof obligation is the step condition, condition 2, from the Trace Simulation definition. For the step condition suppose that s and u are states of *Burns* and Burns_α, respectively, such that $S_{\{i,j\}}(s,u)$. We then consider cases based on the type of action π_x performed by s on a transition $s \xrightarrow{\pi_x} s'$. For each action π_x we consider $x = i$, $x = j$ and $x \notin \{i,j\}$. The proof is relatively simple, since the execution fragment corresponding to a certain concrete action π_x for the most cases can be picked to be the abstract version of the concrete action. So the proof is a rather straightforward matching up of concrete actions with their abstract counterparts. The main user assistance that LP needs for the proof is the input of the corresponding abstract execution fragment for each concrete action. The rest of the user guidance consists of directing LP to break some proof parts into cases, and directing LP to use whatever information it has already got to try and do some rewriting to complete proof subgoals. Figures 4.7 and 4.8 illustrates the proof in case $\pi_x = \mathit{test\text{-}sml\text{-}fst}(y)_x$, $x = j$, $y = i$, $s.\mathit{flag}_i = 0$, and $s.|S_j| < j - 2$. Figure 4.7 shows the manual proof and Figure 4.8 shows the proof as it is computer-assisted by LP.

The two proofs in Figures 4.7 and 4.8 have the exact same overall structure. The LP proof contains no further user assistance than what is shown. Line 0 shows the overall proof subgoal that leads to the considered case. The subgoal is generated by running the LSL Checker with input trail `BurnsSimulation`. It states as follows. If s and u are states of *Burns* and Burns_α, respectively, such that s and u are related by $S_{\{i,j\}}$, and if $s \xrightarrow{a} s'$ is a step of *Burns* with $a \notin \mathit{dom}(R)$. Then, there exists an execution fragment α of Burns_α with $\mathit{first}(\alpha) = u$, $\mathit{last}(\alpha) = u'$, and $\mathit{trace}(\alpha)|\mathit{ran}(R_{\{i,j\}}) = \lambda$. The proof proceeds by considering the various cases of action a. In Figure

Case $\pi_x = \textit{test-sml-fst}(y)_x$, $x = j$, $y = i$, $s.\mathit{flag}_i = 0$, $|s.S_j| = j - 2$

If $u.S_1 = \{0\}$ the corresponding fragment is $u \xrightarrow{\textit{test-sml-fst-succ}_1} u'$. The fragment is enabled since by definition of state relation $S_{\{i,j\}}$, $u.\mathit{ppc}_1 = s.\mathit{ppc}_j = \textit{test-sml-fst}$ and $u.S_1 = \{0\}$. From *Burns* the only changes resulting from performing action π_x are $s'.S_j = \emptyset$ and $s'.\mathit{ppc}_j = \textit{set-flg-1}$. From Burns_α the only changes are $u'.S_1 = \emptyset$ and $u'.\mathit{ppc}_1 = \textit{set-flg-1}$, so $S_{\{i,j\}}(s', u')$.

If $u.S_1 = \emptyset$ we let the corresponding execution fragment be the following fragment: $u \xrightarrow{\textit{test-other-flg}_1} u'' \xrightarrow{\textit{test-sml-fst-succ}_1} u'$. Action $\textit{test-other-flg}_1$ is enabled in u since by definition of $S_{\{i,j\}}$, $u.\mathit{ppc}_1 = s.\mathit{ppc}_j = \textit{test-sml-fst}$ and $u.S_1 = \emptyset$. From Burns_α, $u''.\mathit{ppc}_1 = \textit{test-sml-fst}$ and $u''.S_1 = \{0\}$ since $u.\mathit{flag}_0 = s.\mathit{flag}_i = 0$. Therefore, $\textit{test-sml-fst-succ}_1$ is enabled in u''. From *Burns* the changes resulting from performing action π_x are $s'.S_j = \emptyset$ and $s'.\mathit{ppc}_j = \textit{set-flg-1}$, and from Burns_α the changes are, $u'.S_1 = \emptyset$ and $u'.\mathit{ppc}_1 = \textit{set-flg-1}$, so $S_{\{i,j\}}(s', u')$.

Figure 4.7: Manual proof

```
(0) prove
     ((S(s, u) /\ inv(s) /\ isStep(s, a, s') /\ ~inR(a) =>
       \E alpha (execFrag(alpha) /\ first(alpha) = u /\
                 S(s', last(alpha)) /\ proR(trace(alpha)) = empty))
    ..

    %% Case a = a3[i1c,i2c], a3 = testsmlfst, i2c = J, i1c = I,
            sc.flag[i1c] = 0, size(sc.S[J] \U {I}) = pred(J)

(1)    resume by case u.S[1] = {0}
(2)      resume by specializing alpha to null(uc){testsmlfstsucc[1], u'c}
(3)        instantiate j by I in *ImpliesHyp*
(4)        instantiate j by 0 in SimulationTheorem* ~ (*Hyp*)
(5)      resume by specializing alpha to
          (null(uc){testotherflg[1], u''c}){testsmlfstsucc[1], u'c}
(6)        instantiate j by I in *ImpliesHyp*
(7)        instantiate j by 0 in SimulationTheorem* ~ (*Hyp*)
```

Figure 4.8: LP proof

4.8 we show the proof case indicated in the remark. In line 1, we tell LP to break the proof into cases based on the value of $u.S_1$. We start by the case $u.S_1 = \{0\}$ and LP automatically provides proof obligations for the other case, $u.S_1 \neq \{0\}$, i.e. $u.S_1 = \emptyset$, as well. The two cases are proved in lines 2–4 and 5–7, respectively. In lines 2 and 5 we direct LP to specialize the execution fragment of automaton $Burns_\alpha$ that we want to correspond to the concrete step of automaton *Burns*. In the LP proof, `uc`, `u'c`, and `u''c` are constants corresponding to u, u', and u'' in the manual proof. The `instantiate` command replace variables by appropriate constants in definitions and case hypotheses, and LP uses the created facts to rewrite the stated conjectures to *true*. Thus, LP automatically checks that the proposed abstract execution fragment is indeed enabled in state u and also that the states s' and u' are related by $S_{\{i,j\}}$. The complete proof of Theorem 4.2 is shown in Appendix A.

4.5 The SPIN Verification

In this section we present the automatic verification of the abstract automaton $Burns_\alpha$ in the SPIN model checker. We translate automaton $Burns_\alpha$ into a PROMELA model and we translate the trace property P_α into an LTL formula suitable for SPIN.

4.5.1 The PROMELA Implementation

Automaton $Burns_\alpha$ is translated into a single PROMELA process called `BurnsAlpha()`. The process has variables representing flags, program counters and index sets of automaton $Burns_\alpha$. Figure 4.9 shows the PROMELA implementation of the state variables. The code should be self-explanatory.

Process `BurnsAlpha` is shown in Figure 4.10. The figure only shows the entries in the `do::od` construction that implements transitions of abstract process 0. Additional entries exist that implements the transitions of process 1. The translation follows the scheme introduced in Chapter 2.

4.5.2 The SPIN Verification

Recall the definition of the abstract trace property P_α. The set $traces(P_\alpha)$ consists of all the sequences of actions in $ext(Burns_\alpha)$ such that no two $crit_0$ and $crit_1$ actions occur (in that order) without an intervening $exit_0$ action, and similarly for indices 0 and 1 switched. Assuming that $Burns_\alpha$ has a state variable v_{act} used to track the most recent action performed by the automaton, we can rephrase the above trace property as the following LTL invariant property:

$$\Box((v_{act} = crit_0) \rightarrow (((v_{act} \neq crit_1)\,\mathcal{U}(v_{act} = exit_0)) \vee \Box(v_{act} \neq crit_1)))\wedge$$
$$\Box((v_{act} = crit_1) \rightarrow (((v_{act} \neq crit_0)\,\mathcal{U}(v_{act} = exit_1)) \vee \Box(v_{act} \neq crit_0)))$$

```
mtype = {rem, try, setflag0, testsmallerfirst, testsmallersecond,
         setflag1, testlarger, leavetry, crit, reset, exit,
         leaveexit, empt, one, zero}

mtype pcP0 = rem;
mtype pcP1 = rem;
mtype pcU0 = rem;
mtype pcU1 = rem;
mtype S0 = empt;
mtype S1 = empt;
bit flag0 = 0;
bit flag1 = 0;
```

Figure 4.9: States of process BurnsAlpha

```
proctype BurnsAlpha()
{
do
:: atomic{ pcU0==rem -> pcU0=try; pcP0=setflag0 }
:: atomic{ pcP0==setflag0 -> flag0=0; pcP0=setflag1 }
:: atomic{ pcP0==setflag0 -> flag0=0; pcP0=testsmallerfirst }
:: atomic{ pcP0==testsmallerfirst -> pcP0=setflag0 }
:: atomic{ pcP0==testsmallersecond -> pcP0=setflag0 }
:: atomic{ pcP0==testsmallerfirst -> pcP0=setflag1 }
:: atomic{ pcP0==setflag1 -> flag0=1; pcP0=testlarger }
:: atomic{ pcP0==setflag1 -> flag0=1; pcP0=testsmallersecond }
:: atomic{ pcP0==testsmallersecond -> pcP0=testlarger }
:: atomic{ (pcP0==testlarger && S0==empt) ->
            if
            :: flag1==0 -> S0=one
            :: else -> skip
            fi }
:: atomic{ pcP0==testlarger -> S0=empt }
:: atomic{ (pcP0==testlarger && S0==one) -> pcP0=leavetry }
:: atomic{ pcP0==leavetry -> pcP0=crit; pcU0=crit }
:: atomic{ pcU0==crit -> pcU0=exit; pcP0=reset }
:: atomic{ pcP0==reset -> flag0=0; S0=empt; pcP0=leaveexit }
:: atomic{ pcP0==leaveexit -> pcP0=rem; pcU0=rem }
..
od
}
```

Figure 4.10: Process BurnsAlpha

```
#define p        (pcP0==crit)
#define q        (pcP1==crit)
#define r        (pcP0==reset)
#define s        (pcP1==reset)

/*  Formula verified:

    ([](p -> (((!q) U r) || ([](!q)))))  &&
    ([](q -> (((!p) U s) || ([](!p)))))

*/
```

Figure 4.11: The LTL property

We can easily extend process `BurnsAlpha` with a variable representing v_{act}. In fact, our general translation scheme introduced in Chapter 2 insists that we do so. However, to simplify our PROMELA code slightly we use the existing state variables of process `BurnsAlpha` to implement the above property. It can easily be observed, that variable `pcP0=crit` whenever $v_{act} = crit_0$ and `pcP0=reset` whenever $v_{act} = exit_0$. Analogously, `pcP1=crit` whenever $v_{act} = crit_1$ and `pcP1=reset` whenever $v_{act} = exit_1$. Thus we verify the LTL property defined in Figure 4.11. SPIN successfully verifies the property from Figure 4.11 of process `BurnsAlpha`. We thus conclude, that the concrete automaton *Burns* satisfies the mutual exclusion property $P_{\{i,j\}}$ for all distinct indices i, j.

Chapter 5

The BCTSS Algorithm

In this chapter we present a formal proof, using abstraction, of one the most complicated algorithms in the distributed systems literature: the Bounded Concurrent Timestamp System (BCTSS) algorithm of Dolev and Shavit [DS89]. We prove a key invariant of the parameterized BCTSS algorithm, where the parameter is the number of processes running the algorithm. Our proof is based on the construction of a finite-state abstraction, that preserves the behavior of the concrete algorithm with respect to the key invariant. The proof is within the I/O automaton framework, and it uses the theory of Chapter 2 to obtain proof obligations for property preservation from the abstract to the concrete algorithm. The proof obligations are discharged manually and the abstract algorithm is automatically verified in the SPIN model checker, using the translation scheme of Chapter 2.

5.1 Background and Contributions

A timestamp system works somewhat like a ticket machine at a bakery, where customers draw tickets when they enter and are served in the order of their ticket numbers. The ticket machine provides a newly arrived customer with a ticket numbered above that of any earlier arrived customer. A person working in the bakery, in order to decide the order customers must be served, need only scan through all the numbers and observe the order among them.

A concurrent timestamp system (CTSS) is a timestamp system in which any process can either take a new ticket or scan the existing tickets simultaneously with other processes. An algorithm implementing a CTSS runs on an asynchronous shared memory model with a set of processes and a set of timestamps, one per process. Each process repeatedly performs either a *label* or a *scan* operation. A label operation consists of a sequence of reads of all timestamps, followed *in a separate step* by a write (update) to the process own timestamp of a value greater than the maximal value read. The values written establish a total order on the label operations with ties broken by process identifiers. A scan operation consists of a sequence of reads of all timestamps, returning a sequence of process indexes ordered consistently with the above total order.

A CTSS is the core in several algorithms for solving fundamental problems in multiprocessor concurrency control. Examples of such algorithms include Lamport's *first come first served* mutual exclusion [Lam74], Vitanyi and Awerbuch's construction of a multi-reader multi-writer atomic register [VA95], Abrahamson's randomized consensus [Abr88], and Afek, Dolev, Gafni, Merritt, and Shavit's *first come first enabled* ℓ-exclusion [ADG+94]. These algorithms are all based on the use of an unbounded concurrent timestamp system (UCTSS), a CTSS in which the timestamps are taken from an unbounded domain, usually the nonnegative reals. This unboundedness is unrealizable in practical implementations of the algorithms, since it allows for behaviours in which timestamp values can grow arbitrarily large. This problem cannot be solved by any simple scheme of cycling through a finite set of integers. Much in analogy with the Year 2000 Problem (Y2K), problems can occur when a new timestamp *wraps around* and starts reusing the smallest value of the domain.

In [DS89], Dolev and Shavit showed that a bounded concurrent timestamp system (BCTSS), a CTSS in which the timestamps are taken from a bounded domain, is constructible. The BCTSS algorithm from [DS89] thus allows for bounded solutions to the concurrency problems referenced above. The particular bounded domain used in the BCTSS algorithm is a certain nested graph, nested to depth $n-1$, where n is the number of processes.

The BCTSS algorithm is widely considered as one the most complicated algorithms in the distributed systems literature. The correctness proof by Dolev and Shavit [DS89], based on ordering relations defined by Lamport [Lam86], is long, detailed, and hard to understand. In [GLS92], Gawlick, Lynch, and Shavit give a correctness proof for a slight simplification of the original BCTSS algorithm, using atomic snapshots of the shared memory. Their proof has a nicer structure than the original proof. It is based on the Input/Output Automaton model [LT89, Lyn96] and uses a set of invariant assertions and a forward simulation mapping [LT87, Lyn96] from the BCTSS model to a model of a UCTSS algorithm. All the complexity of their simulation proof is centered in the use of a key invariant of the BCTSS algorithm. This invariant asserts that in any state, certain sets of timestamps are totally ordered (wrt. a defined timestamp ordering). The proof of the key invariant uses a set of subinvariants, some of which are rather technical and unintuitive, and the proof is somewhat long and detailed (about 10 pages).

We present an alternative proof of the key invariant from [GLS92], using abstraction to combine deductive reasoning with automatic verification. Although it uses a bounded timestamp domain, the BCTSS algorithm is still parameterized in the number n of processes. This implies not only the existence of n timestamps, but also that the domain for these is parameterized by n. We construct a property preserving finite-state abstraction ABCTSS of the n-process BCTSS algorithm. As in [GLS92], our proof is based on the I/O automaton model.

Our abstract ABCTSS algorithm can intuitively be seen as the BCTSS algorithm running on an abstract shared memory model, where the set of n concrete timestamps has been replaced with a finite and nonparameterized set of abstract timestamp *views*, each view having a finite and nonparameterized domain. The timestamp views can be seen as a partitioning of the set of concrete timestamps with respect to a certain equivalence relation. All operations of BCTSS on timestamp variables are replaced in ABCTSS with abstract counterparts operating on the abstracted domains. Our simulation proof, showing property preservation, essentially consists of showing that each of the abstract operators is *homomorphic* with respect to its concrete counterpart. These proofs are easy, since we have intentionally defined the abstract operators with only this one purpose. Only a few subinvariants are used in our simulation proof, all relatively high-level and the proof of these is short (about 3 pages).

5.1.1 Chapter Organization

This chapter is organized as follows. In section 5.2 we introduce an algorithm that implements a UCTSS. The algorithm is simple to understand, and we use it to explain the basic functionality of any CTSS. We illustrate how a CTSS can be used as the core in an algorithm for multiprocessor concurrency control. We end the section with a discussion of the problems involved in going from an unbounded timestamp domain to a bounded domain. In section 5.3 we present the BCTSS algorithm used in the rest of this chapter. The algorithm differs from the UCTSS algorithm only in the underlying timestamp domain and in the implementation of a function that picks new labels for processes. Section 5.3.1 presents the key invariant that we wish to prove about the BCTSS algorithms and section 5.4 presents the high-level proof strategy that we apply for the invariant. The strategy is a combination of induction and abstraction techniques. Section 5.5 provides the bulk of this chapter. Here we present our abstracted version of the BCTSS algorithm as well as our abstracted version of the concrete key invariant and we show that these abstractions satisfy the conditions for property preservation with respect to their concrete counterparts, as stated in the abstraction framework of Chapter 2. Finally, section 5.6 describes the automatic verification in SPIN of our abstract algorithm. Besides verifying the key invariant of interest, we also present a few experiments on the abstract model performed using SPIN. These experiments are used to automatically provide further insight into the workings of the concrete algorithm – in particular into the workings of the bounded timestamp domain.

5.2 The UCTSS Algorithm

In this section we present the UCTSS algorithm from [GLS92]. This unbounded algorithm is simple to understand and use, and it differs from the more complicated BCTSS algorithm, to be introduced later, only in the choice of timestamp domain and in the implementation of a function that picks new timestamps.

The UCTSS algorithm uses as unbounded timestamp domain the set of nonnegative reals, $\mathbf{R}_{\geq 0}$, and it is modeled as an I/O automaton *UCTSS*. Automaton *UCTSS* is the composition of a shared memory automaton and a set of user automata. The shared memory automaton, denoted M, models the n processes in the concurrent timestamp system together with the set of shared timestamp variables. It is modeled as one big I/O automaton, where the process and variable structure is captured by means of some locality restrictions on transitions. For any process index i there exists a user automaton U_i, providing the environment for process i in M.

Each process i in M can perform two operations, a *scan* operation and a *label* operation, both performed upon request from its user U_i. A *scan* operation of process i consists of an input action $beginscan_i$ and an output action $endscan(\overline{s})_i$. The operation performs an atomic snapshot of the set of timestamp variables and returns to user U_i a total ordering of process indexes induced by the order of timestamp values. A *label* operation of process i consists of an input action $beginlabel_i$ and an output action $endlabel_i$. This operation also performs an atomic snapshot of the set of timestamp variables and then computes a new timestamp value for i, greater than the maximal value read. The updating of process i's timestamp variable with the newly computed value is performed in a separate step. In the following we present the shared memory automaton as well as the user automata. In the presentation we will use the words *timestamp* and *label* interchangeably.

Automaton M models the n processes as well as the shared variables. The state of M has the following components, for each $i \in \{1, \ldots, n\}$:

- $t_i \in \mathbf{R}_{\geq 0}$: The current label associated with process i. Initially $t_i = 0$.
- $nt_i \in \mathbf{R}_{\geq 0}$: The new label for i determined by a function *newlabel*. Initially $nt_i = 0$.
- $\overline{t_i} \in \mathbf{R}^n_{\geq 0}$: An array of labels returned by an action $snap_i$. Initially $\overline{t_i} = 0^n$.
- $\overline{o}_i \in \{1, \ldots, n\}^n$: An array of process indexes ordered based on an order $<<$. Initially $\overline{o}_i = (1 \ldots n)$.
- $pc_i \in \{nil, snap, update, endscan, endlabel\}$: The non-input action currently enabled. Initially $pc_i = nil$.

```
input: $beginscan_i$
    Eff: $op_i := scan$
         $pc_i := snap$

internal: $snap_i$
    Pre: $pc_i = snap$
    Eff: $\overline{t_i} := (t_1 \ldots t_n)$
         if $op_i = scan$ then
           $\overline{o}_i :=$ sequence of indexes s.t.
              $j <_{\overline{o}} k$ iff $(t_j, j) << (t_k, k)$
           $pc_i := endscan$
         if $op_i = label$ then
           if $i_{max} = - \vee i = i_{max}$ then
             $pc_i := endlabel$
           else
             $nt_i := newlabel(i, \overline{t_i})$
             $pc_i := update$

output: $endscan(\overline{s})_i$
    Pre: $pc_i = endscan$
         $\overline{s} = \overline{o}_i$
    Eff: $pc_i := nil$

input: $beginlabel_i$
    Eff: $op_i := label$
         $pc_i := snap$

internal: $update_i$
    Pre: $pc_i = update$
    Eff: $t_i := nt_i$
         $pc_i := endlabel$

output: $endlabel_i$
    Pre: $pc_i = endlabel$
    Eff: $pc_i := nil$
```

Figure 5.1: Precondition-Effect code for automaton M

– $op_i \in \{nil, scan, label\}$: The current operation. Initially $op_i = nil$.

All of the above variables are local to automaton M. For any i, the variables with this index models the variables "belonging to" process i. Variable t_i, models the current timestamp of process i. The t_i models a shared variable, writable by process i and readable by all processes. Any other variable with index i models a variable local to process i.

We specify the transitions of M by giving preconditions and effects for all the actions. For any $i \in \{1, \ldots, n\}$ automaton M has actions as shown in Figure 5.1.

Action $beginscan_i$ just sets the operation counter op_i to $scan$ and then enables the atomic snapshot action $snap_i$. The $snap_i$ action first reads (atomically) the value of all timestamp variables into variable $\overline{t_i}$. Then, within the $scan$ operation, the $snap_i$ action sets $\overline{o}_i$ to the total ordering of process indexes given by the timestamp labels in the atomic snapshot, with ties broken by process index (the $<<$ order). Notice, that in the $snap_i$ action, we refer directly to a shared variable t_j rather than its local copy t_{i_j} in vector $\overline{t_i}$. This is safe due to the fact that effects are atomic and $\overline{t_i} = (t_1 \ldots t_n)$ in the effect of $snap_i$. Action $endscan(\overline{s})_i$ resets the pc_i variable to nil and returns the current $\overline{o}_i$ to user U_i.

Action $beginlabel_i$ sets the operation counter op_i to $label$ and then enables

```
output: $beginscan_i$
    Pre: $pc_i = nil$
    Eff: $pc_i := snap$

input: $endscan(\overline{s})_i$
    Eff: $pc_i := nil$

output: $beginlabel_i$
    Pre: $pc_i = nil$
    Eff: $pc_i := snap$

input: $endlabel_i$
    Eff: $pc_i := nil$
```

Figure 5.2: Precondition-Effect code for automaton U_i

the snapshot action $snap_i$. Within the *label* operation, the $snap_i$ action sets the "local" variable nt_i to the new timestamp value for i, computed by the *newlabel* function. The updating of the "shared" variable t_i is performed in a separate step by action $update_i$. Action $endlabel_i$ ends the *label* operation by simply resetting the program counter pc_i to *nil*. Notice, that a new label for i is only computed, in action $snap_i$, under the condition that: $i_{max} \neq -$ and $i \neq i_{max}$. In any state of *UCTSS* we have derived variables t_{max} and i_{max}. Variable t_{max} is the maximal value held by any timestamp variable, $t_{max} = max(t_1, \dots, t_n)$, and i_{max} is the largest process index i such that $t_i = t_{max}$. The value $-$ is used to denote that i_{max} is "undefined" in the case that no maximal timestamp value t_{max} exists. For the unbounded domain, $\mathbf{R}_{\geq 0}$, used in *UCTSS* such a maximal value always exists, since the usual $<$ relation on $\mathbf{R}_{\geq 0}$ is a total ordering. For the bounded domain, to be introduced later, the timestamp relation, also to be introduced later, only defines a total ordering on a subset of the domain. Thus, we cannot immediately conclude that a maximal timestamp value exists in any state of the bounded algorithm.

We now formally define the $<<$ order and the *newlabel* function used in *UCTSS*. For any state of *UCTSS* define as follows.

Definition 5.1 ($<<$ order) *$(t_i, i) << (t_j, j)$ iff either $t_i < t_j$ or $t_i = t_j$ and $i < j$.*

Definition 5.2 *If $i \neq i_{max}$ then,*

$$newlabel(i, \overline{t_i}) = t_{max} + X$$

where X is nondeterministically selected from $\mathbf{R}_{>0}$

Each automaton U_i has a single local variable, pc_i, a program counter, initially having the value *nil*. We specify the transitions of U_i by giving preconditions and effects for all the different actions of U_i as shown in Figure 5.2.

The parameter $\overline{s}$ of input action $endscan(\overline{s})_i$ is the array of process indexes returned by the *scan* operation of process i in M.

Process i :

```
(1)   $choosing(i) := 1$
(2)   $number(i) := 1 + \max_{j \neq i} number(j)$
(3)   $choosing(i) := 0$
(4)   for $j \neq i$ do
(5)       if $choosing(j) \neq 0$ then goto (5)
(6)       if $number(j) \neq 0$ and $(number(j), j) < (number(i), i)$ then goto (6)
(7)   ** critical region **
(8)   $number(i) := 0$
(9)   ** noncritical region **
(10)  goto (1)
```

Figure 5.3: Pseudo-code for process i in Bakery algorithm

5.2.1 An Application

As we have mentioned before, a CTSS is the core in several algorithms for multiprocessor concurrency control. The UCTSS algorithm presented so far provides the service of a CTSS. To better understand the use of the algorithm, we now present an application using it as underlying service. The application is Lamport's first-come first-serve mutual exclusion algorithm [Lam74], better known as the *Bakery algorithm.* We first present the Bakery algorithm without the explicit use of the UCTSS algorithm. Our presentation follows closely the original presentation in [Lam74]. We then present the Bakery algorithm rewritten to make use of the UCTSS algorithm as underlying service. This presentation follows closely a presentation of Gawlick in [Gaw92]. Actually, the latter algorithm merely assumes an underlying CTSS with an action interface identical to the one for the UCTSS algorithm. Thus, it is independent of the underlying timestamp domain being unbounded or bounded, and the BCTSS algorithm to be presented can replace the UCTSS algorithm without causing any changes.

The presentation of the standard Bakery algorithm as well as the Bakery algorithm using a CTSS service, called Bakery-CTSS, will be in informal pseudo-code style rather than in I/O automaton language. Our intention is to provide a high-level understanding of the use of a CTSS as service for an application, not to prove properties of the Bakery algorithm.

The Bakery algorithm runs on a shared memory model, where processes communicate using single-writer/multi-reader shared variables. Besides guaranteeing mutual exclusion between any pair of processes, the Bakery algorithm also guarantees a certain FIFO (first-in first-out) property among processes waiting to enter their critical regions. Any process i controls two variables, $choosing(i) \in \{0, 1\}$ and $number(i) \in \mathbf{N}$, both writable by i and readable by all $j \neq i$, and both initially 0. The algorithm run by any process i is presented in Figure 5.3.

Process $\boldsymbol{i}$:

```
(1)   $choosing(i) := 1$
(2)   $beginlabel_i$
(3)   $endlabel_i$
(4)   $choosing(i) := 0$
(5)   for $j \neq i$ do
(6)       if $choosing(j) \neq 0$ then goto (6)
(7)       $beginscan_i$
(8)       $endscan(\overline{s})_i$
(9)       if $choosing(j) = 0$ and $j < i$ in $\overline{s}$ then goto (7)
(10)  ** critical region **
(11)  $choosing(i) := nil$
(12)  ** noncritical region **
(13)  goto (1)
```

Figure 5.4: Pseudo-code for process i in Bakery-CTSS algorithm

Process i is said to be *in the doorway* while $choosing(i) = 1$, i.e. while in lines (1)-(2). While in the doorway, process i chooses a *number* that is greater than all the numbers that it reads for the other processes. It reads the other processes' *number*s one at a time, in any order, then writes its own *number*. While it is reading and choosing numbers, i makes sure that $choosing(i) = 1$, as a signal to the other processes. It is possible for two processes to be in the doorway at the same time, which can cause them to choose the same number. To break such ties, processes compare their $(number, index)$ pairs. The comparison it done lexicographically, thus breaking ties in favor of the process with the smaller index. In the remaining part of the *trying* region, the process waits for the other processes to finish choosing and also waits for its $(number, index)$ pair to become the lowest.

Mutual exclusion follows from an easily proved invariant, stating that for any two processes i and j, $i \neq j$, if i is in the critical region (line (7)) and j is in either the critical region (line (7)) or in the part of the trying region outside the doorway (lines (3)-(6)), then $(number(i), i) < (number(j), j)$.

In the Bakery-CTSS algorithm we assume that each process i can use the service provided by the the shared memory automaton M from the UCTSS algorithm. That is, process i has output actions $beginscan_i$ and $beginlabel_i$, and input actions $endscan(\overline{s})_i$ and $endlabel_i$. The CTSS will take care of the handling of timestamps. Hence, there is no need in the Bakery algorithm, running on top of the CTSS, to deal with this. Therefore, process i will not need the *number* variable anymore. Only variable $choosing(i)$ is needed. The pseudo-code for algorithm Bakery-CTSS is shown in Figure 5.4.

To compare with the Bakery algorithm in Figure 5.3, the following have changed. In Bakery-CTSS, lines (2)-(3) take the place of line (2) in Bakery, and lines (7)-(9) take the place of line (6). Moreover, since the *number* variable no longer exists in Baker-BCTSS, the *choosing* variable, with an extension of its domain to include a special *nil* value, is used to replace the first condition in line (6). The condition $number(j) \neq 0$ is replace by $choosing(j) = 0$.

Note, that Bakery-CTSS is independent of the implementation of the timestamp domain in the underlying CTSS service. Thus, the BCTSS algorithm, to be introduced later, can provide the underlying service. This gives a bounded version of the Bakery algorithm.

5.2.2 From Unbounded to Bounded Timestamp Domain

It is easy to see, that the unbounded timestamp domain of the UCTSS algorithm always allows processes to pick new timestamps ordered above the timestamps of all other processes. However, it is not obvious how the same property can be obtained within a bounded domain. A first intuitive idea might be to use a simple wrap around strategy to cycle through some bounded domain. In the following we will examine the inadequacies of this simple strategy, thereby hopefully providing some intuition for the more complicated bounded timestamp domain used in the BCTSS algorithm of the next section.

Consider a system of n processes and assume that the timestamp domain consists of the natural numbers from 0 to n with the usual $<$ ordering. As before, ties between processes are broken based on process index. The obvious problem with this label set is deciding what happens when some process has the label n and another process needs a new, bigger label. According to the simple wrap around strategy, the latter process would pick the number 0 as new label. Thus, the ordering among labels would have the additional feature that $n < 0$.

Using this strategy provides a good solution for two processes. In particular consider two processes p_1 and p_2 with label set $\{0, 1, 2\}$. The wrap around strategy obviously works in this case since there will always be an extra number between the labels of p_1 and p_2 to make sure that they are totally ordered. Figure 5.5 illustrates the above label set with the extended $<$ order, in the situation where p_1 has label 1 and p_2 has label 2.

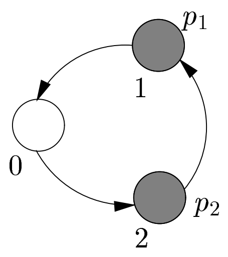


Figure 5.5: The extended $<$ order on $\{0, 1, 2\}$

The wrap around strategy does however not work for three processes. Consider the following situation for three processes. Let each p_i have label i and assume that process p_2 with label 2 wants a new label that is bigger than the label 3 of p_3. The situation is illustrated in Figure 5.6.

