Partial State Space Analysis of Safety-Critical Systems

Bernd-Holger Schlingloff

Contents

Chapter 0

Introduction

This book is about bugs. More specifically, it is about software bugs and ways to find or avoid them. We use the term *software* in a very broad sense: it refers to any man-made unambiguous description of some technical or real-life task. For example, a sequence of machine instructions, a message sequence chart, a gate-level diagram of some chip-layout, or a genetically engineered DNA sequence are all software systems. In our world, software is ubiquitous. It is the driving force in most innovations. Many functions formerly performed by elaborate hardware devices are now implemented in software. Therefore, software is becoming more and more complex. Up to 90% of the development cost of a typical embedded system is due to the control software. Usually, many people are involved in the design of a complex software system, and there are many stages in the design process. Because of this complexity, software is notoriously error-prone. In general, each difference between the intended or anticipated and the actual, observable behavior of a software system is an error. Errors can occur due to the wrong formulation of intuitive ideas, due to misunderstandings between people about interfaces, due to wrong assumptions about the used formalisms and machinery, and many more reasons. It is estimated that not a single complex software system today is completely error-free. In contrast to other engineering products such as bridges or automobiles, it is not possible to build software with a tolerance threshold for bugs: a single wrong character can cause a whole complex software system with millions of lines of code to fail. There are no well-established mechanisms which could guarantee that a system is robust against an arbitrary software error.

Since we will only be talking about software, subsequently we just write "system" for "software system". A *safety-critical* system is one in which a

large amount of money depends on the correctness of the software. Usually, each system which can threaten the life or health of people must be considered to be safety-critical. For example, ærospace control computers, train signalling devices, mobile phone chipsets or genetically engineered medicines are mostly safety-critical. More and more safety-critical systems are introduced in daily life. Often these systems are an indispensable part of modern society. In contrast to most desktop programs, where users have become accustomed to the idea that their software contains bugs, in safety-critical applications errors can not be accepted. Thus, today it is one of the most challenging problems to find ways of reducing the amount of errors in safety-critical systems.

There are many facets to this problem: it depends on whether the software under consideration is being developed or already given as a finished result, whether it is designed monolithic or modular, machine-independent or only for a specific hardware, calculating an abstract result or controlling some events in time, and so on. For each of these facets, specific techniques have been investigated and found useful. We focus on so-called *formal methods*. By this, we mean all techniques in which there is a formal level between the intuitive ideas of a developer about the intended behavior of a system and the actual code constituing or running on the target machine. A description is called *formal*, if it is well-formed: built from well-defined symbols according to well-defined rules, with a well-defined meaning. That is, it must be unambiguously decidable whether a specific symbol is allowed in the description, whether a composition of symbols forms a valid description, and whether a specific semantics is the meaning of a particular description. In some sense, each software system is a formal object, since it is an unambiguous denotation of a task. However, some formal descriptions are more *abstract* than others: they use fewer symbols with a more complex semantics to describe the same thing. Usually, the abstract description is also easier to deal with. We refer to the abstract level as the *specification*, and to the concrete level as the *implementation*. In other words, a specification is the formal description of the intended behavior of an implementation. With this terminology, an error is a deviation of the actual behavior of an implementation from its specification.

A system implements a certain behavior by moving through a sequence of *states*. A state is a description of the logical, physical or geometrical relations which form the system in each moment in time. For example, a certain variable assignment and value of the program counter form the state of a program. A certain voltage and current distribution in the wires and latches forms the state of a VLSI circuit. A certain configuration of hormones

and dendrites forms the state of a neuron. The *state space* is the set of all possible states a system can be in, and the possible transitions between these states. To a large extent, the behavior of a system can be predicted by constructing its state space, and errors can be found by comparing the state spaces of the specification and the system. By *state space analysis* we summarize all techniques which have been developed for finding errors in complex systems by looking at its states and transitions.