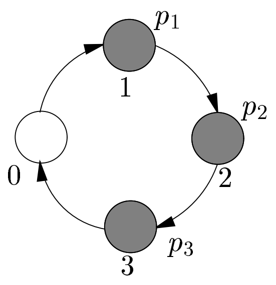


Figure 5.6: The extended $<$ order on $\{0, 1, 2, 3\}$

Using the wrap around strategy, the new label for p_2 would be 0. However, now process p_2's label is ordered below that of p_1 which has label 1. This violates the ordering properties of a timestamp system since p_1's current label was acquired before p_2 acquired the label 0. One solution might be to extend the set of numbers from which the labels are chosen so that the wrap around happens later. However, it is easy to see that this will not help. In particular, processes p_2 and p_3 can ask for new labels alternately until one of them reaches the highest label. The first process to wrap around will encounter the same problem as identified above. What is needed is the ability to create a cycle of numbers for processes p_2 and p_3 such that all numbers in that cycle are ordered above the label of p_1.

Consider as alternative label domain the set $\{0, 1, 2\}^2$ equipped with a label ordering being the lexicographical order based on the the usual $<$ order with the additional feature that $2 < 0$. This new ordering can be represented as a nested version of the graph in Figure 5.5. Figure 5.7 illustrates this representation. For clarity we have omitted directions on the edges as well as numbering of the nodes on the subgraphs, but these graphs are merely copies of the graph in Figure 5.5. Figure 5.7 illustrates a situation where processes p_1, p_2, and p_3 have labels 0.0, 0.1, and 1.1, the labels being ordered in the order of process indexes.

Consider again now the situation in which processes p_2 and p_3 alternately asks for new labels. Starting with p_2, they can pick labels alternately in the following sequence: 1.1, 1.2, 1.0, 1.1, 1.2, 1.0, ... In other words, they can use the size three cycle defined by labels with prefix 1. The size three cycle is large enough to accommodate the two processes since there will always be an extra number between the labels of p_2 and p_3 to make sure that they are totally ordered. Moreover, the labels of p_2 and p_3 are always ordered above p_1's label of 0.0. If at some point p_1 becomes active, it can pick a new label

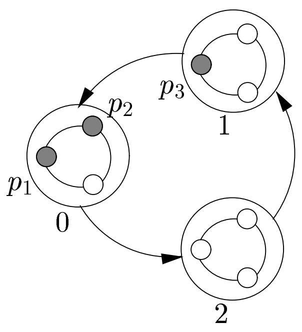


Figure 5.7: The lexicographical order based on the extended $<$ order

2.0 which will be higher than the two labels of p_2 and p_3, both having prefix 1.

The label set from above can be generalized to work for any number n of processes. The generalized label set being $\{0,1,2\}^n$ and the label ordering being the lexicographical order based on the extended $<$ order on the set $\{0,1,2\}$. The label ordering can be represented as the graph in Figure 5.5 nested to depth $n-1$.

In the above discussion, we have implicitly assumed that processes compute new labels and update their timestamp variables in a single atomic step. When considering the more general setting in which the above atomicity cannot be guaranteed, the label set from above needs further generalization. The generalized domain is the topic of the next section in which we present the BCTSS algorithm.

5.3 The BCTSS Algorithm

In this section we present the BCTSS algorithm from [GLS92]. The algorithm differs from the UCTSS algorithm from the previous section only in the underlying timestamp domain and in the implementation of the *newlabel* function. We denote by *BCTSS* the automaton obtained from automaton *UCTSS* by changing the timestamp domain and the *newlabel* function as presented in this section.

The timestamp domain is a generalization of the nested cycle structure introduced in section 5.2.2. The generalization is required to guarantee that the set of process timestamps is always totally ordered in the setting where processes can pick and update new labels non-atomically. To provide some intuition for the timestamp domain, we examine the inadequacies of the nested cycle structure from section 5.2.2 in this setting. Consider the situation illustrated in Figure 5.7, in which processes p_1, p_2, and p_3 have labels 0.0, 0.1, and 1.0, respectively. More precisely, in terms of the variables in the shared memory automaton M of *BCTSS*, $t_1 = 0.0$, $t_2 = 0.1$, and

$t_3 = 1.0$. Now, suppose that p_2 and p_1 both want to pick a new label. Observing the maximal label having the value 1.0, they can both pick as new value, 1.1. Thus, p_1 and p_2 set their "local" variables nt_1 and nt_2 to the value 1.1. Now, p_2 can update its timestamp variable, i.e. it can set $t_2 = nt_2 = 1.1$ and p_2 and p_3 can start to alternately pick new timestamps within the cycle having prefix 1. At some point they may end up in a situation where $t_2 = 1.0$ and $t_3 = 1.2$. Notice, they are still ordered above the timestamp $t_1 = 0.1$ of process p_1. However, at this point p_1 may choose to update its timestamp variable, thus setting $t_1 = nt_1 = 1.1$. Now the three processes have picked labels on each of the nodes in the size three cycle with prefix 1 which implies that the labels are no longer totally ordered. No maximal timestamp exists. This unwanted behavior can be removed by enforcing processes to *move through* a few nodes not occurring in any cycle, before they *enter* a cycle.

We now present the label domain and ordering used by automaton *BCTSS*. We introduce the set $\mathcal{A} = \{1 \dots 5\}$ and we define an order $\prec_{\mathcal{A}}$ on the elements of $\mathcal{A}$ as follows.

Definition 5.3 ($\prec_{\mathcal{A}}$ order) *Define $\prec_{\mathcal{A}}$ as,*

$$1 \prec_{\mathcal{A}} 2,3,4,5; \quad 2 \prec_{\mathcal{A}} 3,4,5; \quad 3 \prec_{\mathcal{A}} 4; \quad 4 \prec_{\mathcal{A}} 5; \quad 5 \prec_{\mathcal{A}} 3.$$

The graph in Figure 5.8 represents the order $\prec_{\mathcal{A}}$, where $a \prec_{\mathcal{A}} b$ iff there is a directed edge from b to a. Note that the ordering is not a partial order, since it is not transitive – it only gives pairwise ordering relationships between nodes. In comparison with the graph of Figure 5.5 we observe that the graph of Figure 5.8 also has a size three cycle. This cycle consists of nodes 3, 4, and 5. Moreover, the graph also has two additional nodes 1 and 2 ordered below any node within the cycle.

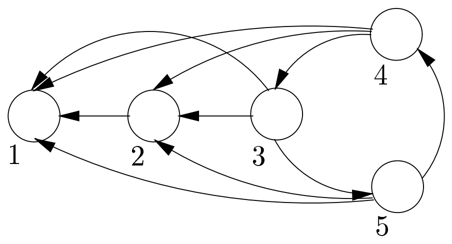


Figure 5.8: The $\prec_{\mathcal{A}}$ order

We define a function *next* on the elements of $\mathcal{A}$. For any k, $next(k)$ returns the least k' such that $k \prec_{\mathcal{A}} k'$.

Definition 5.4 (*next*) *For $k \in \mathcal{A}$ define the function next as,*

$$next(k) = \begin{cases} k+1 & \text{if } k \in \{1,2,3,4\} \\ 3 & \text{if } k = 5 \end{cases}$$

A label is an element of $\mathcal{A}^{n-1}$ where n is the number of processes in the system. All t- and nt-variables of *BCTSS* are initially set to 1^{n-1}. The order between labels that is used in the BCTSS algorithm is based upon the following order $\prec$, defined between elements of $\mathcal{A}^m$ for any natural number m. This order is the lexicographic ordering on $\mathcal{A}^m$ based on the $\prec_{\mathcal{A}}$ order. We refer to elements of $\mathcal{A}^m$ using array notation. Specifically, the h'th digit of $\ell \in \mathcal{A}^m$ will be denoted by $\ell[h]$. In the following let m be any natural number.

Definition 5.5 ($\prec$ order) *Let ℓ_1, ℓ_2 be elements of $\mathcal{A}^m$. Then, $\ell_1 \prec \ell_2$ iff there exists $h \in \{1 \dots m\}$ such that $\ell_1[h'] = \ell_2[h']$ for all $h' < h$ and $\ell_1[h] \prec_{\mathcal{A}} \ell_2[h]$.*

The order between labels used by *BCTSS* is the $\prec$ order on elements of $\mathcal{A}^{n-1}$, where n is the number of processes. This order can be seen as the nesting of the graph in Figure 5.8 to depth $n-1$.

In the following we will state and prove a lemma that gives a necessary and sufficient condition for a set of elements from $\mathcal{A}^m$, m any natural number, to be totally ordered. This condition will later serve as the basis in our definition of the concrete key invariant that we want to prove of *BCTSS*. The following lemma shows that any two elements of $\mathcal{A}^m$ are always totally ordered by the $\prec$ order.

Lemma 5.1 *If ℓ_1 and ℓ_2 are elements of $\mathcal{A}^m$, then exactly one of the following is true: $\ell_1 \prec \ell_2$,$\ell_2 \prec \ell_1$, or $\ell_1 = \ell_2$.*

Proof. If $a, b \in \mathcal{A}$, then by definition of $\prec_{\mathcal{A}}$ exactly one of the following is true: $a \prec_{\mathcal{A}} b$, $b \prec_{\mathcal{A}} a$, or $a = b$. The lemma now follows since $\prec$ is a lexicographical order defined by $\prec_{\mathcal{A}}$. ■

Definition 5.6 *If $\ell \in \mathcal{A}^m$ and $h \in \{1, \dots, m\}$, then ℓ^h is the prefix of ℓ up to and including digit h; ℓ^0 is the empty prefix denoted ϵ.*

We can now state and prove the lemma giving a necessary and sufficient condition for a set of elements from $\mathcal{A}^m$ to be totally ordered.

Lemma 5.2 *A set $\mathcal{L}$ of elements from $\mathcal{A}^m$ is totally ordered by $\prec$ iff for all ℓ_1, ℓ_2, ℓ_3 in $\mathcal{L}$ and for all $h \in \{1, \dots, m\}$,*

$$\neg(\ell_1^{h-1} = \ell_2^{h-1} = \ell_3^{h-1} \ \wedge \ \{\ell_1[h], \ell_2[h], \ell_3[h]\} = \{3,4,5\})$$

Proof. $\Rightarrow$: Assume for the sake of contradiction that there exists ℓ_1, ℓ_2, ℓ_3 $\in \mathcal{L}$ and $h \in \{1, \dots, m\}$ such that $\ell_1^{h-1} = \ell_2^{h-1} = \ell_3^{h-1}$ and $\{\ell_1[h], \ell_2[h], \ell_3[h]\}$ $= \{3,4,5\}$. By definition of $\mathcal{A}$ we can conclude without loss of generality that $\ell_1[h] \prec_{\mathcal{A}} \ell_2[h] \prec_{\mathcal{A}} \ell_3[h]$ and $\ell_1[h] \not\prec_{\mathcal{A}} \ell_3[h]$. Since $\ell_1^{h-1} = \ell_2^{h-1} = \ell_3^{h-1}$

and $\prec$ is a lexicographical order, $\ell_1 \prec \ell_2 \prec \ell_3$, and $\ell_1 \not\prec \ell_3$. Hence $\prec$ is not transitive, contradicting our hypothesis that $\prec$ is a total order.

$\Leftarrow$: Assume for the sake of contradiction that $\prec$ is not total. By definition $\prec$ is irreflexive and by Lemma 5.1 it is antisymmetric. Therefore, it must be that transitivity does not hold. Specifically, there must exist ℓ_1, ℓ_2, $\ell_3 \in \mathcal{L}$ such that $\ell_1 \prec \ell_2 \prec \ell_3$, and $\ell_1 \not\prec \ell_3$. Suppose, for the sake of contradiction, that $\ell_1 = \ell_3$. We then have $\ell_1 \prec \ell_2$ and $\ell_2 \prec \ell_1$ which directly contradicts Lemma 5.1. Thus, $\ell_3 \prec \ell_1$. Knowing, that $\ell_1 \prec \ell_2$, $\ell_2 \prec \ell_3$, and $\ell_3 \prec \ell_1$, we now show that there exists $h \in \{1, \dots, m\}$ such that $\ell_1^{h-1} = \ell_2^{h-1} = \ell_3^{h-1}$ and $\ell_1[h] \prec_{\mathcal{A}} \ell_2[h]$, $\ell_2[h] \prec_{\mathcal{A}} \ell_3[h]$ and $\ell_3[h] \prec_{\mathcal{A}} \ell_1[h]$. Assume for the sake of contradiction that no such h exists. By definition of $\prec$, we know that there exists h, h', and h'' such that all of the following three hold. (1): $\ell_1^{h-1} = \ell_2^{h-1}$ and $\ell_1[h] \prec_{\mathcal{A}} \ell_2[h]$. (2): $\ell_2^{h'-1} = \ell_3^{h'-1}$ and $\ell_2[h'] \prec_{\mathcal{A}} \ell_3[h']$. (3): $\ell_3^{h''-1} = \ell_1^{h''-1}$ and $\ell_3[h''] \prec_{\mathcal{A}} \ell_1[h'']$. Furthermore, by assumption not all of h, h', and h'' can be equivalent. Assume without loss of generality that $h < h'$. From (2), $\ell_2^{h'-1} = \ell_3^{h'-1}$ and since $h < h'$ we can conclude from (1) that $\ell_1^{h-1} = \ell_3^{h-1}$ and $\ell_1[h] \prec_{\mathcal{A}} \ell_3[h]$, i.e. $\ell_1 \prec \ell_3$. This however contradicts (3) by which $\ell_3 \prec \ell_1$. Thus, there exists $h \in \{1, \dots, m\}$ such that $\ell_1^{h-1} = \ell_2^{h-1} = \ell_3^{h-1}$ and $\ell_1[h] \prec_{\mathcal{A}} \ell_2[h]$, $\ell_2[h] \prec_{\mathcal{A}} \ell_3[h]$ and $\ell_3[h] \prec_{\mathcal{A}} \ell_1[h]$. Now, by definition of $\prec_{\mathcal{A}}$, $\{\ell_1[h], \ell_2[h], \ell_3[h]\} = \{3, 4, 5\}$ contradicting our hypothesis that $\neg$ ($\ell_1^{h-1} = \ell_2^{h-1} = \ell_3^{h-1} \wedge \{\ell_1[h], \ell_2[h], \ell_3[h]\} = \{3, 4, 5\}$). ■

Having defined the notions of labels and orderings used by automaton *BCTSS*, we can now present the new implementation of the $<<$ order and the *newlabel* function.

Definition 5.7 ($<<$ order) *$(t_i, i) << (t_j, j)$ iff either $t_i \prec t_j$ or $t_i = t_j$ and $i < j$.*

To define the new *newlabel* we require a set of preliminary definitions.

Definition 5.8 (t_{max}^h) *For any $h \in \{1, \dots, n-1\}$, if for some index i, $t_j^h \preceq t_i^h$ for all $j \neq i$ then $t_{max}^h = t_i^h$; Otherwise, $t_{max}^h = -$. We define $t_{max}^0 = \epsilon$. We write t_{max} for t_{max}^{n-1}. If $t_{max} \neq -$ then i_{max} is the largest index i such that $t_i = t_{max}$; Otherwise, $i_{max} = -$.*

In any state, t_{max}^h returns the maximal value of the set of h-prefixes of t-labels; if such a maximal value exists (we will prove that t_{max}^h always exists). Otherwise, t_{max}^h returns $-$. If $t_{max} \neq -$ then i_{max} returns the maximal index of the set of processes having their timestamp value equal to t_{max}.

Suppose $t_{max}^h \neq -$. For any $h' < h$, we will use parentheses $(t_{max}^h)^{h'}$ to denote the prefix of t_{max}^h up to and including digit h'.

Lemma 5.3 *For any $h \in \{1, \dots, n-1\}$, if $t_{max}^h \neq -$ then for any $h' \in \{0, \dots, h-1\}$, $(t_{max}^h)^{h'} = t_{max}^{h'}$.*

Proof. Assume for the sake of contradiction that $t^h_{max} \neq -$ and for some $h' < h$, $(t^h_{max})^{h'} \neq t^{h'}_{max}$. Let i be such that $t^h_i = t^h_{max}$. Then by assumption there exists $k \neq i$ such that $t^{h'}_k \succ t^{h'}_i$. But then by definition of $\preceq$ and since $h' < h$, $t^h_k \succ t^h_i$ which contradicts that $t^h_i = t^h_{max}$. Hence for all $k \neq i$, $t^{h'}_k \preceq t^{h'}_i$ and by definition then $t^{h'}_i = t^{h'}_{max}$. ■

In the following definitions we assume that $t_{max} \neq -$. We do so since all the definitions are used to implement the function *newlabel* and as can be seen from the code of automaton M, see Figure 5.1, this function is only called in states where $t_{max} \neq -$. When $t_{max} \neq -$ we have from Lemma 5.3 that $t^h_{max} = (t_{max})^h$ for any $h \in \{0, \ldots, n-1\}$ (note $t^0_{max} = \epsilon$). The following list of functions, on states of $BCTSS$, forms the basis of the *newlabel* function.

Definition 5.9 (*agree*, *num*, *num$_i$*) *For any $h \in \{0, \ldots, n-1\}$, $\ell \in \mathcal{A}^{n-1}$, and $v \in \mathcal{A}$,*

$$\begin{array}{lcl} agree(\ell^h) & = & \{i \mid t^h_i = \ell^h\} \\ num(\ell^h) & = & |agree(\ell^h)| \\ num_i(\ell^h) & = & |agree(\ell^h) - \{i\}| \\ num(\ell^h, v) & = & |agree(\ell^h) \cap \{i \mid t_i[h+1] = v\}| \quad (h \neq n-1) \end{array}$$

In any state, $agree(\ell^h)$ returns the set of process indexes i such that the prefix t^h_i is equivalent to the prefix ℓ^h of label ℓ. $num(\ell^h)$ is the cardinality of $agree(\ell^h)$. $num_i(\ell^h)$ is the cardinality of $agree(\ell^h)$ once index i is removed from $agree(\ell^h)$. Finally, $num(\ell^h, v)$ is the cardinality of the intersection between $agree(\ell^h)$ and the set of indexes i such that the $(h+1)$st element of t_i equals the value v.

Definition 5.10 (*full$_i$*) *For any $i \in \{1, \ldots, n\}$ and $h \in \{1, \ldots, n-1\}$,*

$$full_i(h) = \begin{cases} true & \text{if } num_i(t^h_{max}) \geq n-h \\ false & \text{otherwise} \end{cases}$$

In any state, $full_i(h)$ returns *true* if at least $n-h$ t-labels, excluding t_i, agree with the prefix of t_{max} up to and including the h'th digit.

Definition 5.11 (*next-label*) *For any $h \in \{1 \ldots n-1\}$, $\ell_2 = next\text{-}label(\ell_1, h)$ iff $\ell^{h-1}_2 = \ell^{h-1}_1$, $\ell_2[h] = next(\ell_1[h])$ and $\ell_2[h'] = 1$ for all $h' > h$.*

Definition 5.12 (*newlabel*) *For any $i \in \{1, \ldots, n\}$, if $i \neq i_{max}$ then,*

$$newlabel(i, \overline{t_i}) = next\text{-}label(t_{max}, h')$$

where, $h' = min\{h \in \{1, \ldots, n-1\} \mid full_i(h) = true\}$.

Function *newlabel*$(i, \overline{t_i})$ finds the minimum integer h such that $full_i(h)$ returns *true*. That is the minimum h such that at least $n - h$ t-labels, excluding t_i, agree with the prefix of t_{max} up to and including the h'th digit. Then the new label is the same as t_{max} for the first $h - 1$ digits, it differs from t_{max} at the h'th digit based on the function *next*, and its remaining digits are equal to 1. We know that the *newlabel* function is executed by a process i of automaton M, see Figure 5.1, only in states where $i \neq i_{max}$. In such a state there always exist an $h \in \{1, \ldots, n-1\}$ such that $full_i(h)$ returns *true*, since $num_i(t_{max}^{n-1}) \geq 1$ and hence $full_i(n-1) = true$.

The graph in Figure 5.9 illustrates the $\prec$ order on the label domain $\mathcal{A}^2$ used when the number n of processes is three. For clarity we have omitted directions on edges and numbering of nodes in the subgraphs. The subgraphs are all identical to the graph of Figure 5.8.

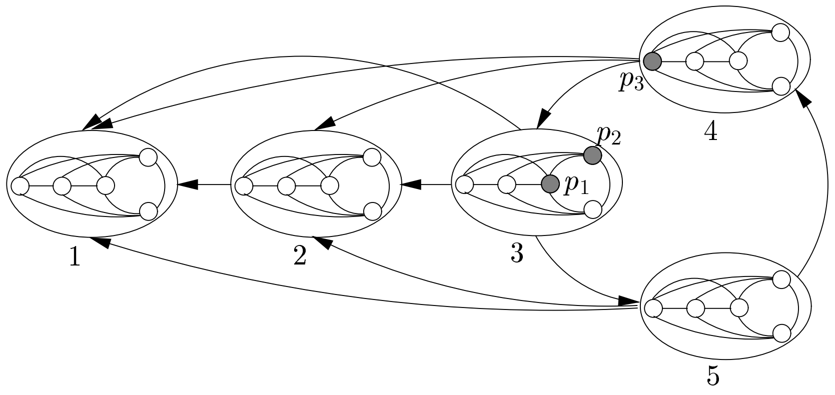


Figure 5.9: The $\prec$ order for $n = 3$

Consider the situation in which processes p_1, p_2, and p_3 have their timestamp variables as illustrated in Figure 5.9. That is, $t_1 = 3.3$, $t_2 = 3.4$, and $t_3 = 4.1$. Suppose now that processes p_2 and p_3 starts to alternately pick new labels. Then they can use the cycle defined by labels with prefix 4 to keep picking labels ordered above the label of p_1. The cycle can accommodate the two processes, guaranteeing that they are always totally ordered. Let us examine in detail how the *newlabel* function will provide the new labels for processes p_2 and p_3. Suppose p_2 is the first process to call the *newlabel* function. The function first determines the minimum level $h \in \{1, 2\}$ such that $full_2(h)$ returns *true*. In terms of the graph in Figure 5.9, $full_2(h)$ returns *true* if the node at level h, holding the maximal timestamp, cannot accommodate more processes. A node at level 1 is a node in the outer graph, that is a node defined by a set of labels agreeing in the first digit. A node at level 2 is a node in some inner graph, that is a node defined by exactly one label value. In our example, $full_2(1)$ returns *false*. Thus, the node at level 1 holding the maximal timestamp, that is the node numbered 4, is not full yet. This node can hold two processes. Also, $\mathit{full}_2(2)$ returns *true* implying that the node at level 2 that holds the maximal timestamp, that is the node 4.1, is full. This node can hold only one process. The rule used by function *newlabel*() to pick a new label is now to choose lowest node dominating the node 4.1. The new label for p_2 thus becomes $\mathit{next\text{-}label}(t_{max}, 2)$ $= \mathit{next\text{-}label}(4.1, 2) = 4.2$. Suppose p_2 updates it timestamp variable leaving $t_2 = 4.2$. Now, when p_3 calls the *newlabel* function it will see $\mathit{full}_3(1) = \mathit{false}$ and $\mathit{full}_3(2) = \mathit{true}$. Note that even though both p_2 and p_3 resides in the node 4 at level 1, $\mathit{full}_3(1)$ returns *false* since the timestamp of p_3, the calling process, is not taken into account when determining the full criteria. The new label for p_3 becomes $\mathit{next\text{-}label}(4.2, 2) = 4.3$. Processes p_2 and p_3 will if they continue to pick labels use the cycle in subgraph 4. In here they will always be totally ordered and also ordered above the label 3.3 of p_1. If at some point p_1 decides to pick a new label, it will observe that $\mathit{full}_1(1) = \mathit{true}$ which tells p_1 that no further processes can be accommodated by the subgraph numbered 4. Thus, p_1 will pick as new label the value $\mathit{next\text{-}label}(4.5, 1)$ $= 5.1$, assuming that $t_{max} = 4.5$.

Using Figure 5.9 we can also illustrate the role played by the values 1 and 2 in the label domain. Consider the situation from above, where p_1 has label $t_1 = 3.3$, p_2 has label $t_2 = 3.4$, and p_3 has label $t_3 = 4.1$. Then subgraph 4 will be the next graph in which new labels will be picked. Suppose p_2 is the next process to pick a new label. It sets $nt_2 = 4.2$. Now, before p_2 updates its t-variable, assume that process p_1 picks a new label and updates its timestamp variable. Process p_1 thus picks $nt_1 = 4.2$ and subsequently sets $t_1 = 4.2$. Now, processes p_3 and p_1 can continue to pick labels ending up in the cycle of subgraph 4. At some point p_2 may complete the update of its timestamp variable, writing $t_2 = 4.2$. This value however, will not be in the cycle of the subgraph. Moreover, any subsequent label operation of a process will see the subgraph 4 as full and hence move onto subgraph 5, preventing three processes to occur in the cycle of subgraph 4.

As mentioned earlier, the user automata guarantees well formedness of the behaviors of *BCTSS*. This implies for instance, that in any behavior of *BCTSS*, for any process i, no two $\mathit{beginlabel}_i$ actions can occur without an intervening pair of consecutive actions, update_i, $\mathit{endlabel}_i$. As a result, the following simple invariant, used in the later sections, holds for the states of *BCTSS*. The proof follows by trivial induction on the length of an execution.

Lemma 5.4 *For any reachable state of BCTSS, for any* $i \in \{1, \ldots, n\}$,

$$M.pc_i = \mathit{snap} \;\Rightarrow\; t_i = nt_i$$

5.3.1 The Total Orderedness Property

The original requirement specification for the BCTSS algorithm, as stated in [DS89], uses an axiomatic specification formalism of Lamport [Lam86] to define a set of ordering properties with respect to the *label* and *scan* operations of the algorithm. These properties have been widely criticized as "hard-to-use" and the original proof in [DS89] is long and difficult to understand. In [GLS92], Gawlick, Lynch, and Shavit provides a proof of the BCTSS algorithm introduced in previous section, showing that it implements an unbounded concurrent timestamp system (UCTSS) algorithm, which can remarkably easy be shown to satisfy the original ordering properties. All the complexity in the (simulation) proof that the BCTSS algorithm implements the UCTSS algorithm, is centered in the use of a key invariant of the BCTSS algorithm. This key invariant asserts that in any state of the algorithm, the set of all timestamps, newly picked as well as already updated, is totally ordered. The proof in [GLS92] of the invariant uses a set of subinvariants, some of which are rather technical and unintuitive, and the proof is somewhat long and detailed. We consider an alternative proof of this invariant, using abstraction strategies.

In order to define the key property (invariant) that we wish to prove, we define the notion of a choice vector for any state of automaton $BCTSS$.

Definition 5.13 (Choice Vector) *A choice vector is any vector $(\ell_1 \ldots \ell_n)$ such that $\ell_i \in \{t_i, nt_i\}$ for each i.*

Our goal is to prove the invariant that for any reachable state of $BCTSS$, the set of values in every choice vector is totally ordered by $\prec$. Notice that this invariant implies that the $<<$ order used in automaton M defines a total order. Proving the invariant is equivalent be Lemma 5.2 to proving the following theorem.

Theorem 5.1 (Total Orderedness) *For any reachable state s of $BCTSS$, for any $h \in \{1, \ldots, n-1\}$, for any choice vector $(\ell_1 \ldots \ell_n)$, and for any indexes i, j, k such that $i \neq j$, $i \neq k$, and $j \neq k$,*

$$\neg(s.\ell_i^{h-1} = s.\ell_j^{h-1} = s.\ell_k^{h-1} \ \wedge\ \{s.\ell_i[h], s.\ell_j[h], s.\ell_k[h]\} = \{3, 4, 5\})$$

5.4 The Proof Strategy

Our proof for the Total Orderedness theorem (Theorem 5.1) uses a combination of induction and abstraction. Induction is used as the high-level proof strategy with abstraction applied in the inductive step.

Consider Theorem 5.1. We will prove the theorem by strong induction on h (note n is fixed). For any $h \in \{1, \ldots, n-1\}$ we let $\psi(h)$ denote the property obtained from Theorem 5.1 by eliminating the universal quantification of h.

Thus, h occurs free in $\psi(h)$.

$\psi(h) =$ For any reachable state s of *BCTSS*, for any choice vector $(\ell_1, \dots, \ell_n)$, and for any indexes i, j, k such that $i \neq j$, $i \neq k$, and $j \neq k$,

$$\neg\,(s.\ell_i^{h-1} = s.\ell_j^{h-1} = s.\ell_k^{h-1} \;\wedge$$
$$\{s.\ell_i[h], s.\ell_j[h], s.\ell_k[h]\} = \{3,4,5\})$$

Thus, we want to prove that $\psi(h)$ holds for all h, by strong induction on h. For any state s of *BCTSS*, we will use the notation $\psi(h,s)$ to denote the property obtained from $\psi(h)$ by eliminating the universal quantification of s. Thus, $\psi(h)$ is the property that for all reachable states s of *BCTSS*, $\psi(h,s)$ holds. In the proof of $\psi(h)$ we will be using a slightly weaker property $\varphi(h)$, defined as follows,

$\varphi(h) =$ For any reachable state s of *BCTSS*, for any choice vector $(\ell_1, \dots, \ell_n)$, and for any indexes i, j, k such that $i \neq j$, $i \neq k$, and $j \neq k$,

$$\neg\,(s.\ell_i^{h-1} = s.\ell_j^{h-1} = s.\ell_k^{h-1} = s.t_{max}^{h-1} \;\wedge$$
$$\{s.\ell_i[h], s.\ell_j[h], s.\ell_k[h]\} = \{3,4,5\} \;\wedge$$
$$num(s.t_{max}^{h-1}) \leq n-h+1)$$

For states s of *BCTSS*, we will use the notation $\varphi(h,s)$ to denote the property obtained from $\varphi(h)$ be eliminating the universal quantification of s. Now, to prove $\psi(h)$ we proceed according to the following inductive strategy.

Assume for induction hypothesis,

$$\forall h',\ 1 \leq h' < h,\quad \psi(h'). \tag{5.1}$$

Prove,

$$\varphi(h) \Rightarrow \psi(h) \quad \text{and} \quad \varphi(h). \tag{5.2}$$

For the remainder of this chapter we thus consider the proof of (5.2) under the assumption of hypothesis (5.1). In this section we prove the first part of (5.2), saying that $\varphi(h) \Rightarrow \psi(h)$, by a simple inductive argument. The proof of the second part, $\varphi(h)$, will be by the use of abstraction strategies. This proof is the topic of the rest of this chapter.

Lemma 5.5 $\varphi(h) \Rightarrow \psi(h)$

Proof. Assume $\varphi(h)$. We now prove $\psi(h)$ by showing that $\psi(h, s)$ holds for all reachable states s of *BCTSS*. We proceed by induction on the length of an execution in *BCTSS*.
Base: For any initial state s_0 and for any index i, $s_0.t_i = s_0.nt_i = 1^{n-1}$. This immediately proves the base case.
Step: Assume that state s satisfies $\psi(h, s)$. For any π, s' such that $s \xrightarrow{\pi} s'$ we will show that s' satisfies $\psi(h, s')$. Assume for the sake of contradiction that for some choice vector $(\ell_1 \ldots \ell_n)$, and for some indexes i, j, k such that $i \neq j$, $i \neq k$, and $j \neq k$,

$$s'.\ell_i^{h-1} = s'.\ell_j^{h-1} = s'.\ell_k^{h-1} \;\wedge\; \{s'.\ell_i[h], s'.\ell_j[h], s'.\ell_k[h]\} = \{3,4,5\} \;(*)$$

First suppose, $\pi \in \{\mathit{beginlabel}_l, \mathit{endlabel}_l, \mathit{beginscan}_l, \mathit{endscan}_l\}$. Since no t-labels or nt-labels change as a result of π, $(*)$ immediately leads to a contradiction with the $\psi(h, s)$. Now, suppose $\pi = \mathit{update}_l$. The only label that changes as a result of π is t_l, so $l \in \{i, j, k\}$ since otherwise $(*)$ immediately contradicts $\psi(h, s)$ for s. Assume without loss of generality that $l = k$ and $s'.\ell_k = s'.t_l$. Now, since $i \neq k$ and $j \neq k$ we have that $s'.\ell_i = s.\ell_i$ and $s'.\ell_j = s.\ell_j$. And since $s'.t_k = s.nt_k$ we have from $(*)$,

$$s.\ell_i^{h-1} = s.\ell_j^{h-1} = s.nt_k^{h-1} \;\wedge\; \{s.\ell_i[h], s.\ell_j[h], s.nt_k[h]\} = \{3,4,5\}$$

which contradicts $\psi(h, s)$. Now, finally suppose that $\pi = \mathit{snap}_l$. $s.op_l = \mathit{label}$ since otherwise no t-labels or nt-labels change as a result of π and hence $(*)$ immediately leads to a contradiction with $\psi(h, s)$. By same argument $s.t_{max} \neq -$, $l \neq s.i_{max}$, and $l \in \{i, j, k\}$. Assume without loss of generality that $l = k$ and $s'.\ell_k = s'.nt_k$. From $(*)$ then,

$$s'.\ell_i^{h-1} = s'.\ell_j^{h-1} = s'.nt_k^{h-1} \;\wedge\; \{s'.\ell_i[h], s'.\ell_j[h], s'.nt_k[h]\} = \{3,4,5\}$$

We know that $s'.nt_k = \mathit{next\text{-}label}(s.t_{max}, h')$ for some $h' \in \{1, \ldots, n-1\}$. Also by assumption $s.t_{max} \neq -$. Suppose $\mathit{num}(s.t_{max}^{h-1}) > n-h+1$. Then $\mathit{num}_k(s.t_{max}^{h-1}) \geq n-h+1$ and $\mathit{full}_k(h-1) = \mathit{true}$. Therefore $h' \leq h-1$ and $s'.nt_k[h] = 1$ contradicting that $s'.nt_k[h] \in \{3,4,5\}$. Suppose $\mathit{num}(s.t_{max}^{h-1}) \leq n-h+1$. Then $\mathit{num}_k(s.t_{max}^{h-1}) \leq n-h+1$. If $h' < h$ we reach the same contradiction as above. If $h' \geq h$ then $s'.nt_k^{h-1} = s.t_{max}^{h-1}$. Now, from action, $s'.t_{max} = s.t_{max}$ so $s'.t_{max}^{h-1} = s.t_{max}^{h-1}$. Hence, $s'.nt_k^{h-1} = s'.t_{max}^{h-1}$ contradicting $\varphi(h, s')$. ■

5.5 The Abstraction

In this section we consider the proof of the invariant $\varphi(h)$ from the second part of (5.2). Recall that h denotes our induction constant. From the induction hypothesis (5.1), if $h > 1$ then in any state and for any choice vector, the set consisting of the $h-1$ prefixes of the choice vector elements is totally ordered. Thus, $t^{h-1}_{max} \neq -$ for any $h \geq 1$ ($t^0_{max} = \epsilon$). That is, there exists $i \in \{1, \ldots, n\}$ such that $t^{h-1}_i = t^{h-1}_{max}$.

Our proof will use the abstraction framework introduced in Chapter 2. The proof strategy will be as follows. First, consider the path safety property with state set *states* (*BCTSS*) and with path set being the set of sequences of states s satisfying $\varphi(h, s)$. We identify the invariant $\varphi(h)$ with this induced path property. Thus our goal is to show that *paths* (*BCTSS*) $\subseteq$ *paths* ($\varphi(h)$).

We construct an abstract automaton, $BCTSS_\alpha(h)$, as well as an abstract path property $\varphi_\alpha(h)$. We establish an abstraction relation, S, between the states of *BCTSS* and the states of $BCTSS_\alpha(h)$ and we prove that:

$$BCTSS \leq^{\mathrm{p}} BCTSS_\alpha(h) \text{ via } S, \quad \text{and} \tag{5.3}$$

$$S^{-1}(\mathit{paths}\,(\varphi_\alpha(h))) \subseteq \mathit{paths}\,(\varphi(h)). \tag{5.4}$$

Then by Theorem 2.8 (Path Safety Preservation) we can conclude that if $BCTSS_\alpha(h)$ satisfies $\varphi_\alpha(h)$ then *BCTSS* satisfies $\varphi(h)$. Our abstract automaton will have a finite state model and we therefore use automatic verification (model checking) to prove that $BCTSS_\alpha(h)$ satisfies $\varphi_\alpha(h)$.

In our proof we will make use of the following list of simple invariants on *BCTSS*. These invariants allow us to restrict the amount of information needed to be preserved in the abstraction. Recall that $t^{h-1}_{max} \neq -$.

Lemma 5.6 *For any reachable state of BCTSS, for any* $i \in \{1, \ldots, n\}$,

1. $(\forall j \neq i,\ t^{h-1}_j \neq t^{h-1}_{max}) \ \Rightarrow\ t^{h-1}_i = nt^{h-1}_i = t^{h-1}_{max}$
2. $t^{h-1}_i = t^{h-1}_{max} \ \Rightarrow\ nt^{h-1}_i \succeq t^{h-1}_{max}$
3. $nt^{h-1}_i \succ t^{h-1}_{max} \ \Rightarrow\ nt_i[h] = 1$

Proof. See Appendix B. ■

In the following we give an intuitive explanation of the claims of Lemma 5.6. In the following when we mention t- or nt-values we will mean their $h-1$-prefix. The first invariant states that, if only a single process has its t-value agreeing with t^{h-1}_{max}, then this process' t- and nt-values agree. This will imply, by total orderedness of the h–1-prefixes of timestamps, that a process who's nt-value is greater than t^{h-1}_{max} will have its t-value agree with t^{h-1}_{max} after performing an update action. Suppose that s denotes the state before an *update*$_k$ of process k and s' denotes the state after the action. If $s.nt^{h-1}_k \succ s.t^{h-1}_{max}$, claim 1 for s tells us that there exists $i \neq k$ such that $s.t^{h-1}_i = s.t^{h-1}_{max}$. Thus for any $j \neq i$, $j \neq k$, $s.t^{h-1}_j \preceq s.t^{h-1}_i \prec s.nt^{h-1}_k$. From induction hypothesis (5.1) this implies that $s.t_j^{h-1} \prec s.nt_k^{h-1}$ and hence for any $i \neq k$, $s.t_i^{h-1} \prec s.nt_k^{h-1}$. Now, for any $i \neq k$, $s'.t_i^{h-1} = s.t_i^{h-1}$ and moreover $s'.t_k^{h-1} = s.nt_k^{h-1}$. This concludes that $s'.t_i^{h-1} \prec s'.t_k^{h-1}$ for all $i \neq k$ and thus $s'.t_k^{h-1} = s'.t_{max}^{h-1}$. The second invariant states that the newly picked label of a process is greater than or equal to t_{max}^{h-1}. The third invariant states that, if a process' newly picked label is greater than t_{max}^{h-1} then the h'th digit of this label equals 1. This invariant, together with the first invariant, provides us with information about the relation between processes timestamps and t_{max}^{h-1} after an update action, performed by a process who's nt-value is greater than t_{max}^{h-1}. In this particular situation, we will be able to conclude, that the updating process will be the only one agreeing with t_{max}^{h-1} after the update, and moreover its t-value will have its h'th digit equal 1.

5.5.1 Abstract State Space

The fundamental requirement to our abstract automaton, $BCTSS_\alpha(h)$, is that it must be property preserving with respect to the concrete path property $\varphi(h)$. Recall that *paths* $(\varphi(h))$ is the set of sequences of states s of *BCTSS* in which the predicate $\varphi(h, s)$ holds. Hence, this predicate will provide us with information about the parts of a concrete state that needs to be preserved in a corresponding abstract state. The predicate can be seen as a guideline for partitioning the concrete state space into a set of abstract states. The basis of the abstract state space will be two abstract domains of *views* and *num-counts*, respectively.

Views

Consider any state of *BCTSS* in which $num(t_{max}^{h-1}) \leq n-h+1$. From the definition of the predicate $\varphi(h, s)$, we observe the need to preserve the following information from the concrete state s into a corresponding abstract state. For any $i \in \{1, \dots, n\}$ and $\ell_i \in \{t_i, nt_i\}$, we must preserve information telling us whether $\ell_i^{h-1} = t_{max}^{h-1}$, and if so, also whether or not $\ell_i[h]$ equals 3, 4, or 5. We define the information that we will preserve in terms of a process *view*.

Definition 5.14 *For any state of BCTSS and for any $i \in \{1, \dots, n\}$,*

$$view_i^h = \begin{cases} (t_i[h], nt_i[h]) & \textit{if} \quad t_i^{h-1} = t_{max}^{h-1} \ \wedge \ nt_i^{h-1} = t_{max}^{h-1} \\ (t_i[h], 0) & \textit{if} \quad t_i^{h-1} = t_{max}^{h-1} \ \wedge \ nt_i^{h-1} \succ t_{max}^{h-1} \\ (0, nt_i[h]) & \textit{if} \quad t_i^{h-1} \prec t_{max}^{h-1} \ \wedge \ nt_i^{h-1} = t_{max}^{h-1} \\ (0,0) & \textit{otherwise} \end{cases}$$

That this definition actually provides us with the intended information follows from the following lemma. Let $view_i^h[1]$ and $view_i^h[2]$ denote the first and second component of $view_i^h$, respectively.

Lemma 5.7 *For any state of BCTSS, for any $i \in \{1, \dots, n\}$, and for any $v \in \mathcal{A}$,*

1. $t_i^{h-1} = t_{max}^{h-1} \wedge t_i[h] = v \;\Leftrightarrow\; view_i^h[1] = v.$
2. $nt_i^{h-1} = t_{max}^{h-1} \wedge nt_i[h] = v \;\Leftrightarrow\; view_i^h[2] = v.$

Proof. Let v be any element of $\mathcal{A}$. Consider 1. Suppose $t_i^{h-1} = t_{max}^{h-1}$ and $t_i[h] = v$. From Lemma 5.6 part 2, $nt_i^{h-1} \succeq t_{max}^{h-1}$ and 1 follows from Definition 5.14. Suppose $view_i^h[1] = v$. Since $v \neq 0$ we have directly from Definition 5.14, $t_i^{h-1} = t_{max}^{h-1}$ and $t_i[h] = v$. Consider 2. Suppose $nt_i^{h-1} = t_{max}^{h-1}$ and $nt_i[h] = v$. From definition of t_{max}^{h-1}, $t_i^{h-1} \preceq t_{max}^{h-1}$ and hence directly from Definition 5.14, $view_i^h[2] = v$. Suppose $view_i^h[2] = v$. Again, since $v \neq 0$ we have from Definition 5.14, $nt_i^{h-1} = t_{max}^{h-1}$ and $nt_i[h] = v$. ■

Notice that for any process i, we actually preserve more information in $view_i^h$ than what we initially motivated. Namely, in case $\ell_i^{h-1} = t_{max}^{h-1}$ we preserve the actual value of $\ell_i[h]$ - not only when it equals 3, 4, or 5, but also when it equals 1 or 2. We will see later that this is necessary in order for our abstract algorithm to be precise enough to satisfy the abstract property of interest. We now introduce the following state variable for $BCTSS_\alpha$.

- $V \subseteq (\mathcal{A} \cup \{0\})^2$: A set intended to describe the set of process views in a corresponding concrete state, initially $\{(1, 1)\}$.

Since we have not yet fully specified the abstract state space, we cannot yet define the abstraction relation S between concrete and abstract states. However, we will at this point state a requirement to S, which describes the intended interpretation of the abstract set V. The requirement will later be part of the definition of S. Let s and u be states of $BCTSS$ and $BCTSS_\alpha$, respectively. At this point all we know about u is that $u.V$ exists. If $S(s, u)$ then,

$$num(s.t_{max}^{h-1}) \leq n-h+1 \;\Rightarrow\; u.V = \{(v, w) \mid \exists i : \; s.view_i^h = (v, w)\}$$

From the definition of predicate φ, we only need to preserve the information about process views from concrete states where $num(t_{max}^{h-1}) \leq n-h+1$. This explains the implication in the above requirement to the abstraction relation S.

Several processes may have identical views, but our abstract set V does not preserve information about the number of such processes. We will see that the only information necessary for property preservation is whether one or more processes have a particular view, and this information will be preserved in a different abstract state variable.

Num-counts

We want the relation S to be a path simulation relation from $BCTSS$ to $BCTSS_\alpha(h)$. Therefore, from the requirement to S stated in the previous section, $BCTSS_\alpha(h)$ must be able to match any view-changing action of $BCTSS$ that preserves $num(t^{h-1}_{max}) \leq n-h+1$, such that the concrete view-changes are preserved in the abstract state variable V. The view of a concrete process may change as a result of this process either picking a new nt-value or updating its t-value. In the latter case the view change simply consists of copying the second view element into the first position. Thus, in this case no additional information beside the view needs to be preserved from the concrete state. In the former case the view change is determined by the result of the *newlabel* function. In order for our abstraction to be precise enough, we must preserve some information, in abstract states, about the values, in corresponding concrete states, of the state functions used by function *newlabel*. In particular, we must preserve information about the values of $num(t^{h-1}_{max})$ and $num(t^{h-1}_{max}, v)$ for any $v \in \mathcal{A}$. Actually, it will be enough to preserve only certain critical ranges of values for these concrete functions. As we will explain in detail later, these value ranges provide sufficient information to let concrete view changes be matched "closely enough" in corresponding abstract states. We will represent the critical ranges of $num(t^{h-1}_{max})$ and $num(t^{h-1}_{max}, v)$ by abstract state variables np and n_v, respectively. The critical value domain will be represented by a single abstract domain of num-counts, denoted $\mathcal{N}$.

Definition 5.15 $\mathcal{N} = \{\mathbf{0},\ \mathbf{1},\ (\mathbf{1}, \mathbf{n}-\mathbf{h}),\ \mathbf{n}-\mathbf{h},\ \mathbf{n}-\mathbf{h}+\mathbf{1},\ (\mathbf{n}-\mathbf{h}+\mathbf{1}, \mathbf{n}]\}$.

We define a function, g, being the (obvious) mapping from elements of $\{1, \ldots, n\}$ to elements of $\mathcal{N}$.

Definition 5.16 *Define function* $g : \{1, \ldots, n\} \to \mathcal{N}$ *as follows,*

$$g(x) = \begin{cases} \mathbf{0} & \textit{if} \ \ x = 0 \\ \mathbf{1} & \textit{if} \ \ x = 1 \wedge h \neq n-1 \\ (\mathbf{1}, \mathbf{n}-\mathbf{h}) & \textit{if} \ \ x > 1 \wedge x < n-h \\ \mathbf{n}-\mathbf{h} & \textit{if} \ \ x = n-h \\ \mathbf{n}-\mathbf{h}+\mathbf{1} & \textit{if} \ \ x = n-h+1 \\ (\mathbf{n}-\mathbf{h}+\mathbf{1}, \mathbf{n}] & \textit{if} \ \ x > n-h+1 \wedge x \leq n \end{cases}$$

Notice, if $h = n-1$ then $n-h = n-(n-1) = 1$. Therefore, $g(1) = g(n-h) = \mathbf{n}-\mathbf{h}$ in this case. We now introduce the following abstract state variables of $BCTSS_\alpha(h)$.

– $np \in \mathcal{N}$: The abstract interpretation of $num(t^{h-1}_{max})$. Initially, $np = \mathbf{n}-\mathbf{h}+\mathbf{1}$ if $h = 1$, $np = (\mathbf{n}-\mathbf{h}+\mathbf{1}, \mathbf{n}]$ otherwise.

– $n \in \mathcal{N}^{\mathcal{A}}$: Each n_v the abstract interpretation of $num(t^{h-1}_{max}, v)$. Initially, $n_1 = \mathbf{n{-}h{+}1}$ if $h = 1$, $n_1 = (\mathbf{n{-}h{+}1}, \mathbf{n}]$ otherwise, $n_v = \mathbf{0}$ for $v \neq 1$.

The variables introduced above together with the abstract view set V, fully characterizes the state space of our abstract automaton $BCTSS_\alpha(h)$. Thus, we can now make precise our intended interpretation of the abstract state variables, by providing the abstraction relation S from *states*$(BCTSS)$ to *states*$(BCTSS_\alpha(h))$.

Definition 5.17 *$S(s, u)$ iff*

1. $u.np = g(num(s.t^{h-1}_{max}))$
2. $num(s.t^{h-1}_{max}) \leq n{-}h{+}1 \;\Rightarrow$

$$1. \quad u.S = \{(v, w) \mid \exists i : s.view^h_i = (v, w)\}$$

$$2. \quad u.n_v = g(num(s.t^{h-1}_{max}, v))$$

In the following we explain the details behind the information preserved in abstract state variables np and n.

For any process i, the $snap_i$ action of $BCTSS$ is a potential view-changing action. Suppose $(s, snap_i, s')$ is a step of $BCTSS$ and $s.op_i = label$, $s.t_{max} \neq -$, and $i \neq s.i_{max}$. As a result of this action the nt_i-label changes, implying that $view^h_i[2]$ may change. Suppose that u is a state of $BCTSS_\alpha(h)$ corresponding to s. We want $BCTSS_\alpha(h)$ to match the concrete $snap_i$ action by a corresponding abstract action from u to some state u', such that s' and u' correspond by S. Suppose $num(s.t^{h-1}_{max}) \leq n{-}h{+}1$. No t-labels change as a result of the $snap_i$ action, so $s'.t_{max} = s.t_{max}$ and $num(s'.t^{h-1}_{max}) \leq n{-}h{+}1$. Thus, $s'.view^h_i$ must be an element of $u'.V$. If the abstract action non-deterministically adds some element of $(\mathcal{A} \cup \{0\})^2$ to the set $u.V$, it will obviously allow $s'.view^h_i \in u'.V$. However, this strategy will make $BCTSS_\alpha$ too abstract to satisfy the abstract property of interest. In order to make the abstract action more precise we will preserve information, in u, from the concrete state s, used to determine the new value of nt_i. We are interested in information determinining the new $view^h_i$; i.e. information determining whether $s'.nt^{h-1}_i = s'.t^{h-1}_{max}$ and if so, also information determining the value of $s'.nt_i[h]$.

Assume that $full_i(h') = false$ in s for all $h' \leq h-1$. Then $s'.nt^{h-1}_i = s'.t^{h-1}_{max}$, and the value of $full_i(h)$ in s determines the value of $s'.nt_i[h]$. If $full_i(h) = true$ then $s'.nt_i[h] = next(s.t_{max}[h])$, and if $full_i(h) = false$ then $s'.nt_i[h] = s.t_{max}[h]$. We will preserve in u the information determining $full_i(h)$ in s.

From definition we have that $full_i(h) = true$ if $num_i(s.t^h_{max}) \geq n-h$, and $full_i(h) = false$, otherwise. From the definition of num_i, $num_i(s.t^h_{max})$

$\geq n-h$ if either $num(s.t^h_{max}) > n-h$, or $num(s.t^h_{max}) = n-h$ and $s.t^h_i \neq s.t^h_{max}$. Furthermore, $num_i(s.t^h_{max}) < n-h$ if either $num(s.t^h_{max}) < n-h$, or $num(s.t^h_{max}) = n-h$ and $s.t^h_i = s.t^h_{max}$. We already preserve in $u.V$ the value of $s.view^h_i$, and we will see later that the value of $s.t_{max}[h]$ is preserved in $u.V$ as well. We therefore preserve enough information to determine whether $s.t^h_i = s.t^h_{max}$, holding if $s.view^h_i[1] = s.t_{max}[h]$. Therefore, in order to preserve in u the information determining $full_i(h)$ in s we only need, in addition, to preserve whether $num(s.t^h_{max}) < n-h$, $num(s.t^h_{max}) = n-h$, or $num(s.t^h_{max}) > n-h$. From the abstract property of interest, we will see, that we also need to preserve, for any $v \in \mathcal{A}$, whether $num(s.t^{h-1}_{max}, v) = 1$. If $s.t_{max} \neq -$, then by Definition 5.9, $num(s.t^h_{max}) = num(s.t^{h-1}_{max}, s.t_{max}[h])$ $= num(s.t^{h-1}_{max}, v)$ for some $v \in \mathcal{A}$. We therefore just preserve, for any $v \in \mathcal{A}$, which of the following ranges of naturals, 0, 1, $(1, n-h)$, $n-h$, or $(n-h, n]$ includes the value of $num(s.t^{h-1}_{max}, v)$.

In s, $\sum_v num(s.t^{h-1}_{max}, v) = num(s.t^{h-1}_{max})$ and the value of $num(s.t^{h-1}_{max})$ determines whether $full_i(h-1)$ is *true* in s. If $full_i(h-1)$ is *true* then $s'.nt^{h-1}_i \succ s'.t^{h-1}_{max}$. We know that $full_i(h-1) = true$ if $num_i(s.t^{h-1}_{max}) \geq n-h+1$. Furthermore, $num_i(s.t^{h-1}_{max}) \geq n-h+1$ if either $num(s.t^{h-1}_{max}) > n-h+1$, or $num(s.t^{h-1}_{max}) = n-h+1$ and $s.t^{h-1}_i \neq s.t^{h-1}_{max}$. We will preserve in u which of the ranges $[0, n-h+1)$, $n-h+1$, or $(n-h+1, n]$ includes the value of $num(s.t^{h-1}_{max})$. Notice, that we do not preserve information about the value of $full_i(h')$ for any $h' < h-1$. We add non-determinism to the abstract action allowing it to "assume" any value of $full_i(h')$ for $h' < h-1$.

The above discussion has motivated the particular value ranges preserved by the abstract variables *np* and *n*.

5.5.2 The Abstract Property

Having defined the abstract state space, we can now define the abstract path property $\varphi_\alpha(h)$, and show that $\varphi_\alpha(h)$ "implies" the concrete path property $\varphi(h)$ as required in condition (5.4), i.e. $S^{-1}(paths\,(\varphi_\alpha(h))) \subseteq paths\,(\varphi(h))$.

Definition 5.18 *For any state u of $BCTSS_\alpha(h)$, $\varphi_\alpha(h,u) = true$ iff for all distinct $x, y, z \in u.V$ and for all $i, j, k \in \{1, 2\}$,*

$$\{x_i, y_j, z_k\} \neq \{3,4,5\} \;\wedge\; \{x_1, x_2, y_i\} = \{3,4,5\} \Rightarrow n_{x_1} = one$$

where one = $\mathbf{n{-}h}$ if $h = n-1$ and one = $\mathbf{1}$ otherwise. Let $\varphi_\alpha(h)$ be the path safety property with $states\,(\varphi_\alpha(h)) = states\,(BCTSS_\alpha(h))$ and with $paths\,(\varphi_\alpha(h))$ the set of sequences of states u such that $\varphi_\alpha(h,u)$ holds.

Consider any pair of concrete and abstract states s and u such that $S(s,u)$, it its our intention that $\varphi_\alpha(h,u)$ implies $\varphi(h,s)$. Intuitively, $\varphi_\alpha(h,u)$ states as follows. First, there must not be three distinct views in $u.V$, from which we can generate the set $\{3,4,5\}$ by choosing either the first or the second view element from each view. This will imply that in s, no three distinct processes *with distinct views* can be used to construct a choice vector violating $\varphi(h,s)$. However, we do not know whether three distinct processes *with overlapping views* can construct a violating choice vector. This may occur if two processes in state s have the same view, with both view elements in $\{3,4,5\}$ and with the first element different from the second. In this case $num(s.t^{h-1}_{max}, v) \neq 1$, where v denotes the first view element. The second part of $\varphi_\alpha(h,u)$ implies that in s, three processes with overlapping views cannot construct a violating choice vector.

Notice that $\varphi_\alpha(h)$ is equivalent to the conjunction of the two nonparameterized properties obtained from the cases $h = n-1$ and $h \neq n-1$. Similarly, we shall see, that the abstract automaton $BCTSS_\alpha(h)$ describes exactly two distinct nonparametrized automata, namely for the cases of $h = 1$ and $h \neq 1$. Thus, our abstract verification problem consisting of showing, *paths* $(BCTSS_\alpha(h)) \subseteq$ *paths* $(\varphi_\alpha(h))$, gives rise to only four nonparametrized problem instances.

Lemma 5.8 $S^{-1}(\mathit{paths}\,(\varphi_\alpha(h))) \subseteq \mathit{paths}\,(\varphi(h))$.

Proof. We show that if $S(s,u)$ and $\varphi_\alpha(h,u)$ holds then $\varphi(h,s)$ holds. The Lemma then follows immediately. Assume for the sake of contradiction that $\varphi(h,s)$ does not hold. Then there exists a choice vector $(\ell_1 \ldots \ell_n)$ and distinct indexes i,j,k such that,

$$\begin{aligned}(*)\quad & s.\ell_i^{h-1} = s.\ell_j^{h-1} = s.\ell_k^{h-1} = s.t^{h-1}_{max} \;\wedge \\ & \{s.\ell_i[h], s.\ell_j[h], s.\ell_k[h]\} = \{3,4,5\} \;\wedge \\ & num(s.t^{h-1}_{max}) \leq n-h+1\end{aligned}$$

Suppose that ℓ_i, ℓ_j, and ℓ_k all denote the same type of label (t or nt). Let $c = 1$ if the type is t and let $c = 2$ if the type is nt. From $(*)$ and Lemma 5.7 we have that $s.view^h_i[c] = s.\ell_i[h]$, $s.view^h_j[c] = s.\ell_j[h]$, and $s.view^h_k[c] = s.\ell_k[h]$. Hence, $\{s.view^h_i[c], s.view^h_j[c], s.view^h_k[c]\} = \{3,4,5\}$. This implies that $s.view^h_i$, $s.view^h_j$ and $s.view^h_k$ are all distinct, and from S there exists distinct $x,y,z \in u.V$ such that $x = s.view^h_i$, $y = s.view^h_j$, and $z = s.view^h_k$. Hence, $\{x_c, y_c, z_c\} = \{3,4,5\}$ which contradicts $\varphi_\alpha(h,u)$.

Now, suppose that ℓ_i and ℓ_j denote the same type of label, different from the type denoted by ℓ_k. Let $c = 1$ if ℓ_i and ℓ_j denote label type t and let $c = 2$ if they denote type nt. Let $\overline{c} = 1$ if $c = 2$ and $\overline{c} = 2$ if $c = 1$. From $(*)$ and Lemma 5.7 we have that $\{s.view^h_i[c], s.view^h_j[c], s.view^h_k[\overline{c}]\} = \{3,4,5\}$. This implies that $s.view^h_i \neq s.view^h_j$. Suppose that $s.view^h_k \neq s.view^h_i$ and $s.view^h_k \neq s.view^h_j$. Then from S there exists distinct $x,y,z \in u.V$ such $x = s.view^h_i$, $y = s.view^h_j$, and $z = s.view^h_k$. Hence, $\{x_c, y_c, z_{\overline{c}}\} = \{3,4,5\}$ which contradicts $\varphi_\alpha(h,u)$. Therefore, suppose that $s.view_k^h = s.view_i^h$ or $s.view_k^h = s.view_j^h$. We assume without loss of generality the former case. We know that $s.view_i^h[c]$, $s.view_k^h[\overline{c}] \in \{3,4,5\}$, and since $\{c,\overline{c}\} = \{1,2\}$ and $s.view_{i,1}^h = s.view_{k,1}^h$ we get from Lemma 5.7 that $s.t_i^{h-1} = s.t_k^{h-1} = s.t_{max}^{h-1}$ and $s.t_i[h] = s.t_k[h]$. Let $v = s.t_i[h] = s.t_k[h]$. Then $num(s.t_{max}^{h-1}, v) \neq 1$. Now, from S there exists distinct $x,y \in u.V$ such that $x = s.view_i^h = s.view_k^h$ and $y = s.view_j^h$. Hence, $\{x_c, x_{\overline{c}}, y_c\} = \{3,4,5\}$. Furthermore, since $x_1 = v$ we get from S that $u.n_{x_1} \neq one$. This however contradicts $\varphi_\alpha(h,u)$. ■

5.5.3 The Abstract Automaton

We first define a set of operators used in the actions of the abstract automaton $BCTSS_\alpha(h)$. These operators can all be seen as abstract counterparts of corresponding concrete operators. We show for each abstract operator, that it preserves (is homomorphic w.r.t.) its concrete counterpart in a sense to be made precise later.

The following two definitions provide abstract successor and predecessor functions on elements of $\mathcal{N}$. For each abstract function we provide a lemma stating that the function preserves its concrete counterpart, defined on the concrete domain of naturals. Due to the abstraction of information, the abstract functions return sets of elements from $\mathcal{N}$. In the following we still assume that $one = \mathbf{n{-}h}$ if $h = n-1$ and $one = \mathbf{1}$ otherwise.

Definition 5.19 *For $x \in \mathcal{N}$,*

$$x\mathbf{+1} = \begin{cases} \{\mathbf{1}\} & \text{if } x = \mathbf{0} \wedge one = \mathbf{1} \\ \{\mathbf{n{-}h}\} & \text{if } x = \mathbf{0} \wedge one = \mathbf{n{-}h} \\ \{(\mathbf{1}, \mathbf{n{-}h}),\ \mathbf{n{-}h}\} & \text{if } x \in \{\mathbf{1},\ (\mathbf{1}, \mathbf{n{-}h})\} \\ \{\mathbf{n{-}h{+}1}\} & \text{if } x = \mathbf{n{-}h} \\ \{(\mathbf{n{-}h{+}1}, \mathbf{n}]\} & \text{if } x \in \{\mathbf{n{-}h{+}1},\ (\mathbf{n{-}h{+}1}, \mathbf{n}]\} \end{cases}$$

Lemma 5.9 *For all $x \in \{0, \ldots, n-1\}$, $g(x+1) \in g(x)\mathbf{+1}$.*

Proof. By definition of $\mathbf{+1}$ and g. ■

Definition 5.20 *For $x \in \mathcal{N}$,*

$$x\mathbf{-1} = \begin{cases} \{\mathbf{0}\} & \text{if } x = \mathbf{0} \vee x = \mathbf{1} \vee \\ & \quad x = \mathbf{n{-}h} \wedge one = \mathbf{n{-}h} \\ \{\mathbf{1},\ (\mathbf{1}, \mathbf{n{-}h})\} & \text{if } x = (\mathbf{1}, \mathbf{n{-}h}) \vee \\ & \quad x = \mathbf{n{-}h} \wedge one = \mathbf{1} \\ \{\mathbf{n{-}h}\} & \text{if } x = \mathbf{n{-}h{+}1} \\ \{\mathbf{n{-}h{+}1},\ (\mathbf{n{-}h{+}1}, \mathbf{n}]\} & \text{if } x = (\mathbf{n{-}h{+}1}, \mathbf{n}] \end{cases}$$

Lemma 5.10 *For all $x \in \{1, \ldots, n\}$, $g(x-1) \in g(x)\mathbf{-1}$.*

Proof. By definition of $\mathbf{-1}$ and g. ■

The following two definitions provide predicates on the abstract state variables n, np, and V describing when a valuation of these three variables is *valid*. The validity predicates preserve information in an abstract state about the relationship, in a corresponding concrete state, between the values of $num(t^{h-1}_{max})$ and $num(t^{h-1}_{max}, v)$, for any v, and between these *num*-values and the timestamp values.

Definition 5.21 *valid*(n, np) *iff for all* $v \in \mathcal{A}$,

$$np = \mathbf{n{-}h{+}1} \Rightarrow (n_v = \mathbf{n{-}h} \Leftrightarrow \bigvee_{w \neq v} (n_w = one \wedge \bigwedge_{u \neq v, u \neq w} n_u = \mathbf{0}))$$

The intuition behind the predicate $valid(n, np)$ is as follows. Consider a situation in which concrete processes may potentially move into the cycle defined by the set of labels agreeing with t^{h-1}_{max} and having their h'digit in $\{3, 4, 5\}$. Such movement can only occur if a process picks a new nt label directly into the cycle. This can only occur if $num(t^{h-1}_{max}) = n{-}h{+}1$ and if the process picking the new nt-value agrees with t^{h-1}_{max}. Moreover, if $num(t^{h-1}_{max}) = n{-}h{+}1$ then $num(t^{h-1}_{max}, v) = n{-}h$ if and only if there exists exactly one $w \neq v$ such that $num(t^{h-1}_{max}, w) = 1$ and $num(t^{h-1}_{max}, u) = 0$ for all other u. Preserving this information in abstract states will be used to ensure that old elements of the abstract view set V must eventually be removed during any continuous addition of new elements.

Lemma 5.11 *If* $S(s, u)$ *and* $num(s.t^{h-1}_{max}) \leq n{-}h{+}1$ *then* $valid(u.n, u.np)$.

Proof. Consider the case that $h \neq n{-}1$. Then $one = \mathbf{1}$. The case $h = n{-}1$ is completely analogous. Let v be any element of $\mathcal{A}$. Suppose $u.np = \mathbf{n{-}h{+}1}$. From S, $num(s.t^{h-1}_{max}) = n{-}h{+}1$. Suppose $u.n_v = \mathbf{n{-}h}$. From S, $num(s.t^{h-1}_{max}, v) = n{-}h$ and from the definition of num, there exists $w \neq v$ such that $num(s.t^{h-1}_{max}, w) = 1$ and for each $z \neq v$, $z \neq w$, $num(s.t^{h-1}_{max}, z) = 0$. Hence from S, $u.n_w = \mathbf{1}$ and $u.n_z = \mathbf{0}$ proving the $\Rightarrow$ direction of the biimplication. Now, suppose there exists $w \neq v$ such that $u.n_w = \mathbf{1}$ and for each $z \neq v$, $z \neq w$, $u.n_z = \mathbf{0}$. From S, $num(s.t^{h-1}_{max}, w) = 1$ and $num(s.t^{h-1}_{max}, z) = 0$, and by definition of num, $num(s.t^{h-1}_{max}, v) = n{-}h$. Now, from S, $u.n_v = \mathbf{n{-}h}$ proving the $\Leftarrow$ direction. ■

Definition 5.22 *valid*(V, n, np) *iff valid*(n, np) *and for all* $v \in \mathcal{A}$,

1. $\exists x \in V \; x_1 = v \;\Leftrightarrow\; n_v \neq \mathbf{0}$

2. $\exists x, y \in V \; x_1 = y_1 = v \wedge x_2 \neq y_2 \;\Rightarrow\; n_v \neq one$

The intuition behind 1 and 2 in the definition of $valid(V, n, np)$ is rather obvious. Part 1 preserves the fact that in any concrete state, if some process i has $t_i^{h-1} = t_{max}^{h-1}$ and $t_i[h] = v$, i.e. $view_i^h[1] = v$, then $num(t_{max}^{h-1}, v) \neq 0$. Part 2 preserves the fact that, if two distinct processes i and j have $t_i^h = t_j^h = t_{max}^h$ and $nt_i^h \neq nt_j^h$, i.e. $view_i^h[1] = view_j^h[1]$ and $view_i^h[2] \neq view_j^h[2]$, and if $view_i^h[1] = view_j^h[1] = v$, then $num(t_{max}^{h-1}, v) \neq 1$.

Lemma 5.12 *If $S(s,u)$ and $num(s.t_{max}^{h-1}) \leq n{-}h{+}1$ then $valid(u.V, u.n, u.np)$.*

Proof. Consider the case that $h \neq n{-}1$. Then $one = \mathbf{1}$. The case $h = n{-}1$ is completely analogous. From Lemma 5.11 we have that $valid(u.n, u.np)$ holds so we consider the two additional requirements. Let v be any element of $\mathcal{A}$. Consider 1. Suppose $x \in u.V$ and $x_1 = v$. From S there exists i such that $s.t_i^{h-1} = s.t_{max}^{h-1}$ and $s.t_i[h] = v$. Now, from the definition of num, $num(s.t_{max}^{h-1}, v) \neq 0$ and from S and g, $u.n_v \neq \mathbf{0}$. Suppose $u.n_v \neq \mathbf{0}$. From S, $num(s.t_{max}^{h-1}, v) \neq 0$ and hence there exists i such that $s.t_i^{h-1} = s.t_{max}^{h-1}$ and $s.t_i[h] = v$. Hence from S there exists $x \in u.V$ such that $x_1 = v$.

Consider 2. Suppose $x, y \in u.V$, $x_1 = y_1 = v$ and $x_2 \neq y_2$. Then from S there exists i, j such that $s.t_i^{h-1} = s.t_j^{h-1} = s.t_{max}^{h-1}$ and $s.t_i[h] = s.t_j[h] = v$ and $s.nt_i^h \neq s.nt_j^h$. Hence $i \neq j$ and from the definition of num, $num(s.t_{max}^{h-1}, v) \neq 1$. Now, from S, $u.n_v \neq \mathbf{1}$. ■

The following definition and corresponding preservation lemma shows, that enough information is preserved in abstract states to deduce the value of $t_{max}[h]$ in corresponding concrete states where $t_{max} \neq -$.

Definition 5.23

$$max = \begin{cases} max_{\prec_{\mathcal{A}}}\{x_1 \mid x \in V \wedge x_1 \neq 0\} & \textit{if } \{3,4,5\} \not\subseteq \\ & \quad \{x_1 \mid x \in V \wedge x_1 \neq 0\} \\ - & \textit{otherwise} \end{cases}$$

Lemma 5.13 *If $S(s,u)$, $s.t_{max} \neq -$, and $num(s.t_{max}^{h-1}) \leq n-h+1$ then $u.max = s.t_{max}[h]$.*

Proof. From the definition of $s.t_{max}$ it follows that for any i, if $s.t_i^{h-1} = s.t_{max}^{h-1}$ then $s.t_i[h] \preceq_{\mathcal{A}} s.t_{max}[h]$. Hence $s.t_{max}[h] = max_{\prec_{\mathcal{A}}} \{s.t_i[h] \mid s.t_i^{h-1} = s.t_{max}^{h-1}\}$. From S we have that for any $v \in \mathcal{A}$ there exists i such that $s.t_i^{h-1} = s.t_{max}^{h-1}$ and $s.t_i[h] = v$ iff there exists $x \in u.V$ such that $x_1 = v$. Now, since $s.t_{max} \neq -$ it follows directly from the definition of $u.max$ that $u.max = s.t_{max}[h]$. ■

In the following we assume that $max \neq -$. We define abstract functions *full-level* and *newlabel*. The *full-level* function takes an element of V and returns a subset of abstract values from a set $\{\mathbf{smlh}, \mathbf{h}, \mathbf{lrgh}\}$. The *newlabel* function takes a view x from V and an abstract element in $\{\mathbf{smlh}, \mathbf{h}, \mathbf{lrgh}\}$ and returns a new view.

The idea behind the functions is as follows. Suppose in any state of *BCTSS*, that process k determines a new nt_k-value. It does so by performing a $snap_k$-action with $op_k = label$. If $t_{max} \neq -$ and $k \neq i_{max}$, then k uses function $full_k$ to determine the h' such that the new nt_k will be equal to $next\text{-}label(t_{max}, h')$. The h' will be the minimal h'' such $full_k(h'') = true$. The abstract function $full\text{-}level(x)$ with $x = view_k^h$ is intended to preserves the order of h' with respect to h. Based on the result of $full\text{-}level(x)$, the *newlabel* function returns the new view of k.

Definition 5.24 *For $x \in V$,*

$$full\text{-}level(x) = \begin{cases} \{\mathbf{h}, \mathbf{smlh}\} & if \;\; x_1 \neq 0 \wedge np = \mathbf{n}-\mathbf{h}+\mathbf{1} \wedge \\ & \quad (x_1 = max \wedge n_{max} = \mathbf{n}-\mathbf{h}+\mathbf{1} \vee \\ & \quad\; x_1 \neq max \;\wedge n_{max} = \mathbf{n}-\mathbf{h}) \vee \\ & \\ & \;\; x_1 = 0 \wedge np = \mathbf{n}-\mathbf{h} \wedge n_{max} = \mathbf{n}-\mathbf{h} \\ & \\ \{\mathbf{lrgh}, \mathbf{smlh}\} & otherwise \end{cases}$$

Lemma 5.14 *Let $S(s, u)$, $s.t_{max} \neq -$ and $num(s.t_{max}^{h-1}) \leq n{-}h{+}1$. For any $k \neq s.i_{max}$, if $h' = min\{h'' \mid full_k(h'') = true\}$ then for $x = view_k^h$:*

1. *If $h' < h$ then* $\mathbf{smlh} \in full\text{-}level(x)$.
2. *If $h' = h$ then* $\mathbf{h} \in full\text{-}level(x)$.
3. *If $h' > h$ then* $\mathbf{lrgh} \in full\text{-}level(x)$.

Proof. Let $k \in \{1, \ldots, n\}$ be such that $k \neq s.i_{max}$. Let h' be $min\{h'' \mid full_k(h'') = true\}$ and let $x = view_k^h$.

Suppose $h' < h$. Directly from the definition of *full-level* we have that $\mathbf{smlh} \in full\text{-}level(x)$.

Suppose $h' = h$. From the definition of $full_k$, $num_k(s.t_{max}^h) \geq n-h$ and $num_k(s.t_{max}^{h-1}) < n-h+1$. From the definition of num, $num(s.t_{max}^{h-1}) \geq num(s.t_{max}^h)$ and also, if $num(s.t_{max}^{h-1}) = num(s.t_{max}^h)$ and $s.t_k^{h-1} = s.t_{max}^{h-1}$ then $s.t_k^h = s.t_{max}^h$. Hence, $num_k(s.t_{max}^{h-1}) \geq num_k(s.t_{max}^h)$, and from above $num_k(s.t_{max}^{h-1}) = num_k(s.t_{max}^h) = n-h$. Assume that $s.t_k^{h-1} = s.t_{max}^{h-1}$. Then $num(s.t_{max}^{h-1}) = n{-}h{+}1$. From S, $x_1 \neq 0$ and $u.np = \mathbf{n}-\mathbf{h}+\mathbf{1}$. Now, if $s.t_k[h] = s.t_{max}[h]$ then $num(s.t_{max}^h) = n-h+1$. From Lemma 5.13, $s.t_{max}[h] = u.max$ so by S, $x_1 = u.max$ and $u.n_{max} = \mathbf{n}-\mathbf{h}+\mathbf{1}$. Directly from definition, $\mathbf{h} \in full\text{-}level(x)$. If $s.t_k[h] \neq s.t_{max}[h]$ then $num(s.t_{max}^h) = num_k(s.t_{max}^h) = n-h$. From S, $x_1 \neq u.max$ and $u.n_{max} = \mathbf{n}-\mathbf{h}$, and from definition, $\mathbf{h} \in full\text{-}level(x)$. Assume now that $s.t_k^{h-1} \prec s.t_{max}^{h-1}$. Then $num(s.t_{max}^{h-1}) = num_k(s.t_{max}^{h-1}) = n-h$ and $num(s.t_{max}^h) = num_k(s.t_{max}^h) = n-h$. From S,

$x_1 = 0$, $u.np = \mathbf{n{-}h}$, and $u.n_{max} = \mathbf{n{-}h}$, and again directly by definition, $\mathbf{h} \in$ *full-level*(x).

Suppose $h' > h$. Then $num_k(s.t^h_{max}) < n{-}h$, hence $num(s.t^h_{max}) \leq n{-}h$. Suppose $num(s.t^h_{max}) < n-h$. Since $u.n_{max} = s.t_{max}[h]$, we have from S, $u.n_{max} \notin \{\mathbf{n{-}h}, \mathbf{n{-}h{+}1}\}$ and $\mathbf{lrgh} \in$ *full-level*(x). Suppose $num(s.t^h_{max})$ $= n{-}h$. Then $s.t^h_k = s.t^h_{max}$. From S, $u.n_{max} = \mathbf{n{-}h}$ and $x_1 = u.max$, and directly from definition $\mathbf{lrgh} \in$ *full-level*(x). ■

Definition 5.25 *For $x \in V$ and $l \in \{\mathbf{smlh}, \mathbf{h}, \mathbf{lrgh}\}$*

$$newlabel(x,l) = \begin{cases} (x_1, 0) & \text{if } \ l = \mathbf{smlh} \\ (x_1, next(max)) & \text{if } \ l = \mathbf{h} \\ (x_1, max) & \text{if } \ l = \mathbf{lrgh} \end{cases}$$

We finally define an abstract function *update*, which takes as parameters the set V, an element x of V, and a y in $(\mathcal{A} \cup \{0\})^2$. The function is intended to describe changes to the set V, resulting from abstract counterparts of concrete view-changing actions. The parameter x is intended to describe the view of the process k performing the view-changing action. The action may result in a state where no processes has a view equal to the old view of k. The parameter y is intended to describe the new view of k.

Definition 5.26 *For any $x \in V$, and $y \in (\mathcal{A} \cup \{0\})^2$,*

$$update(V, x, y) = \{V \cup \{y\}, (V \backslash \{x\}) \cup \{y\}\}$$

We now define the abstract automaton $BCTSS_\alpha(h)$. Given a variable x and a set M of values in the range of x. We use the notation $x :\in M$ to denote that x is set nondeterministically to some element of M.

Automaton $BCTSS_\alpha(h)$ has three internal actions, *newlabel*(x), *update*(x), and *updatemax*.

The *newlabel*(x) action is the abstract counterpart of a concrete view-changing $snap_i$ action, within the *label* operation, of a process i with $view^h_i$ $= x$. If the concrete $snap_i$ action is performed in a state where $num(t^{h-1}_{max})$ $> n{-}h{+}1$, then $num(t^{h-1}_{max}) > n{-}h{+}1$ in the resulting state as well. From S, we only require a correspondence between the concrete set of process views and the abstract set V whenever $num(t^{h-1}_{max}) \leq n{-}h{+}1$ in the concrete state. Thus, $np \neq (\mathbf{n{-}h{+}1}, \mathbf{n}]$ is part of the precondition for action *newlabel*(x). If the concrete $snap_i$ action is performed in a state where $view^h_i = (0,0)$ and $num(t^{h-1}_{max}) = n-h+1$, then $view^h_i$ does not change as a result of the action. Thus, $x_1 = 0 \Rightarrow np \neq \mathbf{n{-}h{+}1}$ is part of the precondition for *newlabel*(x). The effect of *newlabel*(x) computes, based on the abstract *newlabel*(x) function, a new abstract view corresponding to the new $view^h_i$. The abstract set S is updated by adding the new view and possibly removing x, based on the *valid* predicate.

```
internal: $newlabel(x)$
    Pre: $x \in V$
         $x_1 = x_2$
         $max \neq -$
         $np \neq (\mathbf{n}-\mathbf{h}+\mathbf{1}, \mathbf{n}]$
         $x_1 = 0 \Rightarrow np \neq \mathbf{n}-\mathbf{h}+\mathbf{1}$
    Eff: $l :\in \mathit{full\text{-}level}(x)$
         $x' := newlabel(x, l)$
         $V' :\in update(V, x, x')$
         if $valid(V', n, np)$ then
            $V := V'$

internal: $updatemax$
    Pre: $true$
    Eff: $V' :\in \mathcal{P}\{(0,0),(0,1)\} \setminus \emptyset$
         $V := V' \cup \{(1,1)\}$
         $n_1 := one$
         $\forall y \neq 1, n_y := \mathbf{0}$
         $np := one$

internal: $update(x)$
    Pre: $x \in V$
         $x_1 \neq x_2 \ \wedge \ x_2 \neq 0$
         $np \neq (\mathbf{n}-\mathbf{h}+\mathbf{1}, \mathbf{n}]$
    Eff: if $x_1 = 0 \ \wedge \ np = \mathbf{n}-\mathbf{h}+\mathbf{1}$ then
           $np := np + 1$
         else
           $x' := (x_2, x_2)$
           $V' :\in update(V, x, x')$
           if $x_1 = 0$ then
              $np' :\in np\mathbf{+1}$
           else
              $np' := np$
              $n'_{x_1} :\in n_{x_1}\mathbf{-1}$
           $n'_{x_2} :\in n_{x_2}\mathbf{+1}$
           $\forall y \notin \{x_1, x_2\} \ n'_y := n_y$
           if $valid(V', n', np')$ then
              $(V, n, np) := (V', n', np')$
```

Figure 5.10: Precondition-Effect code for automaton $BCTSS_\alpha(h)$

The $update(x)$ action is the abstract counterpart of a concrete view-changing $update_i$ action of a process i with $view_i^h = x$ and $nt_i^{h-1} = t_{max}^{h-1}$ i.e. $view_{i,2}^h \neq 0$. Hence, t_{max}^{h-1} does not change as a result of the $update_i$ action. The effect of $update(x)$ computes a new abstract view, corresponding to the new $view_i^h$. Furthermore, it updates the abstract num-count variables np and n. The $valid$ predicate guarantees validity of the updated variables.

The $updatemax$ action is the abstract counterpart of a concrete view-changing $update_i$ action of a process i with $nt_i^{h-1} \succ t_{max}^{h-1}$ i.e. $view_{i,2}^h = 0$. Induction hypothesis (5.1) will prove that the action leads to a state with a new value for t_{max}^{h-1}, this value being the value of nt_i^{h-1}; i.e. the new value of t_i^{h-1}. Furthermore, in this new state, for all processes $j \neq i$, $t_j^{h-1} \prec t_i^{h-1}$. Lemma 5.6 will prove that $t_i[h] = 1$ and hence that $view_i^h = (1,1)$. Lemma 5.6 will also prove that all processes $j \neq i$ will have $view_j^h$ either $(0,0)$ or $(0,1)$ in the new state.

Lemma 5.15 *$BCTSS \leq^{\mathrm{p}} BCTSS_\alpha(h)$ via S*

Proof. The proof is by induction on the length of an execution. If $s_0 \in start(BCTSS)$ then for any i, $s_0.t_i = s_0.nt_i = 1^{n-1}$ and hence $s_0.view_i^h = (1,1)$. Moreover, we have that $num(s_0.t_{max}^{h-1}) = n$, $num(s_0.t_{max}^{h-1}, 1) = 1$, and $num(s_0.t_{max}^{h-1}, v) = 0$ for all $v \in \mathcal{A} - \{1\}$. Suppose $h = 1$. Then $n = n-h+1$ and for any $u_0 \in start(BCTSS_\alpha(h))$, $u_0.V = \{(1,1)\}$, $u.np = \mathbf{n}-\mathbf{h}+\mathbf{1}$, $u.n_1 = \mathbf{n}-\mathbf{h}+\mathbf{1}$, and $u.n_v = \mathbf{0}$ for all $v \in \mathcal{A} - \{1\}$. Hence $S(s_0, u_0)$ in the case

$h = 1$. Suppose $h > 1$. Then $n > n-h+1$ and for $u_0 \in \mathit{start}(\mathit{BCTSS}_\alpha(h))$, the values of abstract variables are as for the case $h = 1$ except from $u_0.\mathit{np}$ and $u_0.n_1$, both of which equals $(\mathbf{n-h+1}, \mathbf{n}]$. Thus $S(s_0, u_0)$ in the case $h > 1$.

Now, let $s \in \mathit{states}(\mathit{BCTSS})$ and let $u \in \mathit{states}(\mathit{BCTSS}_\alpha(h))$ such that $S(s,u)$. We then consider cases based on the type of action π performed by s on a transition $s \xrightarrow{\pi} s'$.

Case 1 $(\pi \in \{\mathit{beginscan}_k,\ \mathit{endscan}(\overline{s})_k,\ \mathit{beginlabel}_k,\ \mathit{endlabel}_k\})$: The corresponding action[1] is $u \xrightarrow{\epsilon} u$. No t-labels or nt-labels change as a result of π so $S(s', u)$.

Case 2 $(\pi = \mathit{snap}_k)$: Suppose $s.t_{max} = -$ or $s.\mathit{op}_k = \mathit{scan}$ or $s.i_{max} = k$. Then no t-labels or nt-labels change as a result of π. We let the corresponding step be $u \xrightarrow{\epsilon} u$ and $S(s', u)$ holds since $S(s,u)$ does so.

For the remainder of the case we assume that $s.t_{max} \neq -$, $s.\mathit{op}_k = \mathit{label}$, and $s.i_{max} \neq k$. From π we have that nt_k is the only label changing and hence all of the state functions of S, except from possibly view_k^h, have the same value in s' as they do in s. More precisely, $s'.t_{max} = s.t_{max}$, $\mathit{num}(s'.t_{max}^{h-1}) = \mathit{num}(s.t_{max}^{h-1})$, $\mathit{num}(s'.t_{max}^{h-1}, v) = \mathit{num}(s.t_{max}^{h-1}, v)$ for all $v \in \mathcal{A}$, and $s'.\mathit{view}_i^h = s.\mathit{view}_i^h$ for all $i \neq k$. Suppose that $\mathit{num}(s.t_{max}^{h-1}) > n–h+1$. Then $\mathit{num}(s'.t_{max}^{h-1}) > n-h+1$. In this case we let the corresponding step be $u \xrightarrow{\epsilon} u$. From $S(s,u)$ we have that $u.\mathit{np} = (\mathbf{n-h+1}, \mathbf{n}]$ and hence $S(s', u)$ holds.

For the remainder of the case we assume that $\mathit{num}(s.t_{max}^{h-1}) \leq n–h+1$. Let h' be such that $s'.nt_k = \mathit{next\text{-}label}(s.t_{max}, h')$. Suppose $s.t_k^{h-1} \prec s.t_{max}^{h-1}$ and $\mathit{num}(s.t_{max}^{h-1}) = n-h+1$. From Lemma 5.4, $s.t_k = s.nt_k$ and hence $s.nt_k^{h-1} \prec s.t_{max}^{h-1}$. Thus $s.\mathit{view}_k^h = (0,0)$. From the definition of $\mathit{next\text{-}label}$, $h' < h$. Hence, $s'.nt_k^{h-1} \succ s.t_{max}^{h-1}$ and so $s'.nt_k^{h-1} \succ s'.t_{max}^{h-1}$. Since no t-labels changes we also have that $s'.t_k^{h-1} \prec s'.t_{max}^{h-1}$. Now from definition, $s'.\mathit{view}_k^h = (0,0) = s.\mathit{view}_k^h$. In this case we again let $u \xrightarrow{\epsilon} u$ be the corresponding step, and $S(s', u)$ holds since $S(s,u)$ does so.

For the remainder of the case assume that $\mathit{num}(s.t_{max}^{h-1}) \neq n–h+1$ or $s.t_k^{h-1} \not\prec s.t_{max}^{h-1}$. Let the corresponding step be $u \xrightarrow{\mathit{newlabel}(x)} u'$ where $x = s.\mathit{view}_k^h$. From $S(s,u)$ we have that $x \in u.V$, $u.\mathit{np} \neq (\mathbf{n-h+1}, \mathbf{n}]$, and $x_1 = 0$ implies $u.\mathit{np} \neq \mathbf{n-h+1}$. From Lemma 5.13, $u.\mathit{max} = s.t_{max}[h]$ $(\neq -)$ and since $s.t_k = s.nt_k$ we have that $x_1 = x_2$. Hence action $\mathit{newlabel}(x)$ is enabled in u. Let u'' be the intermediate state resulting from performing all but the last line in the effect clause of action $\mathit{newlabel}(x)$. Then $u''.\mathit{np} = u.\mathit{np}$, $u''.n_v = u.n_v$ for all $v \in \mathcal{A}$, $u''.V$[2] $\in \mathit{update}(u.V, x, \mathit{newlabel}(x,l))$ where $l \in \mathit{full\text{-}level}(x)$. Let l be the result of resolving the nondeterministic assignment

[1]We assume an always enabled stutter action ϵ s.t. for any state u, $u \xrightarrow{\epsilon} u$

[2]We write $u''.V$ for $u''.V'$.

in action *newlabel*(x) such that, $l = \mathbf{smlh}$ if $h' < h$, $l = \mathbf{h}$ if $h' = h$, and $l = \mathbf{lrgh}$ if $h' > h$. According to Lemma 5.14 we can resolve in this manner. Recall that $s'.t_{max} = s.t_{max}$. If $h' < h$ then $s'.nt_k^{h-1} \succ s'.t_{max}^{h-1}$. Hence $s'.view_k^h[2] = 0$. Let $x' = newlabel(x, l)$. From definition, $x'_2 = 0$. If $h' = h$ then $s'.nt_k^{h-1} = s'.t_{max}^{h-1}$ and $s'.nt_k[h] = next(s'.t_{max}[h])$ and so $s'.view_k^h[2] = next(s'.t_{max}[h])$. Now, since $u.max = s'.t_{max}[h]$ we have from $newlabel(x, l)$, $x'_2 = next(u.max) = next(s'.t_{max}[h])$. If $h' > h$ then $s'.nt_k^{h-1} = s'.t_{max}^{h-1}$ and $s'.nt_k[h] = s'.t_{max}[h]$, and so $s'.view_k^h[2] = s'.t_{max}[h]$. From $newlabel(x, l)$, $x'_2 = u.max = s'.t_{max}[h]$. Since no t-labels change by π, $s'.view_{k,1}^h = s.view_{k,1}^h$, and from $newlabel(x, l)$, $x'_1 = x_1$. This concludes that $x' = s'.view_k^h$. Now, finally we resolve the last nondeterministic assignment in action *newlabel*(x) such that $x \in u''.V$ iff there exists i such that $s'.view_i^h = s.view_k^h$. Then from S, $S(s', u'')$ and hence by Lemma 5.12, $valid(u''.V, u''.n, u''.np)$. Now let $u' = u''$ and we have that $S(s', u')$.

Case 3 $(\pi = update_k)$: We consider cases based on the value of $s.nt_k^{h-1}$.

Case 3.1 $(s.nt_k^{h-1} \prec s.t_{max}^{h-1})$: Let the corresponding action be $u \xrightarrow{\epsilon} u$. From Lemma 5.6 part 2, $s.t_k^{h-1} \neq s.t_{max}^{h-1}$ and hence from the definition of t_{max}^{h-1}, $s.t_k^{h-1} \prec s.t_{max}^{h-1}$. Thus $s.view_k^h = (0, 0)$. From π, $s'.t_k = s.nt_k$ and since t_k is the only label changing we have that $s'.t_{max}^{h-1} = s.t_{max}^{h-1}$. Hence, $s'.view_k^h = (0, 0) = s.view_k^h$. Furthermore, the rest of the state functions of f do not change their value from s to s'. To be precise, $num(s'.t_{max}^{h-1}) = num(s.t_{max}^{h-1})$, $num(s'.t_{max}^{h-1}, v) = num(s.t_{max}^{h-1}, v)$ for all $v \in \mathcal{A}$, and $s'.view_i^h = s.view_i^h$ for all $i \neq k$. Now, $S(s', u)$ holds since $S(s, u)$ does so.

Case 3.2 $(s.nt_k^{h-1} = s.t_{max}^{h-1})$: Since t_k is the only label changing and since $s'.t_k = s.nt_k$, we have that $s'.t_{max}^{h-1} = s.t_{max}^{h-1}$ and $num(s'.t_{max}^{h-1}) \geq num(s.t_{max}^{h-1})$. Also, for all $i \neq k$, $s'.view_i^h = s.view_i^h$. Suppose $num(s.t_{max}^{h-1}) > n{-}h{+}1$. Then $num(s'.t_{max}^{h-1}) > n{-}h{+}1$. In this case we let $u \xrightarrow{\epsilon} u$ be the corresponding step. From $S(s, u)$, $u.np = (\mathbf{n{-}h{+}1}, \mathbf{n}]$ and hence $S(s', u)$.

For the remainder of the case we assume $num(s.t_{max}^{h-1}) \leq n{-}h{+}1$. Now, suppose $s.t_k^{h-1} = s.t_{max}^{h-1}$ and $s.t_k[h] = s.nt_k[h]$. Again we let the corresponding step be $u \xrightarrow{\epsilon} u$. From π, $s'.view_k^h = s.view_k^h$, $num(s'.t_{max}^{h-1}) = num(s.t_{max}^{h-1})$, and $num(s.t_{max}^{h-1}, v) = num(s.t_{max}^{h-1}, v)$ for all $v \in \mathcal{A}$. Therefore $S(s', u)$ holds since $S(s, u)$ does so. For the remainder of the case we therefore also assume that $s.t_k^{h-1} \neq s.t_{max}^{h-1}$ or $s.t_k[h] \neq s.nt_k[h]$. We let the corresponding action be $u \xrightarrow{update(x)} u'$ where $x = s.view_k^h$. From S, $x \in u.V$, $x_2 = s.nt_k[h]$ $(\neq 0)$, and $x_1 \neq x_2$. Also, $u.np \neq (\mathbf{n{-}h{+}1}, \mathbf{n}]$. Hence $update(x)$ is enabled in u. Suppose $s.t_k^{h-1} \neq s.t_{max}^{h-1}$ and $num(s.t_{max}^{h-1}) = n{-}h{+}1$. Then $num(s'.t_{max}^{h-1}) > n{-}h{+}1$. From S, $x_1 = 0$ and $u.np = \mathbf{n{-}h{+}1}$, and from action $update(x)$, $u'.np = (\mathbf{n{-}h{+}1}, \mathbf{n}]$. Hence $S(s', u')$.

Now, suppose $s.t_k^{h-1} = s.t_{max}^{h-1}$ or $num(s.t_{max}^{h-1}) < n-h+1$. From S, $x_1 \neq 0$ or $u.np \neq \mathbf{n-h+1}$. From π we have as follows. If $s.t_k^{h-1} \prec s.t_{max}^{h-1}$ then $num(s'.t_{max}^{h-1}) = num(s.t_{max}^{h-1}) + 1$, $num(s'.t_{max}^{h-1}, x_2) = num(s.t_{max}^{h-1}, x_2) + 1$, and for all $v \neq x_2$, $num(s'.t_{max}^{h-1}, v) = num(s.t_{max}^{h-1}, v)$. If $s.t_k^{h-1} = s.t_{max}^{h-1}$ then $num(s'.t_{max}^{h-1}) = num(s.t_{max}^{h-1})$, $num(s'.t_{max}^{h-1}, x_1) = num(s.t_{max}^{h-1}, x_1) - 1$, $num(s'.t_{max}^{h-1}, x_2) = num(s.t_{max}^{h-1}, x_2)+1$, and for all $v \notin \{x_1, x_2\}$, we have that $num(s'.t_{max}^{h-1}, v) = num(s.t_{max}^{h-1}, v)$. Furthermore, $s'.view_k^h = (x_2, x_2)$. Let u'' be the intermediate state resulting from performing all but the last line in the effect clause of action $update(x)$.

If $s.t_k^{h-1} \prec s.t_{max}^{h-1}$ we have from S, $x_1 = 0$. In this case, $u''.np \in u.np\mathbf{+1}$ and $u''.n_{x_2} \in u.n_{x_2}\mathbf{+1}$. If $s.t_k^{h-1} = s.t_{max}^{h-1}$ we have from S, $x_1 \neq 0$. In this case, $u''.n_{x_1} \in u.n_{x_1}\mathbf{-1}$, $u''.np = u.np$, and $u''.n_{x_2} \in u.n_{x_2}\mathbf{+1}$. In either of the above cases we can, by Lemma 5.9 and Lemma 5.10, resolve the nondeterministic assignments of action $update(x)$ such that $u''.np = g(num(s'.t_{max}^{h-1}))$, $u''.n_{x_1} = g(num(s'.t_{max}^{h-1}, x_1))$, $u''.n_{x_2} = g(num(s'.t_{max}^{h-1}, x_2))$. Furthermore, $u''.n_v = u.n_v$ for all $v \notin \{x_1, x_2\}$ and $u''.V \in update(u.V, x, x')$ where $x' = (x_2, x_2)$. Resolve the *update* assignment such that $x \in u''.V$ iff there exists i such that $s'.view_i^h = s.view_k^h$. Now directly from S, $S(s', u'')$ and by Lemma 5.12, $valid(u''.V, u''.n, u''.np)$. Now, let $u' = u''$ and $S(s', u')$ holds.

Case 3.3 $(s.nt_k^{h-1} \succ s.t_{max}^{h-1})$: In this case let the corresponding step be $u \xrightarrow{updatemax} u'$. This action is always enabled. From Lemma 5.6 part 1, there exists $i \neq k$ such that $s.t_i^{h-1} = s.t_{max}^{h-1}$. Hence for any j such that $j \neq i$, $j \neq k$, $s.t_j^{h-1} \preceq s.t_i^{h-1} \prec s.nt_k^{h-1}$. From induction hypothesis (5.1) this implies that $s.t_j^{h-1} \prec s.nt_k^{h-1}$ and hence for any $i \neq k$, $s.t_i^{h-1} \prec s.nt_k^{h-1}$. Consider any $i \neq k$. Since $s'.t_i^{h-1} = s.t_i^{h-1}$ and $s'.t_k^{h-1} = s.nt_k^{h-1}$ we now have that $s'.t_i^{h-1} \prec s'.t_k^{h-1}$. Hence $s'.t_k^{h-1} = s'.t_{max}^{h-1}$ and $num(s'.t_{max}^{h-1}) = 1$. From Lemma 5.6 part 3, $s.nt_k[h] = 1$ so $num(s'.t_{max}, 1) = 1$, and for all $v \neq 1$, $num(s'.t_{max}^{h-1}, v) = 0$. Also, $s'.view_k^h = (1, 1)$. If there exists $j \neq k$ such that $s.nt_j^{h-1} = s.nt_i^{h-1}$, then from Lemma 5.6 part 3, $s.nt_j[h] = 1$. Thus, since $s.t_j^{h-1} \prec s.nt_i^{h-1}$, we have that $s'.view_j^h = (0, 1)$. Let $u'.V$ be such that $(0, 1) \in u'.V$ iff there exists $j \neq k$ as above. If there exists $j \neq k$ such that $s.nt_j^{h-1} \neq s.nt_i^{h-1}$, then $s'.view_j^h = (0, 0)$. Therefore, let $u'.V$ be such that $(0, 0) \in u'.V$ iff there exists $j \neq k$ as above. Then directly from S, $S(s', u)$. ■

5.6 The SPIN Verification

In this section we describe the automatic verification of our abstract algorithm using the SPIN model checker. Besides verifying the abstract total orderedness property of interest, we also present a few experiments on the abstract model performed using SPIN. These experiments demonstrate how SPIN can be used to automatically provide further insight into the workings of the concrete algorithm - in particular into the workings of the bounded timestamp domain.

5.6.1 The PROMELA Implementation

In the following we present the implementation of the abstract automaton $BCTSS_\alpha(h)$ in the input language PROMELA used by the SPIN model checker. The implementation follows the translation scheme presented in Chapter 2.

Views and Num-counts

In automaton $BCTSS_\alpha(h)$ we use the state variable V to describe sets of process views. Variable V is a set of elements from $(\mathcal{A} \cup \{0\})^2$. In PROMELA there is no built in support for a general set-datatype. We thus implement the variable V using a two-dimensional array as described in Figure 5.11. The PROMELA code declares a two dimensional bit-array where a bit-value

```
typedef a1 {bit nt[6]};
a1 t[6];
```

Figure 5.11: Views

in position (i, j) is indexed `t[i].nt[j]`. The interpretation of `t[i].nt[j]` having value 1 is the obvious one, namely that (i, j) is an element of variable V.

In $BCTSS_\alpha(h)$ we furthermore use the num-count variables np and n. Variable np ranges over elements of the abstract domain of num-counts $\mathcal{N}$ and n is an array ranging over $\mathcal{N}^{\mathcal{A}}$. The abstract domain $\mathcal{N}$ is implemented as a PROMELA message type and we declare variables `np` and `n` to represent np and n, see Figure 5.12.

```
mtype = {zero, one, onh, nh, nho, nhon}

mtype np;
mtype n[6];
```

Figure 5.12: Num-counts

The Automaton

Based on the above declarations, the PROMELA implementation of automaton $BCTSS_\alpha(h)$ is constructed as a single `proctype` declaration with a skeleton as presented in Figure 5.13.

```
proctype BCTSS-alpha()
{
...
do
:: t[i].nt[j]==1 && i==j &&                    /* newlabel(i,j) */
   (i!=0 || np!=nho) && np!=nhon ->  ...

:: t[i].nt[j]==1 && i!=j &&                    /* update(i,j) */
   j!=0 &&  np!=nhon ->  ...

:: true -> ...                                 /* updatemax */
od
...
}
```

Figure 5.13: Process `BCTSS-alpha()`

Each action *newlabel*(x), *update*(x), and *updatemax* from automaton $BCTSS_\alpha(h)$ is described by a separate entry in the `do::od` loop construction of `proctype BCTSS-alpha()`. In Figure 5.13 only the precondition for each action is presented. The first entry in the loop implements action *newlabel*(x) with $x = (i, j)$. The second entry implements action *update*(x) and the third entry implements action *updatemax*. The code in Figure 5.13 relies on a simple scheme for picking a random view, i.e. picking `i` and `j`. The preconditions are obtained by a direct translation from automaton $BCTSS_\alpha(h)$ in Figure 5.10.

In the following we describe the PROMELA implementation of the effect-code for action *newlabel*(x), i.e. the code that is to follow the `->` symbol in the first entry of the `do::od` loop. We assume that $x = (i, j)$. The code consists of a sequence of four code fragments implementing the separate function calls in the effect clause of action *newlabel*(x), see Figure 5.10. The code in Figure 5.14, implements the result of the first line in the effect of action *newlabel*(x). The variable `flevel` implements variable l. The nondeterministic assignment, $:\in$, is implemented by the nondeterminstic `if::fi` structures having all selections guarded by *true* (`1`). The code uses a value `max` which implements the value of *max* from Definition 5.23. The code follows directly from the definition of *full-level*(x), Definition 5.24. The abstract values **smlh**, **h**, and **lrgh** from Definition 5.24 are represented by values `0`, `1`, and `2`, respectively.

```
/* compute flevel */

if
:: ((i!=0 && np==nho) && ((i==max && n[max]==nho)
   || (i!=max && n[max]==nh))) ||
   (i==0 && np==nh && n[max]==nh) ->
   if
   :: 1 -> flevel=0
   :: 1 -> flevel=1
   fi
:: else ->
   if
   :: 1 -> flevel=0
   :: 1 -> flevel=2
   fi
fi;
```

Figure 5.14: Code computing `flevel`

Based on the value of `flevel` the next code fragment, see Figure 5.15, computes the result of the second line in the effect of action *newlabel*(x). Variable `nl` implements the value of $x'[2]$. Note, that $x'[1] = x[1]$ and the value of $x[1]$ is already implemented in `i`. The code follows immediately from Definition 5.25.

```
/* compute newlabel */

if
:: flevel==0 -> nl=0
:: flevel==1 -> if
                :: max<5 -> nl=max+1
                :: max==5 -> nl=3
                fi
:: flevel==2 -> nl=max
fi;
```

Figure 5.15: Code computing `nl`

Now, the final code fragment, see Figure 5.16, implements the last two lines in the effect of action *newlabel*(x). The result of performing these last two lines is simply that the new view x' is added to V and the old view x is either removed from V or kept, based on the value of the *valid* predicate. Assuming that the *valid* predicate holds in the pre-state of action *newlabel*(x), the only way the predicate can be violated in the post-state is if the value of n_{x_1} equals *one*. Note, that this value does not change by the action. Thus in case $n_{x_1} = one$ we must make sure to remove the old view x from V in order not to violate condition 2 in Definition 5.22. If $n_{x_1} \neq one$ the old view x can nondeterministically be removed or kept without causing violations. Having decided whether the old view should be kept or removed, the code then adds the new view to the viewset by setting `t[i].nt[nl]=1`.

```
/* make valid (S,n,np) */

if
:: n[i]==one -> t[i].nt[j]=0
:: else -> if
           :: 1 -> t[i].nt[j]=0
           :: 1 -> t[i].nt[j]=1
           fi
fi;

t[i].nt[nl]=1
```

Figure 5.16: Code guaranteeing validity

The code for actions *update*(x) and *updatemax* is implemented in a similar straightforward manner.

5.6.2 The SPIN Verification

The abstract path property $\varphi_\alpha(h)$ that we wish to verify describes an invariance property. Namely the property that the predicate $\varphi_\alpha(h, u)$ must hold in all reachable states u of $BCTSS_\alpha(h)$. Thus, the property can be stated directly in terms of an LTL invariant. In the PROMELA implementation we maintain a boolean state variable `violate` which implements the above state predicate. Variable `violate` is updated at the end of each entry in the `do::od` construction in `BCTSS-alpha()`, and our goal is then to verify that the `BCTSS-alpha()` satisfies the LTL property `[](violate==0)`.

The updating of variable `violate` is performed as shown in Figure 5.17. The boolean variables `three`, `four`, and `five` are all set if there exists three distinct views spanning the set $\{3, 4, 5\}$. The variables `ethree`, `efour`, and `efive` are all set if there exists three, not necessarily distinct, views spanning $\{3, 4, 5\}$. In this last situation variable `count1` will be nonzero if there exists a view x with $x_1, x_2 \in \{3, 4, 5\}$, $x_1 \neq x_2$, and $n_{x_1} \neq one$. Thus variable `violate` is set to 1 exactly in the case that the state predicate from Definition 5.18 is false.

```
if
:: (three==1 && four==1 && five==1) ||
   (ethree==1 && efour==1 && efive==1 && count1!=0) -> violate=1
:: else -> violate=0
fi;
```

Figure 5.17: Predicate `violate`

We have successfully verified the property `[](violate==0)` for process `BCTSS-alpha()`. As mentioned earlier, the parameter h in automaton $BCTSS_\alpha(h)$ and in the property $\varphi_\alpha(h)$ splits the verification into four cases. The initial values of the num-count variables are defined based on the two cases $h = 1$ and $h \neq 1$, and for each of these cases the code for automaton $BCTSS_\alpha(h)$ and the property $\varphi_\alpha(h)$ distinguishes two cases, based on $h = n - 1$ and $h \neq n - 1$. In the former case variable $one = n-h$ and in the latter case $one = 1$. Our verification has been carried out successfully for all cases. We can thus conclude, that the concrete invariant $\varphi(h)$ holds for the concrete automaton $BCTSS$ for all possible values of h. This concludes the proof of the Total Orderedness theorem, Theorem 5.1.

5.6.3 Further Experiments using SPIN

In the following we describe how SPIN can be used to experiment with the abstract model in order to support our understanding of the bounded timestamp domain used in the concrete model. We first use SPIN to demonstrate the need for values 1 and 2 in the set $\mathcal{A}$ that forms the basis of the bounded timestamp domain. The values 1 and 2 are required to guarantee total orderedness of timestamp values in the concurrent setting where processes pick and update new labels nonatomically. Changing the concrete algorithm such that any process picks and updates a new label in an atomic step removes the need for values 1 and 2. We use SPIN to demonstrate this fact by changing our abstract model correspondingly. We can basically reuse our earlier proofs of property preservation to apply for the slightly changed models. Thus, we actually prove that the changed concrete algorithm has no need for values 1 and 2.

In the following we use SPIN to demonstrate the need for the value 1 in the set $\mathcal{A}$. A similar demonstration can be done for value 2. Consider the concrete automaton $BCTSS$ changed to run on the bounded timestamp domain $(\mathcal{A} - \{1\})^{n-1}$ instead of $\mathcal{A}^{n-1}$. The only changes to $BCTSS$ occur in the definition of function *next-label*, Definition 5.11, and in the initial values of timestamp variables. Definition 5.11 is changed such that $\ell_2[h'] = 2$ for all $h' > h$, and the initial value of all t- and nt variables is changed from

1^{n-1} to 2^{n-1}. Changing the abstract automaton $BCTSS_\alpha(h)$ to "match" the changes in $BCTSS$ is quite simple. The initial value of the abstract view set V is changed from $\{(1,1)\}$ to $\{(2,2)\}$ and in the effect of action *updatemax* all occurrences of the value 1 are replaced by the value 2. Now, upon verifying in SPIN the same abstract total orderedness property as before but now for the slightly changed model, immediately leads to an unsuccessful verification. Furthermore, SPIN provides us with a counter example, which can easily be used to construct an example behavior of the concrete automaton that leads to a violation of the concrete total orderedness property.

The counter example produced by SPIN is illustrated in Figure 5.18. The

Action	State	State nr.
	$V = \{(0,0),(2,2)\}$, $np = \mathbf{n{-}h}$, $n[2] = \mathbf{n{-}h}$	(1)
$newlabel(0,0)$		
	$V = \{(0,0),(0,3),(2,2)\}$, $np = \mathbf{n{-}h}$, $n[2] = \mathbf{n{-}h}$	(2)
$newlabel(0,0)$		
	$V = \{(0,3),(2,2)\}$, $np = \mathbf{n{-}h}$, $n[2] = \mathbf{n{-}h}$	(3)
$update(0,3)$		
	$V = \{(0,3),(2,2),(3,3)\}$, $np = \mathbf{n{-}h{+}1}$, $n[2] = n[3] = \mathbf{n{-}h}$	(4)
$newlabel(2,2)$		
	$V = \{(0,3),(2,4),(3,3)\}$, $np = \mathbf{n{-}h{+}1}$, $n[2] = n[3] = \mathbf{n{-}h}$	(5)
$update(2,4)$		
	$V = \{(0,3),(3,3),(4,4)\}$, $np = \mathbf{n{-}h{+}1}$, $n[3] = n[4] = \mathbf{n{-}h}$	(6)
$newlabel(3,3)$		
	$V = \{(0,3),(3,5),(4,4)\}$, $np = \mathbf{n{-}h{+}1}$, $n[3] = n[4] = \mathbf{n{-}h}$	(7)

Figure 5.18: Counter example produced by SPIN

counter example is in the form of an execution of automaton $BCTSS_\alpha(h)$. The execution starts in state 1 and ends in state 7. For each state, the contents of the abstract variables V, np, and n are shown. For the array n, we only show the components with a content different from $\mathbf{0}$. Thus, in state 1, $n[i] = \mathbf{0}$ for all $i \in \mathcal{A} - \{1,2\}$. The action leading from one state to the next in the execution is shown in between the states. The final state,

state 7, of the execution violates the abstract state property since the view set V contains three distinct views spanning the set $\{3, 4, 5\}$.

Even though our new abstract automaton might be property preserving with respect to the new concrete automaton, we do not know whether the above negative verification result carries over to the concrete automaton. Our preservation conditions, stated in Chapter 2, only guarantees positive results to carry over. However, the counter example from above can easily be used to construct a concrete execution violating the concrete total orderedness property.

Consider the concrete automaton running on timestamp domain $(\mathcal{A} - \{1\})^{n-1}$ in the situation where $n = 3$. The timestamp domain can be represented by the graph obtained from Figure 5.9 by removing all nodes (both at level 1 and level 2) defined by labels including the digit 1. Now, consider the situation in which three processes p_1, p_2, and p_3 have their timestamps variables as follows: $t_1 = nt_1 = 2.2$, $t_2 = nt_2 = 2.2$, and $t_3 = nt_2 = 3.2$. This situation does describe a reachable state in the concrete automaton. Now, consider the case $h = 2$. From the concrete state we have that $t_{max}^{h-1} = t_{max}^{1} = 3$ and thus from Definition 5.14 the set of concrete process views for $h = 2$ is described exactly by the set V in the abstract state 1 of Figure 5.18. Processes p_1 and p_2 have $view_1^h = view_2^h = (0, 0)$ and process p_3 has $view_3^h = (2, 2)$. Moreover, in the concrete state, $num(t_{max}^1) = n - h = 1$, $num(t_{max}^1, 2) = n - h = 1$, and $num(t_{max}^1, v) = 0$ for all $v \neq 2$. The abstract state 1 abstracts these values in variables np and n. Now, in the concrete state suppose process p_1 picks a new label. It will pick its new nt_1 label as 3.3 thus changing $view_1^h$ from $(0, 0)$ to $(0, 3)$. This step is represented in Figure 5.18 as the transition from state 1 to state 2 via abstract action $newlabel(0, 0)$. Continuing the generation of the concrete counter example based on the abstract example leads to the following behavior.

After p_1 has picked its new label it delays its update. Now process p_2 picks a new label choosing $nt_2 = 3.3$ as well. Having picked its new label, p_2 immediately updates its timestamp $t_2 = nt_2 = 3.3$. Now, process p_3 picks a new label $nt_3 = 3.4$ and then updates its timestamp $t_3 = nt_3 = 3.4$. Finally, process p_2 picks a new label resulting in $nt_2 = 3.5$. Now, in this situation variables $nt_1 = 3.3$, $nt_2 = 3.5$, and $t_3 = 3.4$ generates a choice vector violating the concrete total orderedness property. If process p_1 had not delayed its update, the above situation could not have occurred.

Consider the concrete automaton $BCTSS$ changed to run on the timestamp domain $(\mathcal{A} - \{1, 2\})^{n-1}$. Moreover, suppose additional changes that guarantees that processes pick new nt-labels and update corresponding t-labels atomically. A few changes to the abstract automaton $BCTSS_\alpha(h)$ can be made such that the resulting automaton path simulates the new concrete automaton. Moreover, the simulation proof can be done almost entirely by reusing the proof that $BCTSS_\alpha(h)$ path simulates $BCTSS$. We verify using SPIN that the new abstract automaton satisfies the abstract total orderedness property and thus by property preservation the new concrete automaton satisfies the concrete total orderedness property. Thus values 1 and 2 are not needed in the timestamp domain in the setting where processes pick and update new labels in an atomic sequence.

Chapter 6

Fischer's Mutual Exclusion Algorithm

This chapter presents an application of abstraction techniques to prove correctness of a timing-based distributed algorithm. The considered algorithm is Fischer's mutual exclusion algorithm [AL93]. We prove that the parameterized algorithm satisfies the mutual exclusion property, where the parameter is the number of processes running the algorithm. Our proof exploits both *induction*, *compositionality*, and *abstraction* to reduce the unboundedness of the input problem. The proof is within the timed framework presented in Chapter 3. The algorithm and the the mutual exclusion specification are formalized as timed transition systems and our proof relies on showing the existence of a timed ready simulation between these systems. The algorithm is specified as a composition of timed automata and a proof requires reasoning for any number of automata in the composition. Our proof strategy first reduces the required amount of reasoning to involve only three automata. The reduction strategy is based on the construction of an abstract *network invariant* that correctly represents the behaviour of the algorithm regardless of the number of components in its specification. The reduced problem is translated into a reachability problem using the testing approach of Chapter 3 and the resulting problem is directly verifiable using the UPPAAL tool.

6.1 Background and Contributions

For any distributed system it is essential that the applied proof methods supports compositionality. It should be possible to deduce properties of a composed system based on reasoning about its components. Moreover, any method of abstraction must be compositional as well. Meaning that the abstraction method can be applied independently to components of a system "before" they are composed together, as even defining the abstraction relation directly on the full composed system may very well be intractable. In this chapter we show how our timed abstraction framework from Chapter 3 supports compositional reasoning by considering a proof of Fischer's mutual exclusion algorithm parameterized in the number of processes.

Besides compositionality our proof relies on the use of network invariants and induction strategies. We construct an abstract model serving as an invariant for the compositionally defined concrete model of the algorithm. The invariant is shown to correctly simulate the concrete model independently of its number of components. Proving that the abstract model is indeed an invariant is done by induction on the number of components of the concrete model. The seperate steps in the inductive proof only requires reasoning about a small number of processes and therefore model checking can be used to establish the subgoals. We use the testing approach of Chapter 3 to obtain input problems suitable for the UPPAAL model checker.

The idea of reasoning about parameterized systems using network invariants and induction on processes is not a new one. See [WL89b] and [KM89] for some early applications in the untimed setting. The novelty of our strategy is the adaption to a real-time setting and the use of parameterized network invariants.

Fischer's algorithm has been analyzed for a finite number of processes by real-time model checking tools [KLL$^+$97] and theorem proving methods have been applied for the parameterized problem [Luc95]. Methods have also been considered to automatically verify the parameterized algorithm [AJ98].

6.1.1 Chapter Organization

This chapter is organized as follows. In section 6.2 we present Fischer's algorithm formalized in our timed automaton language. We also introduce the method used to specify the mutual exclusion property. Section 6.3 presents the high-level proof strategy that we use to prove the mutual exclusion property. In section 6.4 we present the details of the network invariant and we use it to extract our final proof obligations. Finally, in section 6.5 we show how UPPAAL is used to discharge the obtained proof obligations.

6.2 Fischer's Algorithm

In this section we present Fischer's n-process distributed mutual exclusion algorithm. The algorithm provides asynchronously executing processes with mutually exclusive access to their critical regions based on local timing information. The algorithm usually runs on a shared memory model with a single shared variable. We model the algorithm using our notion of timed automata from Chapter 3 and in this model we have no shared variables. Thus, we use a set of local variables, one per process, to model the single shared variable of the algorithm. Each local variable maintains a copy of the shared variable. Whenever some process wants to update the shared variable, it updates its local copy and synchronizes with all other processes, initiating an identical setting of the local variables in these processes. The synchronization will be atomic, guaranteeing consistency among the local copies of the shared variable. A process wanting to read the shared variable simply reads its local copy.

We assume that processes are indexed by the natural numbers $\mathbf{N}$. Each process i is modelled by a timed automaton P_i as shown in Figure 6.1. Automaton P_i has a local variable $turn_i \in \mathbf{N} \cup \{0\}$, initially having the value 0. This variable models the shared variable as mentioned above. Furthermore, P_i has a clock variable x_i used to guard access to its critical region. Automaton P_i has actions $test_i!$, $set_i!$, $enter_i!$, $exit_i!$, and $fail_i!$, and for each $j \neq i$ actions $exit_j?$ and $set_j?$. Actions $test_i!$, $set_i!$, $enter_i!$, $exit_i!$, and $fail_i!$ constitute the set of actions that P_i performs in its protocol for entering and leaving its critical region. We say that these actions are *controlled* by P_i. Actions $exit_j?$ and $set_j?$ are used by process P_i to observe that some other process P_j sets its local variable $turn_j$. This observation initiates an identical setting in P_i of variable $turn_i$. Actions $exit_j?$ and $set_j?$ are said to be *observed* by P_i. For any action $test_i!$, $set_i!$, $enter_i!$, $exit_i!$, and $fail_i!$, we assume a unique complementary action $test_i?$, $set_i?$, $enter_i?$, $exit_i?$, and $fail_i?$, respectively. The behaviour of P_i is as follows. First note that P_i is always ready to observe actions $set_j?$ and $exit_j?$ for any $j \neq i$. That is, these actions are enabled in any state of P_i. An action $set_j?$ sets $turn_i$ to j and an action $exit_j?$ sets $turn_i$ to 0. Let $\boldsymbol{j}$ denote the set of process indices $\{j \mid j \geq 1 \wedge j \neq i\}$. In Figure 6.1 an edge labelled $(set_{\boldsymbol{j}}?, turn_i := \boldsymbol{j})$ denotes a set of edges, one for each $j \in \boldsymbol{j}$. Any edge in this set is labelled uniquely by a label $(set_j?, turn_i := j)$ for some $j \in \boldsymbol{j}$. Analogously for edges labelled $(exit_{\boldsymbol{j}}?, turn_i := 0)$. We will use this notational simplification in following automaton descriptions as well.

Process P_i starts in location l_0 with all variables set to 0. In l_0, P_i tests the $turn_i$ variable. If $turn_i = 0$, P_i resets clock x_i and enters location l_1. In this location, if P_i delays no longer than one time unit it can set its $turn_i$ variable (and hence all $turn$ variables) to its own index i and then enter location l_2. Here P_i will wait at least two time units before testing the condition $turn_i = i$, which if it holds will grant P_i access to its critical region l_3. Note, that we consider the upper and lower bounds on clock x_i to be constants 1 and 2, respectively. Fischer's algorithm does not demand these exact bounds in order to work correctly. It works for any upper bound a and lower bound b satisfying that $a < b$. We consider the exact bounds 1 and 2 in order to simplify matters. Our focus is an abstraction strategy that reduces the unboundedness of the parameterized algorithm where the parameter is the number of processes running the algorithm rather than the clock bounds. Mutual exclusion will be insured since P_i waits longer in l_2

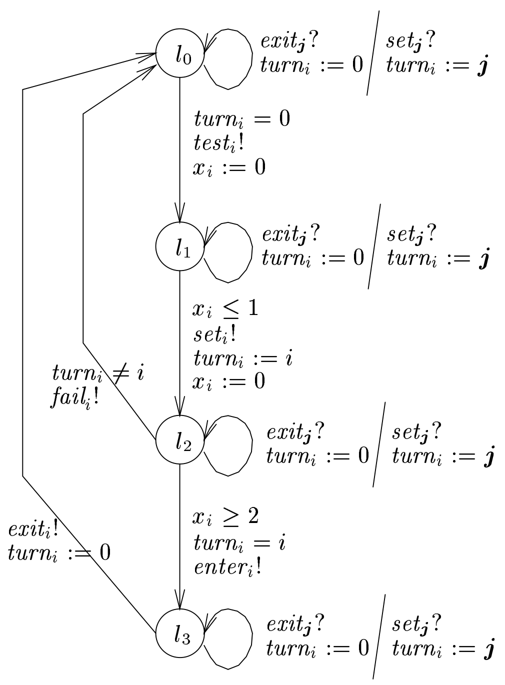

l_0
$exit_{\boldsymbol{j}}?$ $turn_i := 0$
$set_{\boldsymbol{j}}?$ $turn_i := \boldsymbol{j}$
$turn_i = 0$ $test_i!$ $x_i := 0$
l_1
$exit_{\boldsymbol{j}}?$ $turn_i := 0$
$set_{\boldsymbol{j}}?$ $turn_i := \boldsymbol{j}$
$x_i \leq 1$ $set_i!$ $turn_i := i$ $x_i := 0$
$turn_i \neq i$ $fail_i!$
l_2
$exit_{\boldsymbol{j}}?$ $turn_i := 0$
$set_{\boldsymbol{j}}?$ $turn_i := \boldsymbol{j}$
$x_i \geq 2$ $turn_i = i$ $enter_i!$
$exit_i!$ $turn_i := 0$
l_3
$exit_{\boldsymbol{j}}?$ $turn_i := 0$
$set_{\boldsymbol{j}}?$ $turn_i := \boldsymbol{j}$

Figure 6.1: Automaton P_i, $\boldsymbol{j} = \{j \mid j \geq 1 \wedge j \neq i\}$

before testing $turn_i$ than any process will delay in l_1 before setting the *turn* variables. Whenever P_i is in location l_2 and it sees the $turn_i$ variable having a value different from i, it will not be able to enter l_3 and thus it returns to node l_0. Finally, upon leaving l_3 process P_i resets the *turn* variables and enter the initial node l_0.

To describe the parallel composition of automata we define a synchronization function f as follows.

Definition 6.1 *Let f be a synchronization function such that for all $i \in \mathbf{N}$,*

- $f(test_i!, \mathbf{0}) = f(\mathbf{0}, test_i!) = test_i!$
- $f(fail_i!, \mathbf{0}) = f(\mathbf{0}, fail_i!) = fail_i!$
- $f(enter_i!, \mathbf{0}) = f(\mathbf{0}, enter_i!) = enter_i!$
- $f(set_i!, set_i?) = f(set_i?, set_i!) = set_i!$
- $f(set_i?, set_i?) = set_i?$
- $f(exit_i!, exit_i?) = f(exit_i?, exit_i!) = exit_i!$
- $f(exit_i?, exit_i?) = exit_i?$

and f takes value $-$ for all other inputs.

Proposition 6.1 *f is associative and commutative.*

For any $n \in \mathbf{N}$, we now define the composition F_n of processes $P_1, \dots, P_n$ as follows.

Definition 6.2 *For any $n \in \mathbf{N}$ define F_n as, $F_n = P_1 \otimes_f \cdots \otimes_f P_n$.*

Note that $\otimes_f$ is associative and commutative since by Proposition 6.1 f is so. Consider any processes P_i and P_j in the composition F_n. Then P_i can perform actions $set_i!$ and $exit_i!$ only if process P_j synchronizes on actions $set_i?$ and $exit_i?$, respectively. This follows from the fact that $f(set_i!, set_i?)$ $= f(set_i?, set_i!) = set_i!$, and $f(set_i!, \mathbf{0}) = f(\mathbf{0}, set_i!) = -$. Analogously for action $exit_i!$. We thus have atomicity of actions setting the local *turn* variables, ensuring consistency among these local copies. Furthermore, for process P_i actions $set_j?$ and $exit_j?$ are always enabled, ensuring that P_i is always ready to observe an update of the *turn* variables by P_j.

We have defined F_n as an *open* system always ready to observe actions from possible processes in the environment. More precisely, for any $m > n$ the actions $set_m?$ and $exit_m?$ are enabled in any state of F_n. When proving mutual exclusion we are interested in F_n as a *closed* system without any processes in its environment. Therefore, we define a closed version of F_n, where the above actions are no longer enabled. We assume a special automaton *nil* consisting of a single location (the initial location), a single clock, no data variables, and no edges. Thus, the only behaviours of *nil* are delay transitions. We define a synchronization function ρ_n as follows: $\rho_n(a, \mathbf{0}) = a$ for all $a \in \{test_i!, set_i!, enter_i!, fail_i!, exit_i!\}$, $i \leq n$, and ρ_n takes value $-$ for all other inputs. Now, we define the closed version of F_n, denoted $\mathcal{F}_n$, as follows.

Definition 6.3 *Define $\mathcal{F}_n$ as, $\mathcal{F}_n = F_n \otimes_{\rho_n}$ nil.*

6.2.1 The Mutual Exclusion Property

Our goal is to prove the mutual exclusion property for $\mathcal{F}_n$ for all $n \geq 1$. For any n, the property states that no reachable state of $\mathcal{F}_n$ exists in which more than one process (in $\mathcal{F}_n$) is in its critical region. Formally, we will state the property using an abstract specification automaton. For any $n \geq 1$, we will define a specification $\mathcal{M}_n$ that will specify the mutual exclusion condition for $\mathcal{F}_n$. Analogous to $\mathcal{F}_n$, we will define $\mathcal{M}_n$ as the closed version of an open specification M_n. We postpone the precise definition of M_n until section 6.4, but assuming its existence we define as follows.

Definition 6.4 *Define $\mathcal{M}_n$ as, $\mathcal{M}_n = M_n \otimes_{\rho_n}$ nil.*

Our goal is now to prove the following,

$$\mathcal{F}_n \preceq \mathcal{M}_n \quad \text{for all} \quad n \geq 1 \tag{6.1}$$

That is, for any $n \geq 1$, there exists a timed ready simulation from $\mathcal{F}_n$ to $\mathcal{M}_n$ parameterized with the identity action relation *id*. The specification $\mathcal{M}_n$ will be stated over the same actions as those of $\mathcal{F}_n$. Therefore, the requirement of the identity action relation. By definition, proving 6.1 amounts to showing that $F_n \otimes_{\rho_n} nil \preceq M_n \otimes_{\rho_n} nil$, for any $n \geq 1$. Using the compositionality principle, stated as Theorem 3.2 of Chapter 3, it will suffice to show separately that $F_n \preceq M_n$ and $nil \preceq nil$. The latter is trivial since the $\preceq$ relation is reflexive. The compositionality theorem imposes a few technical requirements on processes. It requires M_n and *nil* to be τ-free and the identity action relation *id* must be closed with respect to the synchronization function ρ_n. Neither M_n nor *nil* will have any τ-transitions, and by definition *id* is closed with respect to ρ_n. Thus, proving 6.1 reduces to proving,

$$F_n \preceq M_n \quad \text{for all} \quad n \geq 1 \tag{6.2}$$

The proof of 6.2 will be the topic of the rest of this chapter. Recall, that we still have not precisely defined the specification M_n. We merely assume that it exists and that it correctly specifies the mutual exclusion property.

6.3 The Proof Strategy

In this section we come a few steps closer to the precise definition of the specification M_n. We will define M_n as a composition of a *network invariant* and a *well-formedness* specification, and we will provide the overall proof strategy for showing that this composition timed ready simulates process F_n as required in equation 6.2. The precise definition of the network invariant and the well-formedness specification will be postponed until the next section.

For any $n \geq 1$, we will assume the existence of timed automata I_n and WF_n denoted as the n'th network invariant and the n'th well-formedness specification, respectively. Based on these automata we define the mutual exclusion specification M_n as the *synchronous* composition of I_n and WF_n.

Definition 6.5 *Define M_n as $M_n = I_n \otimes_\sigma WF_n$*

Now, to prove equation 6.2 we can once again apply the compositionality principle. To prove $F_n \preceq M_n$ it suffices to prove separately,

$$F_n \preceq I_n \quad \text{for all} \quad n \geq 1 \tag{6.3}$$

and

$$F_n \preceq WF_n \quad \text{for all} \quad n \geq 1 \tag{6.4}$$

The reasoning is valid due to the following argument. If 6.3 and 6.4 hold then by Theorem 3.2 (Compositionality) we have for any $n \geq 1$, $F_n \otimes_\sigma F_n \preceq$

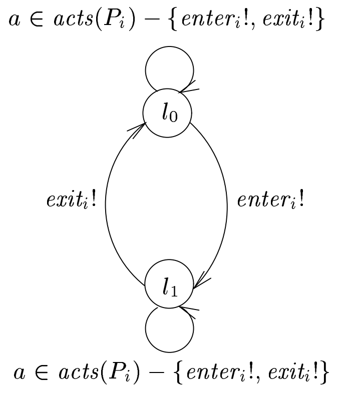

$a \in acts(P_i) - \{enter_i!, exit_i!\}$
l_0
$exit_i!$
$enter_i!$
l_1
$a \in acts(P_i) - \{enter_i!, exit_i!\}$

Figure 6.2: Automaton W_i

$I_n \otimes_\sigma WF_n$. By Theorem 3.1 (Idempotency of Synchronous Composition) we have that $F_n \preceq F_n \otimes_\sigma F_n$. Thus, from Theorem 3.3 (Transitivity) we can conclude that $F_n \preceq I_n \otimes_\sigma WF_n$ i.e. $F_n \preceq M_n$. The additional technical requirements of the involved theorems are all satisfied since none of our processes include neither τ nor urgent actions.

We have now reduced our overall proof goal to the two subgoals of equations 6.3 and 6.4. In the next section we will formally define the network invariant I_n and the well-formedness specification WF_n and we will consider the proofs of equations 6.3 and 6.4.

6.4 The Abstraction

In this section we formally define the components I_n and WF_n of the mutual exclusion specification M_n and we prove that these components satisfy the equations 6.3 and 6.4 from the previous section. The proof of 6.4 will be by a simple application of the compositionality principle. The proof of 6.3 will be using a combination of induction, compositionality, and abstraction. This proof will constitute the main part of the rest of this chapter.

6.4.1 The Well-Formedness Specification

The purpose of the specification WF_n is to specify a certain well-formedness requirement on any process P_i in F_n. The requirement simply says that any behaviour of F_n restricted to the actions $enter_i!$ and $exit_i!$ of P_i is an alternating sequence of these two actions beginning with $enter_i!$.

For any $i \geq 1$ we define an automaton W_i as shown in Figure 6.2. It is obvious that any behaviour of W_i restricted to actions $enter_i!$ and $exit_i!$ is an alternating sequence of these two actions beginning with $enter_i!$. We now define the well-formedness specification as follows.

Definition 6.6 *For any $i \geq 1$, define WF_i as $WF_i = W_1 \otimes_f \cdots \otimes_f W_i$.*

The specification WF_n provides the required well-formedness specification for any process P_i in F_n as follows. First observe that actions $enter_i!$ and $exit_i!$ are not in $acts(W_j)$ for any W_j such that $j \neq i$. Moreover, no such W_j can prevent the execution of any action in W_i. From the definition of f, the only action of W_i that needs synchronization is action $exit_i!$. However, the complementary action $exit_i?$ is always enabled in any W_j, $j \neq i$.

Proving that for all $n \geq 1$, $F_n \preceq WF_n$ (6.4) is simple. Recall that $F_n = P_1 \otimes_f \cdots \otimes_f P_n$ and $WF_n = W_1 \otimes_f \cdots \otimes_f W_n$. Thus, due to Theorem 3.2 (Compositionality) we simply need to show that for every $i \geq 1$, $P_i \preceq W_i$. But this is obvious directly by inspection of Figure 6.1 and Figure 6.2.

6.4.2 The Network Invariant

The overall purpose of the network invariant I_n is to serve as a *weak* mutual exclusion specification for the set of processes in F_n. The idea is that the composition, M_n, of this weak specification, I_n, and the well-formedness specification, WF_n, will serve as the correct mutual exclusion specification. We may say that WF_n is used to strengthen I_n in order to obtain the correct specification. I_n will specify, that for any indices $i, j, k \leq n$, no two $enter_i!$, $enter_j!$ actions can occur without an intervening $exit_k!$ action. This specification is weak in the sense that it does not require that $k = i$. We strengthen I_n to obtain the stronger specification requiring that $k = i$ by composing it synchronously with the well-formedness specification WF_n.

We want to prove that I_n satisfies equation 6.3, that is $F_n \preceq I_n$ for all $n \geq 1$. To prove this we will use induction on n. The strategy will be as follows: Assume that,

$$F_m \preceq I_m \quad \text{for all} \quad 1 \leq m < n \tag{6.5}$$

Prove that,

$$F_1 \preceq I_1 \quad \text{and} \quad I_{n-1} \otimes_f P_n \preceq I_n \quad \text{when } n > 1 \tag{6.6}$$

Then, $F_n \preceq I_n$ for all $n \geq 1$ and equation 6.3 holds. The soundness of this strategy is established as follows. Assume that 6.6 has been proved under the assumption of 6.5. We will see how this can be used to conclude that $F_n \preceq I_n$ for all $n \geq 1$. The case $n = 1$ is trivial. Suppose $n > 1$. By definition we know that $F_n = F_{n-1} \otimes_f P_n$. By induction hypothesis 6.5 we know that $F_{n-1} \preceq I_{n-1}$ and thus by Theorem 3.2 (Compositionality), $F_n \preceq I_{n-1} \otimes_f P_n$. From 6.6 we have that $I_{n-1} \otimes_f P_n \preceq I_n$ and thus by Theorem 3.3 we can conclude that $F_n \preceq I_n$.

Using the term *network invariant* for I_n is motivated by the fact that the automaton satisfies equation 6.6. Intuitively, I_n serves as an abstract representation of all the processes in F_n.

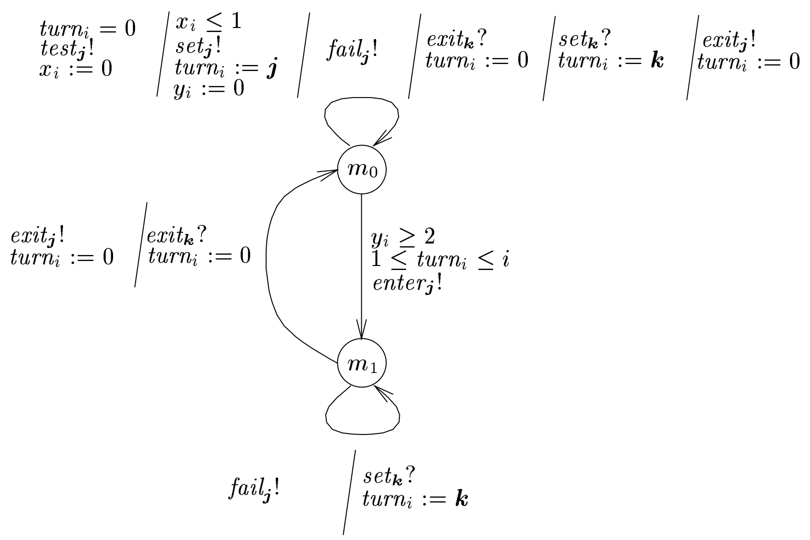


Figure 6.3: Invariant I_i, $\boldsymbol{j} = \{j \mid 1 \leq j \leq i\}$, $\boldsymbol{k} = \{k \mid k > i\}$

We now present the automaton I_i for any $i \geq 1$. The automaton is shown in Figure 6.3. It has actions $test_j!$, $set_j!$, $enter_j!$, $exit_j!$, and $fail_j!$ for all $1 \leq j \leq i$, and $set_k?$, $exit_k?$ for any $k > i$. This is exactly the action interface of F_i. Automaton I_i has a single data variable $turn_i$ and two clock variables x_i and y_i. Variable $turn_i$ ranges over the naturals including 0, and it is used to represent the (unique) value held by the concrete set of *turn* variables in F_i. Clock variable x_i will represent the local clock value of the process in F_i having performed the most recent *test*! action, and the clock y_i will represent the clock value of the process in F_i having performed the most recent *set*! action.

It is obvious directly from Figure 6.3 that automaton I_n specifies the required weak mutual exclusion property for any $n \geq 1$. Recall that this property states that for all $i, j, k \leq n$, no two $enter_i!$, $enter_j!$ can occur without an intervening $exit_k!$ action. If we let *enter*! denote any $enter_i!$ action, $i \leq n$ and *exit*! denote any $exit_i!$ action, $i \leq n$, then the above property simply says that no two *enter*! actions can occur without an intervening *exit*! action. We might say that the property is independent of the indices of actions. If the only purpose of automaton I_n was to specify the above property, a much simpler implementation than the one of Figure 6.3 is possible. However, our inductive strategy for proving that $F_n \preceq I_n$ imposes restrictions on I_n. In particular, we must be able to prove the conditions of equation 6.6 which implies that I_n cannot be *too abstract.*

In the following we make an attempt to create some intuition for automaton I_i as it is defined in Figure 6.3 for any $i \geq 1$. We examine how I_i is intended to represent the behavior of processes in F_i. In the initial state of F_i, as long as the *turn* variables have the value 0, any process in F_i can perform a *test*! action, and thereby reset its local clock to 0 and enter location l_1. In I_i such an action is represented by a *test*! action from m_0 to itself. This action resets clock x_i, thereby letting x_i represent the local clock of the concrete process having performed the most recent *test*! action. In F_i, any process being in location l_1 may perform a *set*! action no later than one time unit from when the process entered l_1. This also implies, that any *set*! action may be performed no later than one time unit from when the most recent process entered l_1. Since the abstract clock x_i preserves the clock value of the most recent process to enter l_1, we have that $x_i \leq 1$ is a safe weakening of the guard for any *set*! action in F_i. Thus, $x_i \leq 1$ serve as guard in I_i for the abstract *set*! action from m_0 to itself. This abstraction may of course allow I_i to perform a $set_j!$ action, for some $j \leq i$, much later than one time unit from the corresponding $test_j!$. However, in I_i all we want to preserve is the fact that any *set*! action occurs no later than one time unit from the most recent *test*! action. Any process in F_i being in location l_2 can enter the critical region l_3 provided it has spent at least two time units in l_2 and provided the *turn* variable holds the index of the process. Due to the overwriting of the *turn* variables on *set*! actions, a process entering l_3 will be the most recent process having performed a *set*! action. In I_i we preserve, in y_i, the local clock of the concrete process having performed the most recent *set*! action. Thus, in I_i an *enter*! action is guarded by the condition $y_i \geq 2$ combined with the condition that the *turn* variable must have some value less than or equal to i.

In the remainder of this chapter we present our proof that $F_n \preceq I_n$ for all $n \geq 1$. We use our inductive proof strategy presented in 6.6 and 6.7. Using the inductive strategy simplifies our proof from a task involving an unbounded number of automata to a task involving only the automata explicitly mentioned in proof obligation 6.7. However, all the automata of 6.6 contains an unbounded number of actions. In the following we construct further abstractions of the automata in 6.6 in order to reduce the number of actions. The resulting automata will allow for an automatic verification of proof obligation 6.7.

Further Abstraction

In the remainder of this section n will denote the induction constant from 6.7. We will use the technique of Section 3.3 to construct finite state abstractions for the automata in 6.7.

Let $\Delta = \{0, 1, 2, 3\}$ and let $h : \mathbf{N} \mapsto \Delta$ be such that:

$$h(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{if } 1 \leq x < n \\ 2 & \text{if } x = n \\ 3 & \text{if } x > n \end{cases}$$

It can easily be seen that (Δ, h) is an abstract data domain for I_{n-1}, I_n, and P_n (If $n = 1$, I_{n-1} is undefined). That is, h preserves, according to the definitions of Section 3.3, the guards and the reset operations of each of these automata.

Assume that actions consist of a *name* part and an *index* part, and that there exists functions *index*() and *name*() extracting the corresponding information in the obvious way. For example $\mathit{name}(\mathit{test}_i!) = \mathit{test}!$ and $\mathit{index}(\mathit{test}_i!) = i$. Let R be the relation from $\mathcal{A} \cup \{\mathbf{0}\}$ to $\mathcal{A} \cup \{\mathbf{0}\}$ such that $(\mathbf{0}, \mathbf{0}) \in R$ and for any actions $a, b \neq \mathbf{0}$, $(a, b) \in R$ iff $\mathit{name}(a) = \mathit{name}(b)$ and $h(\mathit{index}(a)) = \mathit{index}(b)$. We now define sets of abstract actions. We define sets $\Sigma(I_i)$ and $\Sigma(P_i)$ of abstract actions as follows: $\Sigma(I_i) = \{b \mid \exists a \in \mathit{acts}(I_i).\ (a, b) \in R\}$ and $\Sigma(P_i) = \{b \mid \exists a \in \mathit{acts}(P_i).\ (a, b) \in R\}$. Then R is an action relation total on $\mathit{acts}(I_i)$ and $\Sigma(I_i)$ and total on $\mathit{acts}(P_i)$ and $\Sigma(P_i)$. This implies that $(\Sigma(I_i), R)$ and $(\Sigma(P_i), R)$ are abstract action domains for I_i and P_i, respectively. Let $S_1 = ((\Delta, h), (\Sigma(I_{n-1}), R))$, $S_2 = ((\Delta, h), (\Sigma(I_n), R))$, and $S_3 = ((\Delta, h), (\Sigma(P_n), R))$. Then S_1, S_2, and S_3 are abstract domains for I_{n-1}, I_n, and P_n, respectively. Automata $(I_{n-1})_{S_1}$, $(I_n)_{S_2}$, and $(P_n)_{S_3}$ will be our abstractions of I_{n-1}, I_n, and P_n.

In the case $n = 1$, $(I_n)_{S_2}$ will be identical to I_2 with indices of actions ranging over the set $\{2, 3\}$. Also, $(P_n)_{S_3}$ will be identical to P_2 with indices of actions ranging over $\{2, 3\}$ as well. In the case $n > 1$, $(I_{n-1})_{S_1}$ and $(I_n)_{S_2}$ will be identical to I_1 and I_2, respectively, with indices ranging over the set $\{1, 2, 3\}$. Analogously $(P_n)_{S_3}$ will be identical to P_2 with indices over $\{1, 2, 3\}$. In the following we assume that I_1, I_2, and P_2 have index sets as described above.

It can easily be observed that each of the abstract automata are closed under the action relation R. Thus, from Theorem 3.6(3.7) we have that $P_n \preceq^R P_2$ ($P_2 \preceq^{R^{-1}} P_n$) and $I_{n-1} \preceq^R I_1$ ($I_1 \preceq^{R^{-1}} I_{n-1}$). By Theorem 3.2 (Compositionality), and since R (R^{-1}) is closed with respect to f, we have that $P_n \otimes_f I_{n-1} \preceq^R P_2 \otimes_f I_1$ ($P_2 \otimes_f I_1 \preceq^{R^{-1}} P_n \otimes_f I_{n-1}$). We further have, from Theorem 3.7(3.6) that $I_2 \preceq^{R^{-1}} I_n$ ($I_n \preceq^R I_2$). Thus by Theorem 3.5 we have as follows:

$$P_2 \preceq I_2 \ \text{ iff } \ P_n \preceq I_n, \quad \text{for } n = 1 \tag{6.7}$$

and

$$I_1 \otimes_f P_2 \preceq I_2 \ \text{ iff } \ I_{n-1} \otimes_f P_n \preceq I_n, \quad \text{for } n > 1 \tag{6.8}$$

We can thus conclude that our proof obligations stated in equation 6.6 can be replaced by the lefthand sides of equations 6.7 and 6.8. All the automata of the new proof obligations have finite sets of actions. We can now use the testing approach presented in Chapter 3 to verify the lefthand

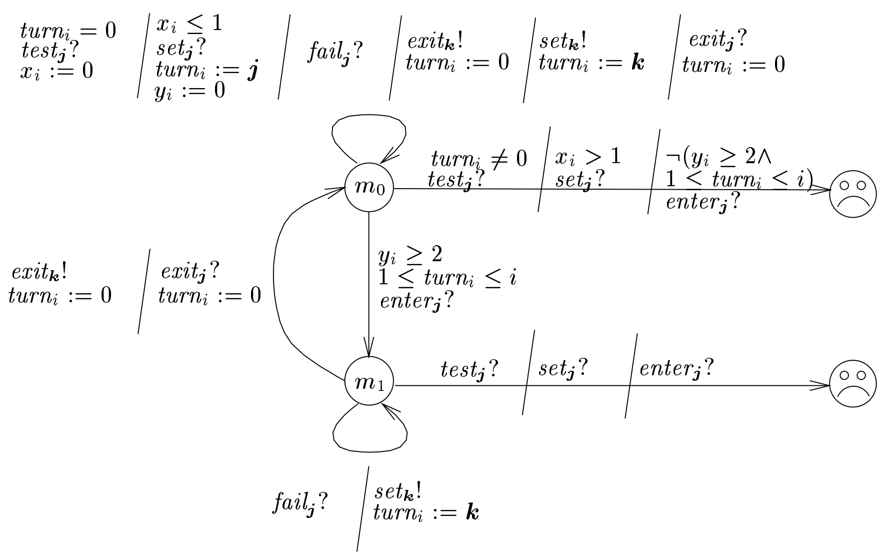


Figure 6.4: Test Automaton T_{I_i}, $\boldsymbol{j} = \{j \mid 1 \leq j \leq i\}$, $\boldsymbol{k} = \{k \mid k > i\}$

sides of equations 6.7 and 6.8. This approach translates the checks for timed ready simulations into reachability questions directly analyzable using the UPPAAL tool. We consider the UPPAAL verification in the next section.

6.5 The UPPAAL Verification

We first consider how the lefthand sides of equations 6.7 and 6.8 are translated into reachability problems. For this we use the testing approach presented in Chapter 3.

Since automaton I_i is τ-free and deterministic, for any i, we have from Theorem 3.9 that,

$$P_2 \textit{ passes the } I_2\textit{-test} \quad \text{iff} \quad P_2 \preceq I_2 \tag{6.9}$$

and

$$I_1 \otimes_f P_2 \textit{ passes the } I_2\textit{-test} \quad \text{iff} \quad I_1 \otimes_f P_2 \preceq I_2 \tag{6.10}$$

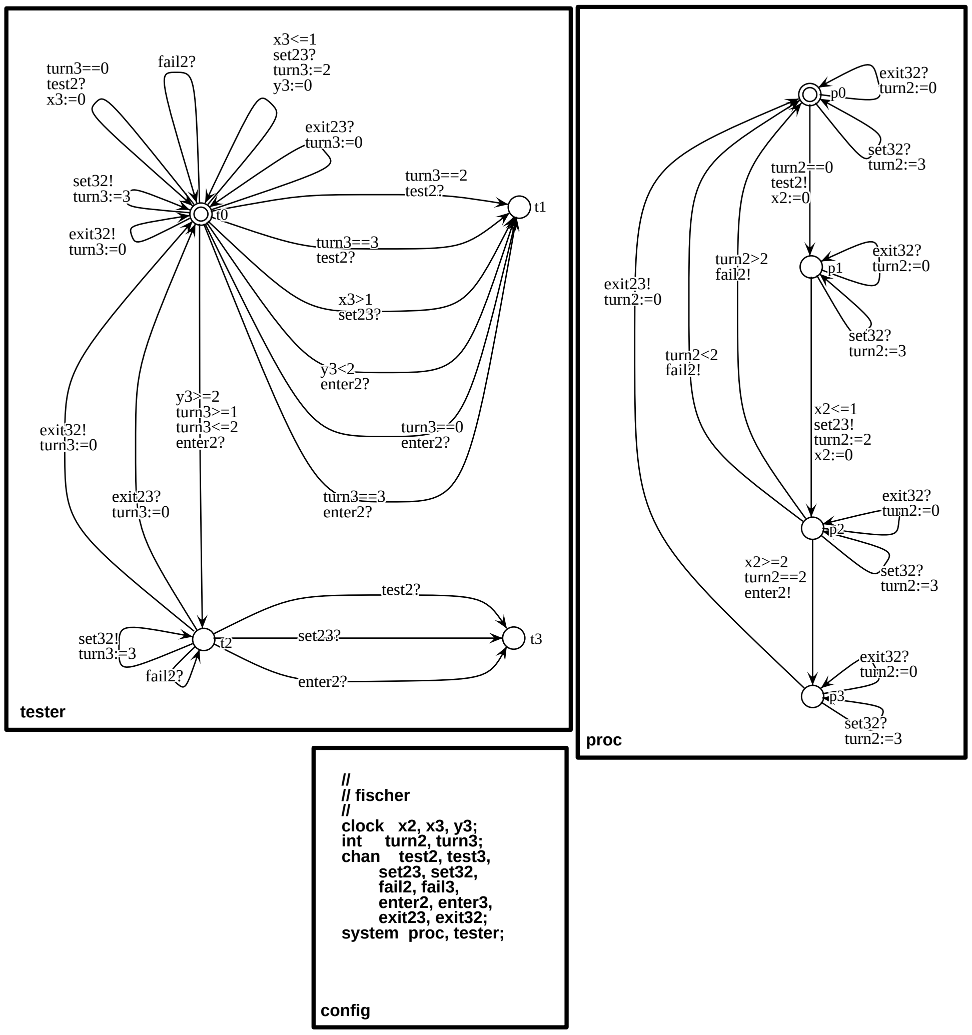
We thus construct the test automaton T_{I_2} for I_2 such that in case $n = 1$ we use the set $\{2, 3\}$ as index domain and in case $n > 1$ we use the set $\{1, 2, 3\}$. We assume that any action $a!(?)$ of I_2 has a unique complementary action $a?(!)$. Let s be the synchronization function defined as follows for any $i \in \{1, 2\}$: $s(test_i!, test_i?) = s(set_i!, set_i?) = s(enter_i!, enter_i?) = s(set_3?, set_3!) = \tau$, and s takes value $-$ for all other inputs. The test automaton T_{I_i} is shown in Figure 6.4. We will verify the lefthand sides of equations 6.9 and 6.10 using the automatic verification tool UPPAAL which allows for reachability analysis upon networks of timed automata. The automata of 6.9 and 6.10 are, with a few modifications, typed directly into the graphical input language provided by the UPPAAL tool. The input for UPPAAL for 6.9 is shown in Figure 6.5 and for 6.10 in Figure 6.6. For simplicity we have denoted I_1 by `invariant`, P_2 by `proc`, and T_{I_2} by `tester`.

The timed automata input language for UPPAAL only allows for binary synchronization between processes in a composition (network). Since all three automata in 6.10 synchronize on actions *set* and *exit*, we have to somehow implement this using only binary synchronization. Fortunately, the input language for UPPAAL allows us to do this using a special form of location called *committed locations.* Consider for example the synchronization consisting of automaton I_1 performing a $set_1!$ action and automata P_2 and T_{I_2} both performing a $set_1?$. In the UPPAAL model, see Figure 6.6, this is implemented as follows. Automaton `invariant` synchronizes in turn with automata `proc` and automata `tester`. First, `invariant` performs a `set12!` action synchronizing with action `set12?` of `proc`. Then `invariant` performs a `set13!` action synchronizing with action `set13?` of `tester`. Atomicity of the synchronization sequence is ensured by labelling the intermediate node `i2` of automaton `invariant` with a prefix `c:`, marking the node as committed. This guarantees that once `invariant` enters location `i2`, the next transition taken in the complete system will be from this location. More precisely, no actions, including time-passage actions, can be taken until `invariant` has left the committed location. Using the same approach, we implement synchronizations on the remaining *set* and *exit* actions as well. No further modifications are made upon translation to UPPAAL model.

The reject nodes of test automaton T_{I_2} correspond to locations (nodes) `t1` and `t3` in `tester`. Thus, we want to verify that none of these locations are reachable in the composition of `invariant`, `proc`, and `tester`. Stated as a property in the logical property language of UPPAAL, this becomes:

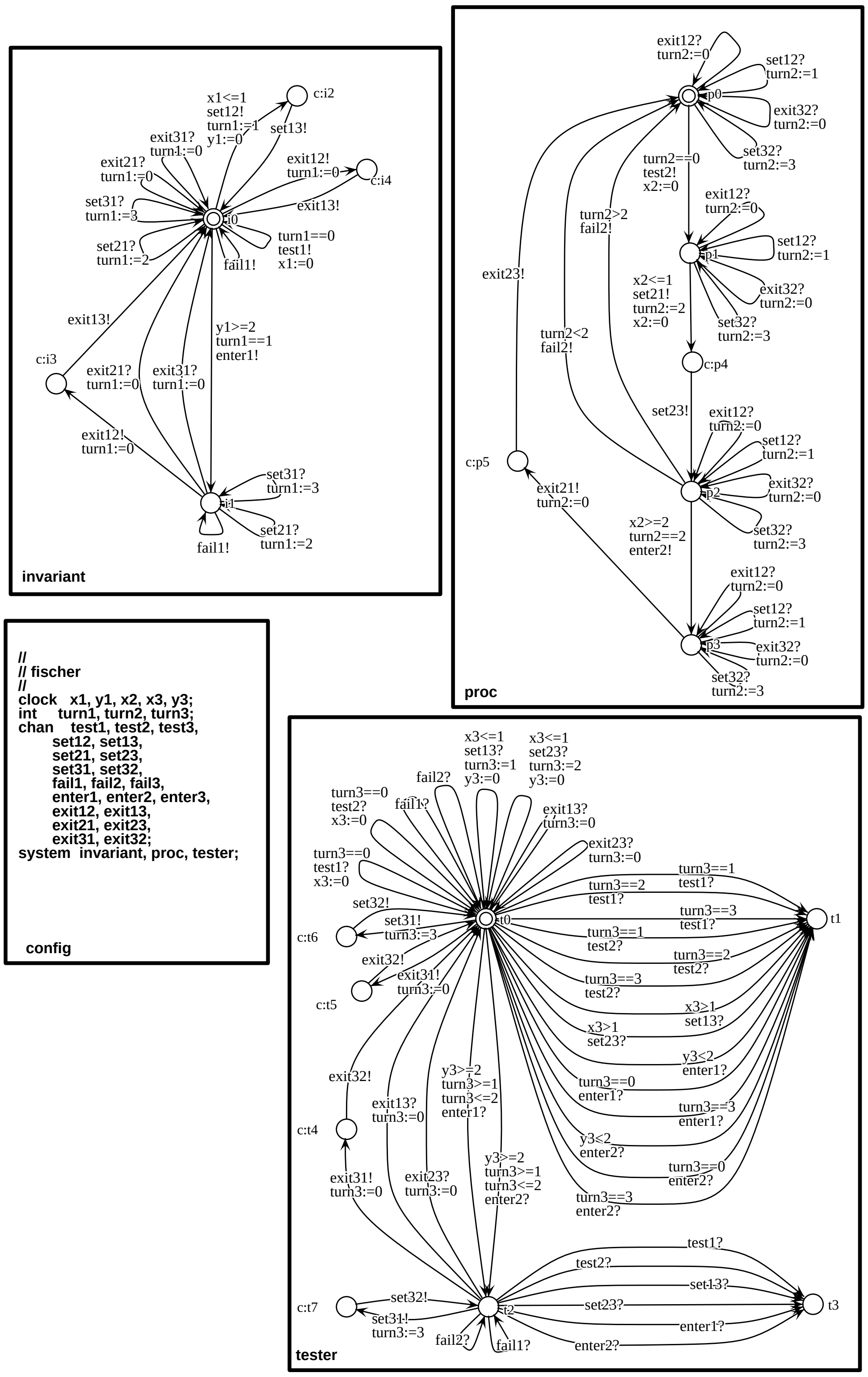
```
A[] not (tester.t1 or tester.t3)
```

We have verified the above property successfully in UPPAAL for both inputs (cases $n = 1$ and $n > 1$). The verification taking less than 2 seconds. We can therefore conclude that I_n is an invariant for F_n for any n.



```
//
// fischer
//
clock   x2, x3, y3;
int     turn2, turn3;
chan    test2, test3,
        set23, set32,
        fail2, fail3,
        enter2, enter3,
        exit23, exit32;
system  proc, tester;
```

Figure 6.5: Automata P_2 and T_{I_2} in case $n = 1$



```
//
// fischer
//
clock   x1, y1, x2, x3, y3;
int     turn1, turn2, turn3;
chan    test1, test2, test3,
        set12, set13,
        set21, set23,
        set31, set32,
        fail1, fail2, fail3,
        enter1, enter2, enter3,
        exit12, exit13,
        exit21, exit23,
        exit31, exit32;
system  invariant, proc, tester;
```

Figure 6.6: Automata I_1, P_2 and T_{I_2} in case $n > 1$

Chapter 7

Conclusion

7.1 Thesis Summary

Abstraction Frameworks. In this thesis we have provided weakly preserving abstraction frameworks for untimed as well as timed systems. The untimed framework is based on I/O automata and it extends the standard I/O automata theory with sound conditions for property preservation from one automaton (the abstract one) to another (the concrete one). We provide conditions for both action-based abstractions and state-based abstractions. The action setting is based on properties expressed as sequences of actions (trace properties) and the state setting is based on properties expressed as sequences of states (path properties). For both settings we provide preservation conditions for safety as well as liveness properties. Our preservation conditions are based on variants of the forward simulation preorder tailored for abstractions based on actions and states, respectively, and the conditions therefore fits well into the existing I/O automata theory. To provide tool support for doing abstraction-based verification in the I/O automaton setting we have formalized parts of our abstraction theory in the Larch tool set and we have examined a rudimentary scheme for translating finite-state I/O automata and trace/path properties into the SPIN model checker.

In the timed setting we have provided an abstraction framework for real-time systems described as timed automata. Our framework provides a link to the UPPAAL real-time model checker. The verification engine of UPPAAL is based on efficient techniques for abstracting (strongly) the dense time domain of real-time systems into a finite representation. These techniques rely on certain restrictions imposed by the timed automaton model on the allowed clock conditions in system descriptions. Thus, UPPAAL can efficiently abstract the timing component of properly described real-time systems. However, it does not allow systems with unbounded control or data information such as for example parameterized systems consisting of a number of composed processes, where the value of the number is the parameter, or systems with unbounded number of actions or unbounded data domains. We provide conditions for sound abstractions of such systems. We rely on system requirements expressed as abstract automata and a satisfaction relation in the form of a timed ready simulation relation. Our preservation conditions are based on a variant of this timed ready simulation as well. We show that this variant enjoys properties such as preservation under system composition thus supporting hierarchical verification. The UPPAAL tool is based on verifying simple reachability properties and thus it does not directly implement methods for checking the existence of timed ready simulations between automata. However, we provide a method for translating the check for timed ready simulations into a reachability problem suitable for the UPPAAL tool.

Applied Abstraction Strategies. A main result of this thesis is the demonstration, through case studies, that our abstraction frameworks are indeed useful in practice. The difficult part in using the frameworks is finding suitable abstractions for given concrete systems. Useful abstraction strategies depend on the concrete problem at hand. We have presented useful abstraction strategies for the proofs of three nontrivial distributed algorithms. All our proofs have the advantage that the essential functionality of the considered algorithms is preserved in their finite-state abstractions. Thus, proving properties about this functionality is performed by model checking.

In our proof of the parameterized version of Burns' mutual exclusion algorithm we use a skolemization abstraction strategy to construct an abstract interpretation of any pair of concrete processes including the possible effects of processes in the environment. The skolemization strategy utilizes that the mutual exclusion property is stated as a conjunction over pairs of process indices. We have used the LP theorem prover to show that our abstraction is indeed property preserving and the SPIN model checker to verify the abstraction.

Our abstraction-based proof of the Bounded Concurrent Timestamp System (BCTSS) algorithm is the most advanced in this thesis. The BCTSS algorithm is one of the most complicated algorithms in the distributed systems literature and existing proofs are all long and hard to understand. We provide an abstraction-based proof of a key invariant of this algorithm established and proved by hand within the I/O automaton model in [GLS92]. Our proof exploits a combination of induction and abstraction strategies and it reduces the required amount of manually proven subinvariants from the original proof.

Our proof of the parametrized version of Fischer's mutual exclusion algorithm utilizes a combination compositionality and abstraction strategies. Our abstraction strategy involves the use of a network invariant. This invariant is shown to timed ready simulate the concrete system independently of its number of component processes. Our network invariant is parametrized (in the number of components it simulates) and has an unbounded number of transitions, but only finitely many locations. Using data abstraction techniques we reduce the parameterized invariant to a simple finite-state system.

7.2 Future Work

Case Studies – Liveness. All case studies considered in this thesis deals with abstraction-based verification of safety properties. For real-time systems most interesting "liveness" properties are *bounded liveness* properties, stating that something *good* happens within a certain time bound, and such properties are actually safety properties. However, for untimed systems real liveness properties do exist. Our untimed abstraction framework provides preservation conditions for liveness properties but we have not yet had any practical experience in using these conditions to prove liveness under abstraction. In the I/O automaton model used in this thesis liveness it actually treated in a restricted form called fairness. In [GSSL93] a generalized I/O automaton model is considered permitting the verification of general liveness properties. Further research may investigate how our abstraction conditions can be generalized to this setting.

We believe that further case studies, regarding safety as well as liveness properties and for timed as well as untimed systems, are of importance to further investigate common abstraction patterns for classes of systems.

Integrated Tool Support. In this thesis we have only presented a rudimentary scheme for translating I/O automata into the SPIN model checker. Thus we have not actually integrated SPIN with the I/O automata framework. Recent research has taken place that defines a specification language IOA [GLV97] for I/O automata supported by a parser and syntax checker. Interesting future research aims at tools for extracting proof obligations from algorithm descriptions and presenting them to theorem provers like LP and model checkers like SPIN. This work is already in progress at the Laboratory for Computer Science at M.I.T.

Further tool support integrating our timed abstraction framework with the UPPAAL model checker also remains to be investigated.

Bibliography

[ABL98] Luca Aceto, Augusto Burgueno, and Kim G. Larsen. Model checking via reachability testing for timed automata. In Bernhard Steffen, editor, *Proc. 4th In.t Conference on Tools ans Algorithms for the Construction and analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 263–280. Springer, 1998.

[Abr88] K. Abrahamson. On Achieving Consensus using a Shared Memory. In *Proceedings of 7th ACM Symposium on the Principles of Distributed Computing, Toronto, Ontario, Canada*, 1988.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–236, 1994.

[ADG+94] Y. Afek, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. A Bounded First-in, First-enabled Solution to the ℓ-exclusion Problem. In *ACM TOPLAS*, pages (16)3:939–953, 1994.

[AHH96] R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, pages 22:181–201, 1996.

[AJ98] Parosh Aziz Abdulla and Bengt Jonsson. Verifying networks of timed processes. In Bernhard Steffen, editor, *TACAS'98, Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 298–312, Lisbon, Portugal, March/April 1998. Springer.

[AL93] Martin Abadi and Leslie Lamport. An Old-Fashioned Recipe for Real Time. *Lecture Notes in Computer Science*, 600, 1993.

[BB89] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. Technical Report P8916, University of Amsterdam, 1989.

[BCM+90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} states and beyond. *Logic in Computer Science*, 1990.

[BHK86] J.A. Bergstra, J. Heering, and P. Klint. Algebra of communicating processes. In *CWI Symposium on Mathematics and Computer Science*, pages 89–138. North-Holland, 1986.

[BIM95] B. Bloom, S. Istrail, and A. Meyer. Bisimulation can't be traced. *Journal of the Association for Computing Machinery*, pages 42(1):232–268, 1995.

[BLL$^+$95] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, Lecture Notes in Computer Science, pages 22–24. Springer Verlag, October 1995.

[Bur78] James E. Burns. Mutual exclusion with linear waiting using binary shared variables. *ACM SIGACT News*, 1978.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks, 1979.

[CC92a] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, pages 13:103–179, 1992.

[CC92b] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, pages 2(4):511–547, 1992.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, 1986.

[CGL92] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1992.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. The edinburgh concurrency workbench: A semantics-based verification tool for finite-state systems. In *Proceedings of the Workshop on Automated Verification Tools for Finite-State Systems*, volume 407 of *Lecture Notes in Computer Science.* Springer Verlag, 1989.

[Dam96] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking.* PhD thesis, Eindhoven University of Technology, 1996.

[DF95] Jürgen Dingel and Thomas Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In *Proc. of CAV'95*, volume 939 of *Lecture Notes in Computer Science*, pages 54–69, 1995.

[DL97] Ekaterina Dolginova and Nancy Lynch. Safety verification for automated platoon maneuvers: A case study. In *Proceedings Int. Workshop on Hybrid and Real-Time Systems (HART'97)*, volume 1201 of *Lecture Notes in Computer Science*, pages 154–170. Springer Verlag, 1997.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool kronos. In *Hybrid Systems III, Verification and Control*, volume 1066 of *Lecture Notes in Computer Science.* Spinger Verlag, 1996.

[DS89] Danny Dolev and Nir Shavit. Bounded Concurrent Time-Stamp Systems Are Constructible. In *Prooceedings of the 21st ACM Symposium on Theory of Computing. Also in SIAM Journal of Computing*, pages (26)2:418–455, 1989.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proc. of the 16*th *IEEE Real-Time Systems Symposium*, pages 66–75, December 1995.

[Gaw92] Rainer Gawlick. Concurrent timestamping made simple. Master's thesis, Massachusetts Institute of Technology, 1992.

[GG91] S.J. Garland and J.V. Guttag. A Guide to LP, the Larch Prover. Technical Report Research Report 82, Digital Systems Research Center, 1991.

[GH93] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer Verlag, 1993.

[GLS92] Rainer Gawlick, Nancy Lynch, and Nir Shavit. Concurrent Timestamping Made Simple. In *Israel Symposium on Theory and Practice of Computing*, 1992.

[GLV97] Stephen Garland, Nancy Lynch, and Mandana Vaziri. IOA: A language for specifying, programming, and validating distributed systems. Technical report, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, 1997.

[GM93] M.C.J. Gordon and T.F. Melham. *Introduction to HOL: a theorem-proving environment for higher-order logic.* Cambridge University Press, 1993.

[GSSL93] Rainer Gawlick, Roberto Segala, Jørgen Søgaard-Andersen, and Nancy Lynch. Liveness in timed and untimed systems. Technical Report MIT/LCS/TR-587, Massachusetts Institute of Technology, Laboratory for Computer Science, December 1993.

[GW94] P. Godefroid and P. Wolper. A partial approach to model checking. *Information and Computation*, pages 110:205–326, 1994.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, pages 8:231–274, 1987.

[HHK95] Monika R. Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing Simulations on Finite and Infinite Graphs. In *36th Annual Symposium on Foundations of Computer Science*, pages 453–462. IEEE Computer Society Press, 1995.

[HK87] Z. Har'El and R.P. Kurshan. The cospan user's guide. Technical report, AT&T Bell Laboratories, 1987.

[HL94] Constance Heitmeyer and Nancy Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. Technical Report MIT/LCS/TM-511, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, November 1994.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, pages 29(6/7):523–543, 1992.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, pages 32(1):137–161, 1985.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice–Hall, 1985.

[Hol91] Gerard Holzmann. *The Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[HP94] G.J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. 7th Int. Conf. on Formal Description Techniques (FORTE'94)*, pages 177–194, Berne, Switzerland, 1994.

[HWT95] Pei-Hsin Ho and Howard Wong-Toi. Automated analysis of an audio control protocol. In *Proceedings of CAV'95*, volume 939 of *Lecture Notes in Computer Science*, 1995.

[IS94] A. Ingolfsdottir and B. Steffen. Characteristic formulae for processes with divergence. *Information and Computation*, pages 110(1):149–163, 1994.

[JL98] Henrik E. Jensen and Nancy A. Lynch. A Proof of Burns N-Process Mutual Exclusion Algorithm Using Abstraction. In Bernhard Steffen, editor, *TACAS'98, Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, Lisbon, Portugal, March/April 1998. Springer.

[KLL$^+$97] K.J. Kristoffersen, F. Larroussinie, K.G. Larsen, P. Petterson, and W. Yi. A compositional proof of a real-time mutual exclusion protocol. In *TAPSOFT'97 7th International Joint Conference on the Theory and Practice of Software Development*, Lecture Notes in Computer Science, Lille, France, April 1997. Springer Verlag.

[KM89] R.P. Kurshan and K. McMillan. A Structural Induction Theorem for Processes. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, 1989.

[Koz82] D. Kozen. Results on the propositional mu-calculus. In *Proc. of International Colloquium on Algorithms, Languages and Programming 1982*, volume 140 of *Lecture Notes in Computer Science, Springer Verlag*, Berlin, 1982.

[Kri98] Kåre Jelling Kristoffersen. *Compositional Verification of Concurrent Systems.* PhD thesis, Aalborg University, Department of Computer Science, Institute for Electronic Systems, Aalborg, Denmark, August 1998.

[Kur89] R.P. Kurshan. Analysis of Discrete Event Coordination. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 414–454. Springer Verlag, 1989.

[Kur94] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes - The Automata-Theoretic Approach.* Princeton University Press, 1994.

[Lam74] Leslie Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, pages 78(8): 453–455, 1974.

[Lam86] Leslie Lamport. On Interprocess Communication. Parts I and II. *Distributed Computing*, pages 1, 77–101, 1986.

[LGS$^+$95] 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*, pages 6:11–44, 1995.

[LLSA94] Butler W. Lampson, Nancy A. Lynch, and Jørgen F. Søgaard-Andersen. Correctness of at-most-one message delivery protocols. In Richard L. Tenney, Paul D. Amer, and M. Ümit Uyan, editors, *Formal Description Techniques VI (Proceedings of the IFIP TC6/WG6.1 Sixth International Conference on Formal Description Techniques, FORTE'93, Boston, October, 1993) IFIP Transactions C*, pages 385–400. North–Holland, Amsterdam, 1994.

[LMWF94] Nancy Lynch, Michael Merrit, William Weihl, and Alan Fekete. *Atomic Transactions.* Morgan Kaufmann, San Mateo, Calif., 1994.

[LS91] K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, pages 94(1):1–28, 1991.

[LSVW95] Nancy Lynch, Roberto Segala, Fritz Vaandrager, and H.B. Weinberg. Hybrid I/O Automata. In R. Alur, T. Henzinger, and E. Sontag, editors, *Hybrid Systems III: Verification and Control (DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems, New Brunswick, New Jersey, October 1995)*, volume 1066 of *Lecture Notes in Computer Science*, pages 496–510. Springer Verlag, 1995.

[LSW95] K.G. Larsen, B. Steffen, and C. Weise. Fischer's protocol revisited: A simple proof using modal constraints. In *4th DIMACS Workshop on Verification and Control of Hybrid Systems*, New Brunswick, New Jersey, 1995.

[LT87] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proc. of the 6th ACM Symposium on Principles of Distributed Computation*, pages 137–151, 1987.

[LT89] Nancy Lynch and Mark Tuttle. An Introduction to Input/Output Automata. *CWI-Quarterly*, pages 2(3)219–246, 1989.

[Luc95] Victor Luchangco. Using simulation techniques to prove timing properties. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, June 1995.

[LV95] N.A. Lynch and F. Vaandrager. Forward and backward simulations - part i: Untimed systems. *Information and Computation*, pages 121(2):214–233, 1995.

[LV96] N.A. Lynch and F. Vaandrager. Forward and backward simulations - part ii: Timing-based systems. *Information and Computation*, 1996.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[McM93] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, Englewood Cliffs, 1989.

[MN95] Olaf Müller and Tobias Nipkow. Combining Model Checking and Deduction for I/O-Automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 1–16. Springer Verlag, 1995.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1992.

[NSY91] X. Nicollin, J. Sifakis, and S. Yovine. From atp to timed graphs and hybrid systems. In *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science.* Springer-Verlag, 1991.

[ORR$^+$96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. Pvs: combining specification, proof checking and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science.* Springer Verlag, 1996.

[Pau94] Lawrence C. Paulson. Isabelle: A generic theorem prover. In *Lecture Notes in Computer Science*, volume 828. Springer Verlag, 1994.

[Pnu86] Amir Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. *In Current Trends in Concurrency, J.W. de Bakker, W.-P. de Roever, and G. Rozenberg (editors). Lecture Notes in Computer Science, 224, Springer-Verlag, Berlin*, pages 510–584, 1986.

[SA93] Jørgen Søgaard-Andersen. *Correctness of Protocols in Distributed Systems.* PhD thesis, Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, December 1993. ID-TR: 1993-131.

[SAGG$^+$93] Jørgen Søgaard-Andersen, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computer-Assisted Simulation Proofs. In Costas Courcoubetis, editor, *Computer-Aided Verification (5th International Conference, CAV'93, Elounda, Greece, June/July 1993)*, volume 697 of *Lecture Notes in Computer Science*, pages 305–319. Springer Verlag, 1993.

[SL95] Roberto Segala and Nancy Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, pages 2(2):250–273, August 1995.

[VA95] P. Vitianyi and B. Awerbuch. Shared Register Access by Asynchronous Hardware. In *27th Symposium on the Foundations of Computer Science, Tel-Aviv*, 1995.

[WL89a] P. Wolper and V. Lovinfosse. Verifying poperties of large sets of processes with network invariants. *Lecture Notes in Computer Science, Springer Verlag*, 407, 1989. Proc. of Workshop on Automatic Verification Methods for Finite State Systems.

[WL89b] Pierre Wolper and Vincianne Lovinfosse. Verifying Properties of Large Sets of Processes with Network Invariants, 1989.

[WLL88] Jennifer Lundelius Welch, Leslie Lamport, and Nancy Lynch. A lattice-structured proof technique applied to a minimum spanning tree algorithm. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, Toronto, Ontario, Canada, 1988.

[Yi90] Wang Yi. Real-time behaviour of asynchronous agents. In J.C.M. Baeten and J.W. Klop, editors, *Proc. of the Conference on Theories of Concurrency: Unification and Extension, CONCUR'90*, volume 458 of *Lecture Notes in Computer Science*, pages 502–520. Springer-Verlag, 1990.

[Yi91] Wang Yi. *A Calculus of Real Time Systems.* PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1991.

Appendix A

Proof of Theorem 4.2

Proof. If $s_0 \in \mathit{start}(\mathit{Burns})$ then,

$$\begin{aligned} s_0.\mathit{upc}_i &= s_0.\mathit{upc}_j = \mathit{rem} \\ s_0.\mathit{ppc}_i &= s_0.\mathit{ppc}_j = \mathit{rem} \\ s_0.\mathit{flag}_i &= s_0.\mathit{flag}_j = 0 \\ s_0.S_i &= s_0.S_j = \emptyset \end{aligned}$$

and if $u_0 \in \mathit{start}(\mathit{Burns}_\alpha)$ then,

$$\begin{aligned} u_0.\mathit{upc}_0 &= u_0.\mathit{upc}_1 = \mathit{rem} \\ u_0.\mathit{ppc}_0 &= s_0.\mathit{ppc}_1 = \mathit{rem} \\ u_0.\mathit{flag}_0 &= u_0.\mathit{flag}_1 = 0 \\ u_0.S_0 &= u_0.S_1 = \emptyset \end{aligned}$$

so $S_{\{i,j\}}(s_0, u_0)$.

Now, let $s \in \mathit{states}(\mathit{Burns})$ and let $u \in \mathit{states}(\mathit{Burns}_\alpha)$ s.t. $S_{\{i,j\}}(s, u)$. We consider cases based on the type of action π_x performed by s on a transition $s \xrightarrow{\pi_x}_{\mathit{Burns}} s'$. For each action π_x we consider $x = i$, $x = j$ and $x \notin \{i, j\}$. If $x \notin \{i, j\}$, it is obvious from *Burns* that no shared or local variables in processes or users with indices i or j change. So any transition $s \xrightarrow{\pi_x} s'$ in *Burns*, with $x \notin \{i, j\}$, can be matched by Burns_α doing no action and we still have $S_{\{i,j\}}(s', u)$.

Case 1 ($\pi_x = \mathit{try}_x$):

Case a ($x = i$):

The corresponding execution fragment is $u \xrightarrow{\mathit{try}_0} u'$. try_0 is enabled in u as $u.\mathit{upc}_0 = s.\mathit{upc}_i = \mathit{rem}$. From *Burns* the only changes are, $s'.\mathit{upc}_i = \mathit{try}$ and $s'.\mathit{ppc}_i = \mathit{set\text{-}flg\text{-}0}$, and from Burns_α the only changes are, $u'.\mathit{upc}_0 = \mathit{try}$ and $u'.\mathit{ppc}_0 = \mathit{set\text{-}flg\text{-}0}$ so $S_{\{i,j\}}(s', u')$.

Case b ($x = j$):

The corresponding execution fragment is $u \xrightarrow{\mathit{try}_1} u'$. Analogous to above.

Case 2 ($\pi_x = \textit{set-flg-0}_x$):

Case a ($x = i$):

Case i ($i = 1$):

The corresponding execution fragment is $u \xrightarrow{\textit{set-flg-0-sml}_0} u'$. *set-flg-0-sml*$_0$ is enabled in u as $u.ppc_0 = s.ppc_i = \textit{set-flg-0}$. From *Burns* the only changes are $s'.\textit{flag}_i = 0$ and $s'.ppc_i = \textit{set-flg-1}$. From $\textit{Burns}_\alpha$ the only changes are, $u'.\textit{flag}_0 = 0$ and $u'.ppc_0 = \textit{set-flg-1}$ so $S_{\{i,j\}}(s', u')$.

Case ii ($i \neq 1$):

The corresponding execution fragment is $u \xrightarrow{\textit{set-flg-0}_0} u'$. Fragment is enabled in u as $u.ppc_0 = s.ppc_i = \textit{set-flg-0}$. From *Burns* the only changes are $s'.\textit{flag}_i = 0$ and $s'.ppc_i = \textit{test-sml-fst}$. From $\textit{Burns}_\alpha$ the only changes are, $u'.\textit{flag}_0 = 0$ and $u'.ppc_0 = \textit{test-sml-fst}$ so $S_{\{i,j\}}(s', u')$.

Case b ($x = j$):

The corresponding execution fragment is $u \xrightarrow{\textit{set-flg-0}_1} u'$. *set-flg-0*$_1$ is enabled in u as $u.ppc_1 = s.ppc_j = \textit{set-flg-0}$. As $j > i$ and $i \geq 1$ we know that $j > 1$ and hence from *Burns* the only changes are $s'.\textit{flag}_j = 0$ and $s'.ppc_j = \textit{test-sml-fst}$. From $\textit{Burns}_\alpha$ the only changes are, $u'.\textit{flag}_1 = 0$ and $u'.ppc_1 = \textit{test-sml-fst}$ so $S_{\{i,j\}}(s', u')$.

Case 3 ($\pi_x = \textit{test-sml-fst}(y)_x$):

Case a ($x = i$):

Case i ($s.\textit{flag}_y = 1$):

The corresponding fragment is $u \xrightarrow{\textit{test-sml-fail}_0} u'$. *test-sml-fail*$_0$ is enabled in u as $u.ppc_0 = s.ppc_i = \textit{test-sml-fst}$. From *Burns* the only changes are $s'.S_i = \emptyset$ and $s'.ppc_i = \textit{set-flg-0}$, and from $\textit{Burns}_\alpha$ the only changes are $u'.ppc_0 = \textit{set-flg-0}$, so $S_{\{i,j\}}(s', u')$.

Case ii ($s.\textit{flag}_y = 0$, $|s.S_i| < i - 2$):

The corresponding fragment is u. From *Burns* the only change is $s'.S_i = s.S_i \cup \{y\}$ and as $y \neq j$, we have $S_{\{i,j\}}(s', u)$.

Case iii ($s.\textit{flag}_y = 0$, $|s.S_i| = i - 2$):

The corresponding fragment is $u \xrightarrow{\textit{test-sml-fst-succ}_0} u'$. Fragment enabled as $u.ppc_0 = s.ppc_i = \textit{test-sml-fst}$. From *Burns* the only changes are $s'.S_i = \emptyset$ and $s'.ppc_i = \textit{set-flg-1}$. From $\textit{Burns}_\alpha$ the only change is $u'.ppc_0 = \textit{set-flg-1}$, so $S_{\{i,j\}}(s', u')$.

Case b ($x = j$, $y = i$):

Case i **(**$s.\mathit{flag}_i = 1$**):**

The corresponding fragment is $u \xrightarrow{\mathit{test\text{-}sml\text{-}fail}_1} u'$. $\mathit{test\text{-}sml\text{-}fail}_1$ is enabled in u as $u.\mathit{ppc}_1 = s.\mathit{ppc}_j = \mathit{test\text{-}sml\text{-}fst}$. From *Burns* the only changes are $s'.S_j = \emptyset$ and $s'.\mathit{ppc}_j = \mathit{set\text{-}flg\text{-}0}$, and from Burns_α the only changes are $u'.S_1 = \emptyset$ and $u'.\mathit{ppc}_1 = \mathit{set\text{-}flg\text{-}0}$, so $S_{\{i,j\}}(s', u')$.

Case ii **(**$s.\mathit{flag}_i = 0$, $|s.S_j| < j - 2$**):**

If $u.S_1 = \{0\}$ the corresponding fragment is u. From *Burns* the only change is $s'.S_j = s.S_j \cup \{i\}$ and as $u.P_1.S = \{0\}$, we have $S_{\{i,j\}}(s', u)$.

If $u.P_1.S = \emptyset$ let the corresponding execution fragment be $u \xrightarrow{\mathit{test\text{-}other\text{-}flg}_1} u'$. Fragment is enabled as $u.P_1.pc = s.\mathit{ppc}_j = \mathit{test\text{-}sml\text{-}fst}$, and $u.P_1.S = \emptyset$. From *Burns* the only change is, as above, $s'.S_j = s.S_j \cup \{i\}$. From Burns_α the only change is $u'.P_1.S = \{0\}$, as $u.\mathit{flag}_0 = s.\mathit{flag}_i$, so $S_{\{i,j\}}(s', u')$.

Case iii **(**$s.\mathit{flag}_i = 0$, $|s.S_j| = j - 2$**):**

If $u.P_1.S = \{0\}$ the corresponding execution fragment is $u \xrightarrow{\mathit{test\text{-}sml\text{-}fst\text{-}succ}_1} u'$. Fragment is enabled as $u.P_1.pc = s.\mathit{ppc}_j = \mathit{test\text{-}sml\text{-}fst}$ and $u.P_1.S = \{0\}$. From *Burns* the only changes are $s'.S_j = \emptyset$ and $s'.\mathit{ppc}_j = \mathit{set\text{-}flg\text{-}1}$. From Burns_α the only changes are $u'.P_1.S = \emptyset$ and $u'.P_1.pc = \mathit{set\text{-}flg\text{-}1}$, so $S_{\{i,j\}}(s', u')$.

If $u.P_1.S = \emptyset$ then let the corresponding execution fragment be $u \xrightarrow{\mathit{test\text{-}other\text{-}flg}_1} u'' \xrightarrow{\mathit{test\text{-}sml\text{-}fst\text{-}succ}_1} u'$. $\mathit{test\text{-}other\text{-}flg}_1$ is enabled in u as $u.P_1.pc = s.\mathit{ppc}_j = \mathit{test\text{-}sml\text{-}fst}$ and $u.P_1.S = \emptyset$. From Burns_α, $u''.P_1.pc = \mathit{test\text{-}sml\text{-}fst}$ and $u''.P_1.S = \{0\}$ as $u.\mathit{flag}_0 = s.\mathit{flag}_i$. Therefore, $\mathit{test\text{-}sml\text{-}fst\text{-}succ}_1$ is enabled in u''. From *Burns* the changes are $s'.S_j = \emptyset$ and $s'.\mathit{ppc}_j = \mathit{set\text{-}flg\text{-}1}$, and from Burns_α the changes are, $u'.P_1.S = \emptyset$ and $u'.P_1.pc = \mathit{set\text{-}flg\text{-}1}$, so $S_{\{i,j\}}(s', u')$.

Case c **(**$x = j$, $y \neq i$**):**

Case i **(**$s.\mathit{flag}_y = 1$**):**

The corresponding fragment is $u \xrightarrow{\mathit{test\text{-}sml\text{-}fail}_1} u'$. $\mathit{test\text{-}sml\text{-}fail}_1$ is enabled in u as $u.\mathit{ppc}_1 = s.\mathit{ppc}_j = \mathit{test\text{-}sml\text{-}fst}$. From *Burns* the only changes are $s'.S_j = \emptyset$ and $s'.\mathit{ppc}_j = \mathit{set\text{-}flg\text{-}0}$, and from Burns_α the only changes are $u'.S_1 = \emptyset$ and $u'.\mathit{ppc}_1 = \mathit{set\text{-}flg\text{-}0}$, so $S_{\{i,j\}}(s', u')$.

Case ii **(**$s.\mathit{flag}_y = 0$, $|s.S_j| < j - 2$**):**

The corresponding fragment is u. From *Burns* the only change is $s'.S_j = s.S_j \cup \{y\}$ and as $y \neq i$, we have $S_{\{i,j\}}(s', u)$.

Case iii **(**$s.\mathit{flag}_y = 0$, $|s.S_j| = j - 2$**):**

The corresponding fragment is $u \xrightarrow{\textit{test-sml-fst-succ}_1} u'$. Fragment enabled as $u.\mathit{ppc}_1 = s.\mathit{ppc}_j = \textit{test-sml-fst}$ and $u.P_1.S = \{0\}$ as $i \in s.S_j$[1]. From *Burns* the only changes are $s'.S_j = \emptyset$ and $s'.\mathit{ppc}_j = \textit{set-flg-1}$. From Burns_α the only changes are $u'.P_1.S = \emptyset$ and $u'.P_1.\mathit{pc} = \textit{set-flg-1}$, so $S_{\{i,j\}}(s', u')$.

Case 4 ($\pi_x = \textit{set-flg-1}_x$)**:**

Case a ($x = i$)**:**

Case i ($i = 1$)**:**

The corresponding execution fragment is $u \xrightarrow{\textit{set-flg-1-sml}_0} u'$. $\textit{set-flg-1-sml}_0$ is enabled in u as $u.\mathit{ppc}_0 = s.\mathit{ppc}_i = \textit{set-flg-1}$. From *Burns* the only changes are $s'.\mathit{flag}_i = 1$ and $s'.\mathit{ppc}_i = \textit{test-lrg}$. From Burns_α the only changes are, $u'.\mathit{flag}_0 = 1$ and $u'.\mathit{ppc}_0 = \textit{test-lrg}$ so $S_{\{i,j\}}(s', u')$.

Case ii ($i \neq 1$)**:**

The corresponding execution fragment is $u \xrightarrow{\textit{set-flg-1}_0} u'$. Fragment is enabled in u as $u.\mathit{ppc}_0 = s.\mathit{ppc}_i = \textit{set-flg-1}$. From *Burns* the only changes are $s'.\mathit{flag}_i = 1$ and $s'.\mathit{ppc}_i = \textit{test-sml-snd}$. From Burns_α the only changes are, $u'.\mathit{flag}_0 = 1$ and $u'.\mathit{ppc}_0 = \textit{test-sml-snd}$ so $S_{\{i,j\}}(s', u')$.

Case b ($x = j$)**:**

The corresponding execution fragment is $u \xrightarrow{\textit{set-flg-1}_1} u'$. $\textit{set-flg-1}_1$ is enabled in u as $u.\mathit{ppc}_1 = s.\mathit{ppc}_j = \textit{set-flg-1}$. As $j > i$ and $i \geq 1$ we know that $j > 1$ and hence from *Burns* the only changes are $s'.\mathit{flag}_j = 1$ and $s'.\mathit{ppc}_j = \textit{test-sml-snd}$. From Burns_α the only changes are, $u'.\mathit{flag}_1 = 1$ and $u'.\mathit{ppc}_1 = \textit{test-sml-snd}$ so $S_{\{i,j\}}(s', u')$.

Case 5 ($\pi_x = \textit{test-sml-snd}(y)_x$)**:**

Case a ($x = i$)**:**

Case i ($s.\mathit{flag}_y = 1$)**:**

The corresponding fragment is $u \xrightarrow{\textit{test-sml-fail}_0} u'$. $\textit{test-sml-fail}_0$ is enabled in u as $u.\mathit{ppc}_0 = s.\mathit{ppc}_i = \textit{test-sml-snd}$. From *Burns* the only changes are $s'.S_i = \emptyset$ and $s'.\mathit{ppc}_i = \textit{set-flg-0}$, and from Burns_α the only changes are $u'.\mathit{ppc}_0 = \textit{set-flg-0}$, so $S_{\{i,j\}}(s', u')$.

Case ii ($s.\mathit{flag}_y = 0$, $|s.S_i| < i - 2$)**:**

The corresponding fragment is u. From *Burns* the only change is $s'.S_i = s.S_i \cup \{y\}$ and as $y \neq j$, we have $S_{\{i,j\}}(s', u)$.

[1]Relies on invariant: $\mathit{pc}_i = \textit{test-sml-fst} \wedge j \in S_i \Rightarrow 1 \leq j < i$

Case iii ($s.\mathit{flag}_y = 0$, $|s.S_i| = i - 2$)**:**

The corresponding fragment is $u \xrightarrow{\mathit{test\text{-}sml\text{-}snd\text{-}succ}_0} u'$. Fragment enabled as $u.\mathit{ppc}_0 = s.\mathit{ppc}_i = \mathit{test\text{-}sml\text{-}snd}$. From *Burns* the only changes are $s'.S_i = \emptyset$ and $s'.\mathit{ppc}_i = \mathit{test\text{-}lrg}$, as $i < j \leq n$. From Burns_α the only change is $u'.\mathit{ppc}_0 = \mathit{test\text{-}lrg}$, so $S_{\{i,j\}}(s', u')$.

Case b ($x = j$, $y = i$)**:**

Case i ($s.\mathit{flag}_i = 1$)**:**

The corresponding fragment is $u \xrightarrow{\mathit{test\text{-}sml\text{-}fail}_1} u'$. $\mathit{test\text{-}sml\text{-}fail}_1$ is enabled in u as $u.\mathit{ppc}_1 = s.\mathit{ppc}_j = \mathit{test\text{-}sml\text{-}snd}$. From *Burns* the only changes are $s'.S_j = \emptyset$ and $s'.\mathit{ppc}_j = \mathit{set\text{-}flg\text{-}0}$, and from Burns_α the only changes are $u'.S_1 = \emptyset$ and $u'.\mathit{ppc}_1 = \mathit{set\text{-}flg\text{-}0}$, so $S_{\{i,j\}}(s', u')$.

Case ii ($s.\mathit{flag}_i = 0$, $|s.S_j| < j - 2$)**:**

If $u.S_1 = \{0\}$ the corresponding fragment is u. From *Burns* the only change is $s'.S_j = s.S_j \cup \{i\}$ and as $u.P_1.S = \{0\}$, we have $S_{\{i,j\}}(s', u)$.

If $u.P_1.S = \emptyset$ let $u \xrightarrow{\mathit{test\text{-}other\text{-}flg}_1} u'$ be the corresponding fragment. Fragment is enabled as $u.P_1.pc = s.\mathit{ppc}_j = \mathit{test\text{-}sml\text{-}snd}$, and $u.P_1.S = \emptyset$. From *Burns* the only change is, as above, $s'.S_j = s.S_j \cup \{i\}$. From Burns_α the only change is $u'.P_1.S = \{0\}$, as $u.\mathit{flag}_0 = s.\mathit{flag}_i$, so $S_{\{i,j\}}(s', u')$.

Case iii ($s.\mathit{flag}_i = 0$, $|s.S_j| = j - 2$)**:**

Case A ($j = n$)**:**

If $u.P_1.S = \{0\}$ the corresponding execution fragment is $u \xrightarrow{\mathit{test\text{-}sml\text{-}snd\text{-}succ\text{-}lrg}_1} u'$. Fragment is enabled as $u.P_1.pc = s.\mathit{ppc}_j = \mathit{test\text{-}sml\text{-}snd}$ and $u.P_1.S = \{0\}$. From *Burns* the only changes are $s'.S_j = \emptyset$ and $s'.\mathit{ppc}_j = \mathit{leave\text{-}try}$. From Burns_α the only changes are $u'.P_1.pc = \mathit{leave\text{-}try}$, so $S_{\{i,j\}}(s', u')$.

If $u.P_1.S = \emptyset$ let the corresponding execution fragment be $u \xrightarrow{\mathit{test\text{-}other\text{-}flg}_1} u'' \xrightarrow{\mathit{test\text{-}sml\text{-}snd\text{-}succ\text{-}lrg}_1} u'$. $\mathit{test\text{-}other\text{-}flg}_1$ is enabled in u as $u.P_1.pc = s.\mathit{ppc}_j = \mathit{test\text{-}sml\text{-}snd}$ and $u.P_1.S = \emptyset$. From Burns_α, $u''.P_1.pc = \mathit{test\text{-}sml\text{-}snd}$ and $u''.P_1.S = \{0\}$ as $u.\mathit{flag}_0 = s.\mathit{flag}_i$. Therefore, $\mathit{test\text{-}sml\text{-}snd\text{-}succ\text{-}lrg}_1$ is enabled in u''. From *Burns* the changes are $s'.S_j = \emptyset$ and $s'.\mathit{ppc}_j = \mathit{leave\text{-}try}$. From Burns_α the changes are, $u'.P_1.pc = \mathit{leave\text{-}try}$, so $S_{\{i,j\}}(s', u')$.

Case B ($j \neq n$)**:**

If $u.P_1.S = \{0\}$ the corresponding execution fragment is $u \xrightarrow{\mathit{test\text{-}sml\text{-}snd\text{-}succ}_1} u'$. Fragment is enabled as $u.P_1.pc$

$= s.ppc_j =$ *test-sml-snd* and $u.P_1.S = \{0\}$. From *Burns* the only changes are $s'.S_j = \emptyset$ and $s'.ppc_j =$ *test-lrg*. From $Burns_\alpha$ the only changes are $u'.P_1.pc =$ *test-lrg*, so $S_{\{i,j\}}(s', u')$.

If $u.P_1.S = \emptyset$ let $u \xrightarrow{\textit{test-other-flg}_1} u'' \xrightarrow{\textit{test-sml-snd-succ}_1} u'$ be the corresponding fragment u'. *test-other-flg*$_1$ is enabled in u as $u.P_1.pc = s.ppc_j =$ *test-sml-snd* and $u.P_1.S = \emptyset$. From $Burns_\alpha$, $u''.P_1.pc =$ *test-sml-snd* and $u''.P_1.S = \{0\}$ as $u.flag_0 = s.flag_i$. Therefore, *test-sml-snd-succ*$_1$ is enabled in u''. From *Burns* the changes are $s'.S_j = \emptyset$ and $s'.ppc_j =$ *test-lrg*. From $Burns_\alpha$ the changes are, $u'.P_1.pc$ $=$ *test-lrg*, so $S_{\{i,j\}}(s', u')$.

Case c $(x = j,\ y \neq i)$**:**

Case i $(s.flag_y = 1)$**:**

The corresponding fragment is $u \xrightarrow{\textit{test-sml-fail}_1} u'$. *test-sml-fail*$_1$ is enabled in u as $u.ppc_1 = s.ppc_j =$ *test-sml-snd*. From *Burns* the only changes are $s'.S_j = \emptyset$ and $s'.ppc_j =$ *set-flg-0*, and from $Burns_\alpha$ the only changes are $u'.S_1 = \emptyset$ and $u'.ppc_1$ $=$ *set-flg-0*, so $S_{\{i,j\}}(s', u')$.

Case ii $(s.flag_y = 0,\ |s.S_j| < j - 2)$**:**

The corresponding fragment is u. From *Burns* the only change is $s'.S_j = s.S_j \cup \{y\}$ and as $y \neq i$, we have $S_{\{i,j\}}(s', u)$.

Case iii $(s.flag_y = 0,\ |s.S_j| = j - 2)$**:**

Case A $(j = n)$**:**

The corresponding fragment is $u \xrightarrow{\textit{test-sml-snd-succ-lrg}_1} u'$. Fragment enabled as $u.ppc_1 = s.ppc_j =$ *test-sml-snd* and $u.P_1.S = \{0\}$ as $i \in s.S_j$[2]. From *Burns* the only changes are $s'.S_j = \emptyset$ and $s'.ppc_j =$ *leave-try*, and from $Burns_\alpha$ the only changes are $u'.P_1.pc =$ *leave-try*, so $S_{\{i,j\}}(s', u')$.

Case B $(j \neq n)$**:**

The corresponding fragment is $u \xrightarrow{\textit{test-sml-snd-succ}_1} u'$. Fragment enabled as $u.ppc_1 = s.ppc_j =$ *test-sml-snd* and $u.P_1.S = \{0\}$ as $i \in s.S_j$[3]. From *Burns* the only changes are $s'.S_j = \emptyset$ and $s'.ppc_j =$ *test-lrg*, and from $Burns_\alpha$ the only changes are $u'.P_1.pc =$ *test-lrg*, so $S_{\{i,j\}}(s', u')$.

Case 6 $(\pi_x = \textit{test-lrg}(y)_x)$**:**

Case a $(x = i,\ y = j)$**:**

[2]Relies on invariant: $pc_i = \textit{test-sml-snd} \wedge j \in S_i \Rightarrow 1 \leq j < i$.

[3]As above.

Case i $(s.\mathit{flag}_j = 1)$**:**

The corresponding fragment is $u \xrightarrow{\textit{test-lrg-fail}_0} u'$. *test-lrg-fail*$_0$ is enabled in u as $u.\mathit{ppc}_0 = s.\mathit{ppc}_i = \textit{test-lrg}$. From *Burns* the only changes are $s'.S_i = \emptyset$, and from *Burns*$_\alpha$ the only changes are $u'.S_0 = \emptyset$, so $S_{\{i,j\}}(s', u')$.

Case ii $(s.\mathit{flag}_j = 0,\ |s.S_i| < n - i - 1)$**:**

If $u.S_0 = \{1\}$ the corresponding fragment is u. From *Burns* the only change is $s'.S_i = s.S_i \cup \{j\}$ and as $u.P_0.S = \{1\}$, we have $S_{\{i,j\}}(s', u)$.

If $u.P_0.S = \emptyset$ let $u \xrightarrow{\textit{test-other-flg}_0} u'$ be the corresponding fragment. Fragment is enabled as $u.P_0.pc = s.\mathit{ppc}_i = \textit{test-lrg}$, and $u.P_0.S = \emptyset$. From *Burns* the only change is, as above, $s'.S_i = s.S_i \cup \{j\}$. From *Burns*$_\alpha$ the only change is $u'.P_0.S = \{1\}$, as $u.\mathit{flag}_1 = s.\mathit{flag}_j$, so $S_{\{i,j\}}(s', u')$.

Case iii $(s.\mathit{flag}_j = 0,\ |s.S_i| = n - i - 1)$**:**

If $u.P_0.S = \{1\}$ let $u \xrightarrow{\textit{test-lrg-succ}_0} u'$ be corresponding fragment. Fragment is enabled as $u.P_0.pc = s.\mathit{ppc}_i = \textit{test-lrg}$ and $u.P_0.S = \{1\}$. From *Burns* the changes are $s'.S_i = s.S_i \cup \{j\}$ and $s'.\mathit{ppc}_i = \textit{leave-try}$. From *Burns*$_\alpha$ the only changes are $u'.P_0.pc = \textit{leave-try}$, so $S_{\{i,j\}}(s', u')$.

If $u.P_0.S = \emptyset$ let the corresponding execution fragment be $u \xrightarrow{\textit{test-other-flg}_0} u'' \xrightarrow{\textit{test-lrg-succ}_0} u'$. *test-other-flg*$_0$ is enabled in u as $u.P_0.pc = s.\mathit{ppc}_i = \textit{test-lrg}$ and $u.P_0.S = \emptyset$. From *Burns*$_\alpha$, $u''.P_0.pc = \textit{test-lrg}$ and $u''.P_0.S = \{1\}$ as $u.\mathit{flag}_1 = s.\mathit{flag}_j$. Therefore, *test-lrg-succ*$_1$ is enabled in u''. From *Burns* the changes are $s'.S_i = s.S_i \cup \{j\}$ and $s'.\mathit{ppc}_i = \textit{leave-try}$, and from *Burns*$_\alpha$ the changes are, $u'.P_0.S = \{1\}$ and $u'.P_0.pc = \textit{leave-try}$, so $S_{\{i,j\}}(s', u')$.

Case b $(x = i,\ y \neq j)$**:**

Case i $(s.\mathit{flag}_y = 1)$**:**

The corresponding fragment is $u \xrightarrow{\textit{test-lrg-fail}_0} u'$. *test-sml-fail*$_0$ is enabled in u as $u.\mathit{ppc}_0 = s.\mathit{ppc}_i = \textit{test-lrg}$. From *Burns* the only changes are $s'.S_i = \emptyset$, and from *Burns*$_\alpha$ the only changes are $u'.S_0 = \emptyset$, so $S_{\{i,j\}}(s', u')$.

Case ii $(s.\mathit{flag}_y = 0,\ |s.S_i| < n - i - 1)$**:**

The corresponding fragment is u. From *Burns* the only change is $s'.S_i = s.S_i \cup \{y\}$ and as $y \neq j$, we have $S_{\{i,j\}}(s', u)$.

Case iii $(s.\mathit{flag}_y = 0,\ |s.S_i| = n - i - 1)$**:**

The corresponding fragment is $u \xrightarrow{\textit{test-lrg-succ}_0} u'$. Fragment enabled as $u.\mathit{ppc}_0 = s.\mathit{ppc}_i = \textit{test-lrg}$ and $u.P_0.S = \{1\}$ as j

$\in s.S_i$[4]. From *Burns* the only changes are $s'.ppc_i = \mathit{leave\text{-}try}$. From $Burns_\alpha$ the only changes are $u'.P_0.pc = \mathit{leave\text{-}try}$, so $S_{\{i,j\}}(s', u')$.

Case c $(x = j)$**:**

Case i $(s.\mathit{flag}_y = 1)$**:**

The corresponding fragment is $u \xrightarrow{\mathit{test\text{-}lrg\text{-}fail}_1} u'$. $\mathit{test\text{-}lrg\text{-}fail}_1$ is enabled in u as $u.ppc_1 = s.ppc_j = \mathit{test\text{-}lrg}$. From *Burns* there are no changes, and from $Burns_\alpha$ neither, so $S_{\{i,j\}}(s', u')$.

Case ii $(s.\mathit{flag}_y = 0,\ |s.S_j| < n - j - 1)$**:**

The corresponding fragment is u. From *Burns* the only change is $s'.S_j = s.S_j \cup \{y\}$ and as $y \neq i$, we have $S_{\{i,j\}}(s', u)$.

Case iii $(s.\mathit{flag}_y = 0,\ |s.S_j| = n - j - 1)$**:**

The corresponding fragment is $u \xrightarrow{\mathit{test\text{-}lrg\text{-}succ}_1} u'$. Fragment enabled as $u.ppc_1 = s.ppc_j = \mathit{test\text{-}lrg}$. From *Burns* the only changes are $s'.ppc_j = \mathit{leave\text{-}try}$. From $Burns_\alpha$ the only changes are $u'.ppc_1 = \mathit{leave\text{-}try}$, so $S_{\{i,j\}}(s', u')$.

Case 7 $(\pi_x = \mathit{crit}_x)$**:**

Case a $(x = i)$**:**

The corresponding execution fragment is $u \xrightarrow{\mathit{crit}_0} u'$. crit_0 is enabled in u as $u.ppc_0 = s.ppc_i = \mathit{leave\text{-}try}$. From *Burns* the only changes are, $s'.ppc_i = \mathit{crit}$, and from $Burns_\alpha$ the only changes are, $u'.ppc_0 = \mathit{crit}$, so $S_{\{i,j\}}(s', u')$.

Case b $(x = j)$**:**

The corresponding execution fragment is $u \xrightarrow{\mathit{crit}_1} u'$. Analogous to above.

Case 8 $(\pi_x = \mathit{exit}_x)$**:**

Case a $(x = i)$**:**

The corresponding execution fragment is $u \xrightarrow{\mathit{exit}_0} u'$. exit_0 is enabled in u as $u.upc_0 = s.upc_i = \mathit{crit}$. From *Burns* the only changes are, $s'.ppc_i = \mathit{reset}$, and from $Burns_\alpha$ the only changes are, $u'.ppc_0 = \mathit{reset}$, so $S_{\{i,j\}}(s', u')$.

Case b $(x = j)$**:**

The corresponding execution fragment is $u \xrightarrow{\mathit{exit}_1} u'$. Analogous to above.

Case 9 $(\pi_x = \mathit{reset}_x)$**:**

[4]Relies on invariant: $pc_i = \mathit{test\text{-}lrg} \wedge j \in S_i \Rightarrow i < j \leq n$.

Case a $(x = i)$**:**

The corresponding execution fragment is $u \xrightarrow{reset_0} u'$. $reset_0$ is enabled in u as $u.ppc_0 = s.ppc_i = reset$. From *Burns* the only changes are, $s'.flag_i = 0$, $s'.S_i = \emptyset$, and $s'.ppc_i = leave\text{-}exit$. From $Burns_\alpha$ the only changes are, $u'.flag_0 = 0$, $u'.S_0 = \emptyset$, and $u'.ppc_0 = leave\text{-}exit$, so $S_{\{i,j\}}(s', u')$.

Case b $(x = j)$**:**

The corresponding execution fragment is $u \xrightarrow{reset_1} u'$. Analogous to above.

Case 10 $(\pi_x = rem_x)$**:**

Case a $(x = i)$**:**

The corresponding execution fragment is $u \xrightarrow{rem_0} u'$. rem_0 is enabled in u as $u.ppc_0 = s.ppc_i = leave\text{-}exit$. From *Burns* the only changes are, $s'.ppc_i = rem$, and from $Burns_\alpha$ the only changes are, $u'.ppc_0 = rem$, so $S_{\{i,j\}}(s', u')$.

Case b $(x = j)$**:**

The corresponding execution fragment is $u \xrightarrow{rem_1} u'$. Analogous to above.

■

Appendix B

Proof of Lemma 5.6

Proof. The proof is by induction on the length of an execution and it is organized as the following sequence of Lemmas and Claims.

Lemma B.1 *The initial state satisfies* 1, 2 *and* 3.

Proof. 1, 2 and 3 holds in the initial state s_0 since for any i, $s_0.t_i = s_0.nt_i = 1^{n-1}$. ■

Now, assume that 1, 2 and 3 holds in s. Consider cases $s \xrightarrow{\pi} s'$ based on the type of action π.

Lemma B.2 *If* $\pi = update_k$ *then* s' *statisfies* 1, 2 *and* 3.

Proof. The following claims prove Lemma B.2.

Claim B.1 *If* $s.nt_k^{h-1} \preceq s.t_{max}^{h-1}$ *then* $s'.t_{max}^{h-1} = s.t_{max}^{h-1}$.

Proof. Suppose $s.t_j^{h-1} \neq s.t_{max}^{h-1}$ for all $j \neq k$. Consider any $j \neq k$. From the definition of t_{max}^{h-1}, $s.t_j^{h-1} \prec s.t_k^{h-1}$. By 1, $s.t_k^{h-1} = s.nt_k^{h-1}$ and as a result of the action $s'.t_k^{h-1} = s.nt_k^{h-1}$. Hence $s'.t_k^{h-1} = s.t_k^{h-1}$. Since t_k is the only label changing, $s'.t_j^{h-1} = s.t_j^{h-1}$. Hence, $s'.t_j^{h-1} \prec s'.t_k^{h-1}$ and from the definition of t_{max}^{h-1}, $s'.t_{max}^{h-1} = s'.t_k^{h-1}$. Now since $s'.t_k^{h-1} = s.t_k^{h-1} = s.t_{max}^{h-1}$, we have $s'.t_{max}^{h-1} = s.t_{max}^{h-1}$.

Suppose $j \neq k$ is such that $s.t_j^{h-1} = s.t_{max}^{h-1}$. Consider any $i \neq k$. Since t_k is the only label changing, $s'.t_i^{h-1} = s.t_i^{h-1}$. From definition of t_{max}^{h-1}, $s.t_i^{h-1} \preceq s.t_{max}^{h-1}$. Hence $s.t_i^{h-1} \preceq s.t_j^{h-1}$ and $s'.t_i^{h-1} \preceq s'.t_j^{h-1}$. By action, $s'.t_k^{h-1} = s.nt_k^{h-1}$ and by assumption, $s.nt_k^{h-1} \preceq s.t_j^{h-1}$. Hence $s'.t_k^{h-1} \preceq s'.t_j^{h-1}$. Now, from the definition of t_{max}^{h-1}, $s'.t_{max}^{h-1} = s'.t_j^{h-1}$, and since $s'.t_j^{h-1} = s.t_j^{h-1} = s.t_{max}^{h-1}$, we have that $s'.t_{max}^{h-1} = s.t_{max}^{h-1}$. ■

Claim B.2 *If* $s.nt_k^{h-1} \preceq s.t_{max}^{h-1}$ *then* s' *satisfies* 1.

Proof. Suppose for all $j \neq i$, $s'.t_j^{h-1} \neq s'.t_{max}^{h-1}$. By Claim B.1, $s'.t_{max}^{h-1} = s.t_{max}^{h-1}$ and since $s.t_{max}^{h-1} \neq -$ we have $s'.t_i^{h-1} = s'.t_{max}^{h-1}$. Suppose $i \neq k$. Since t_k is the only label changing, $s'.t_i^{h-1} = s.t_i^{h-1}$. Hence $s.t_i^{h-1} = s.t_{max}^{h-1}$. Also, for all $j \neq i$, $j \neq k$, $s'.t_j^{h-1} = s.t_j^{h-1}$ and hence $s.t_j^{h-1} \neq s.t_{max}^{h-1}$. By assumption $s'.t_k^{h-1} \neq s'.t_{max}^{h-1}$ and since $s'.t_k^{h-1} = s.nt_k^{h-1}$ and $s'.t_{max}^{h-1} = s.t_{max}^{h-1}$ we have that $s.nt_k^{h-1} \neq s.t_{max}^{h-1}$ and by hypothesis then $s.nt_k^{h-1} \prec s.t_{max}^{h-1}$. Thus, by 2 for s, $s.t_k^{h-1} \neq s.t_{max}^{h-1}$ and hence by 1 for s, $s.t_i^{h-1} = s.nt_i^{h-1} = s.t_{max}^{h-1}$. Now, since $s'.t_{max}^{h-1} = s.t_{max}^{h-1}$ and since no labels with index i change we have that $s'.t_i^{h-1} = s'.nt_i^{h-1} = s'.t_{max}^{h-1}$. Suppose $i = k$. Then $s'.t_k^{h-1} = s'.t_{max}^{h-1}$. From action, $s'.t_k^{h-1} = s.nt_k^{h-1}$ and $s'.nt_k^{h-1} = s.nt_k^{h-1}$. Hence $s'.nt_k^{h-1} = s'.t_{max}^{h-1}$. ■

Claim B.3 *If $s.nt_k^{h-1} \preceq s.t_{max}^{h-1}$ then s' satisfies 2.*

Proof. Suppose $s'.t_i^{h-1} = s'.t_{max}^{h-1}$. Suppose $i \neq k$. From action, $s'.t_i^{h-1} = s.t_i^{h-1}$ and by Claim B.1, $s'.t_{max}^{h-1} = s.t_{max}^{h-1}$. Now by 2 for s, $s.nt_i^{h-1} \succeq s.t_{max}^{h-1}$ and since $s'.nt_i^{h-1} = s.nt_i^{h-1}$, $s'.nt_i^{h-1} \succeq s'.t_{max}^{h-1}$. Suppose $i = k$. By action, $s'.nt_k^{h-1} = s'.t_k^{h-1}$ hence $s'.nt_k^{h-1} = s'.t_{max}^{h-1}$. ■

Claim B.4 *If $s.nt_k^{h-1} \preceq s.t_{max}^{h-1}$ then s' satisfies 3.*

Proof. Since $s'.t_{max}^{h-1} = s.t_{max}^{h-1}$ and since no *nt*-labels change as a result of the action, s' satisfies 3 since s does so. ■

Claim B.5 *If $s.nt_k^{h-1} \succ s.t_{max}^{h-1}$ then $s'.t_{max}^{h-1} = s'.t_k^{h-1}$.*

Proof. By 1 for s there exists $j \neq k$ such that $s.t_j^{h-1} = s.t_{max}^{h-1}$. Consider any $i \neq k$. Then $s.t_i^{h-1} \preceq s.t_j^{h-1}$ and since $s.nt_k^{h-1} \succ s.t_j^{h-1}$ we have by induction hypothesis (5.1) that $s.t_i^{h-1} \prec s.nt_k^{h-1}$. Now since $s'.t_i^{h-1} = s.t_i^{h-1}$ and $s'.t_k^{h-1} = s.nt_k^{h-1}$ we conclude that $s'.t_i^{h-1} \prec s'.t_k^{h-1}$ and from definition of t_{max}^{h-1}, $s'.t_{max}^{h-1} = s'.t_k^{h-1}$. ■

Claim B.6 *If $s.nt_k^{h-1} \succ s.t_{max}^{h-1}$ then s' satisfies 1.*

Proof. From Claim B.5, $s'.t_k^{h-1} = s'.t_{max}^{h-1}$. If there exists $j \neq k$ such that $s'.t_j^{h-1} = s'.t_{max}^{h-1}$ then 1 vacously. Otherwise, from action, $s'.t_k^{h-1} = s'.nt_k^{h-1}$. ■

Claim B.7 *If $s.nt_k^{h-1} \succ s.t_{max}^{h-1}$ then s' satisfies 2.*

Proof. Suppose $s'.t_i^{h-1} = s'.t_{max}^{h-1}$. We first show that $i = k$. Assume for the sake of contradiction that $i \neq k$. From Claim B.5, $s'.t_{max}^{h-1} = s'.t_k^{h-1}$ and by action $s'.t_k^{h-1} = s.nt_k^{h-1}$ and $s'.t_i^{h-1} = s.t_i^{h-1}$. Hence $s.t_i^{h-1} = s.nt_k^{h-1}$. Since by assumption $s.nt_k^{h-1} \succ s.t_{max}^{h-1}$ we have $s.t_i^{h-1} \succ s.t_{max}^{h-1}$ which contradicts the definition of t_{max}^{h-1}. Therefore $i = k$. From action $s'.nt_k^{h-1} = s'.t_k^{h-1}$ and hence $s'.nt_k^{h-1} = s'.t_{max}^{h-1}$. ■

Claim B.8 *If $s.nt_k^{h-1} \succ s.t_{max}^{h-1}$ then s' satisfies* 3.

Proof. Suppose $s'.nt_i^{h-1} \succ s'.t_{max}^{h-1}$. Then by Claim B.5 $s'.nt_i^{h-1} \succ s'.t_k^{h-1}$ and since $s'.t_k^{h-1} = s'.nt_k^{h-1}$, $s'.nt_i^{h-1} \succ s'.nt_k^{h-1}$. Hence $i \neq k$. By action, no nt-labels change and hence $s.nt_i^{h-1} \succ s.nt_k^{h-1}$. By assumption, $s.nt_k^{h-1}$ $\succ s.t_{max}^{h-1}$ and from 1 for s, there exists $j \neq k$ such that $s.t_j^{h-1} = s.t_{max}^{h-1}$. Suppose $s.t_i^{h-1} = s.t_{max}^{h-1}$. From 2 for s, $s.nt_i^{h-1} \succeq s.t_{max}^{h-1}$ and since $s.nt_i^{h-1}$ $\succ s.nt_k^{h-1}$ and $s.nt_k^{h-1} \succ s.t_{max}^{h-1}$ we have that $s.nt_i^{h-1} \succ s.t_{max}^{h-1}$ and by 3 for s, $s.nt_i[h] = 1$. Thus, since $s'.nt_i = s.nt_i$, $s'.nt_i[h] = 1$. Suppose $s.t_i^{h-1} \neq s.t_{max}^{h-1}$. Then $j \neq i$ and by induction hypothesis (5.1) for s we have that $s.nt_i^{h-1} \succ s.t_{max}^{h-1}$ and from 3, $s.nt_i[h] = 1$. From action $s'.nt_i = s.nt_i$ and hence $s'.nt_i[h] = 1$. ■

This ends the proof of Lemma B.2. ■

Lemma B.3 *If $\pi = scan_k$ then s' satisfies* 1, 2 *and* 3.

Proof. The proof follows from the following claims. We assume that $s.t_{max}$ $\neq -$, $k \neq s.i_{max}$, and $s.op_k$ *label* since otherwise no t-labels or nt-labels change and s' satisfies 1, 2 and 3 since by hypothesis s does so.

Claim B.9 *s' satisfies* 1.

Proof. Suppose for all $j \neq i$, $s'.t_j^{h-1} \neq s'.t_{max}^{h-1}$. From action no t-labels change. Hence, for all $j \neq i$, $s.t_j^{h-1} \neq s.t_{max}^{h-1}$ and $s.t_i^{h-1} = s.t_{max}^{h-1}$. This implies that $i = s.i_{max}$ and hence by assumption $i \neq k$. From 1 for s, $s.t_i^{h-1}$ $= s.nt_i^{h-1}$ and since no i labels change, $s'.t_i^{h-1} = s'.nt_i^{h-1}$. ■

Claim B.10 *s' satisfies* 2.

Proof. Suppose $i \neq k$. By action, neither nt_i nor any t-labels change so s' satisfies 2 since s does so. Suppose $i = k$. By action, $s'.nt_i =$ *next-label*$(s.t_{max}, h')$ for some $h' \in \{1, \ldots, n-1\}$ and since $s'.t_{max} = s.t_{max}$, $s'.nt_i =$ *next-label*$(s'.t_{max}, h')$. From definition of *next-label*, $s'.nt_i \succ s'.t_{max}$ hence $s'.nt_i^{h-1} \succeq s'.t_{max}^{h-1}$. ■

Claim B.11 *s' satisfies* 3.

Proof. Suppose $s'.nt_i^{h-1} \succ s'.t_{max}^{h-1}$. Since no t-labels change, $s'.t_{max}^{h-1} =$ $s.t_{max}^{h-1}$ and hence $s'.nt_i^{h-1} \succ s.t_{max}^{h-1}$. Suppose $i \neq k$. Then $s'.nt_i = s.nt_i$ and $s.nt_i^{h-1} \succ s.t_{max}^{h-1}$. By 3 for s, $s.nt_i[h] = 1$ and hence $s'.nt_i[h] = 1$. Suppose $i = k$. Then $s'.nt_i =$ *next-label*$(s.t_{max}, h')$ for some $h' \in \{1, \ldots, n-1\}$. If h' $> h-1$ then by definition of *next-label*, $s'.nt_i^{h-1} = s.t_{max}^{h-1}$ and since $s'.t_{max}^{h-1}$ $= s.t_{max}^{h-1}$, 3 holds vacoulsy in s' in this case. If $h' \leq h-1$ then by definition of *next-label*, $s'.nt_i^{h-1} \succ s.t_{max}^{h-1}$, i.e. $s'.nt_i^{h-1} \succ s'.t_{max}^{h-1}$, and for all $h'' > h'$, $s'.nt_i[h''] = 1$. Now, since $h' \leq h-1$ this implies that $s'.nt_i[h] = 1$. ■

This ends the proof of Lemma B.3. ■

Lemma B.4 *If* $\pi \in \{beginlabel_k, endlabel_k, beginscan_k, endscan_k\}$ *then* s' *satisfies* 1, 2 *and* 3.

Proof. No t-labels or nt-labels change so s' satisfies 1, 2 and 3 since s does so. ■

This ends the proof of Lemma 5.6. ■