The word "system" (from the greek $\sigma\upsilon\sigma\tau\eta\mu\alpha$ "sýstēma", composition) refers to something consisting of several parts which are put together. In general, the number of states of a system is the product of the number of states of its components. Thus, if the system has n subsystems, each with up to m states, then it can assume up to m^n states. For example, a program with only 5 integer variables, each 32 bit wide, has more than 10^{50} states. Thus, for any realistic values for n and m this figure is astronomical: it is virtually impossible to traverse all (sequences of) states of a non-trivial system.

Therefore, when analyzing the state space of a system we have to restrict ourselves to selected parts. We call all algorithms which are based on such a selection process as *partial state space analysis* methods. In general, in a partial analysis the state space is separated into into two parts: One part which is examined, and one which is not. Of course, it has to be justified that the selected part in some sense is representative for the complete state space, or at least that there is a high probability of detecting errors within the chosen part.

In practice, there are several approaches for such a partial analysis. One possibility is to disregard all states or transitions between states from which we can predict that they do not contribute in the search for errors. For example, if there is a designated initial state of the system, then we can restrict the search to those states which are reachable from this initial state. Similarly, in many systems it is not necessary to investigate transitions from an error state. More generally, we can build equivalence classes of states and use the class instead of all of its members. For example, if the value of a particular variable is never used, then all states differing only in the value of this variable are equivalent; thus, the value of this variable can be ignored in the analysis. In practice, we can choose a value for the variable in question and check the system's behaviour with this value. If it can not be guaranteed that the value of the specific variable is irrelevant, we could run the system with selected test values for this variable. In such an *automated testing* setup, a safe testing strategy can be used to guarantee that the test coverage steadily increases.

In a similar way, sometimes it is known which part of the state space contains an error. In this case we can restrict attention to that part. This approach is known as *automated debugging*. For example, if we observe an error in the program with a particular combination of input values, then we can check all possible program runs for this particular combination of inputs. If it is unknown which part contains the error, then sometimes it is possible to make a nondeterministic guess, which gives a heuristic search strategy.

A slightly different alternative is to build equivalence classes of *state sequences* instead of equivalence classes of states. We can group all sequences with a similar expected behaviour into the same equivalence class, where "similarity" means invariance under some specification language. Or, we can select certain paths through the state space and analyze only the states on these paths. If the selection is done according to some mathematically sound strategy, again invariance of formulas can be guaranteed.

In practice, often combinations of two or more of the above methods are used. This book contains particular examples of partial state-space analysis methods: In part I, modelling and specification formalisms are introduced. Chapter 1 deals with the modelling of reactive systems by automata theoretic and algebraic techniques. In Chapter 2, we present various specification languages, and relate their semantics to the models of Chapter 1. Then, we describe how models can be transformed such that their "meaning" is preserved. Part II deals with classical verification issues: In Chapter 4, we discuss completeness and decidability issues for temporal logics. Subsequently, we motivate model checking with several practical examples, and give the corresponding algorithms in Chapter 6. Part III introduces real-time verification methods: we define a specification and modelling formalism for real-time and discuss why this is different from untimed verification. In Chapter 8, we then review state space techniques for real time and describe our stubborn algorithms for the logics introduced. Chapters 9 and 10 describe an alternative approach to real-time verification based on the notion of traces and failures from process algebra. In Part IV, more practical work is reconsidered: we describe the application of the formal methods developed in the context of industrial projects. Chapter 11 deals with the development of a tool for partial-order debugging, and finally, Chapter 12 describes the validation of an embedded controller and a protocol stack layer with automated testing.

Part I

Modelling and Specification

Chapter 1

Modelling of Reactive Systems

A *reactive system* [HP85, MP92, MP95] consists of several components which are designed to interact with one another and with the system's environment. In contrast to *functional* systems, where the semantics is given as a function from input to output values, a reactive system is specified by its properties. A *property* is a set of desired behaviors that the system is supposed to possess. From a logical viewpoint, the system is a semantical *model*, and a property is a logical *formula*. Arguing about system correctness, therefore, amounts to determining the *truth of formulas in models*.

In order to be able to perform such a verification, one needs a *specification language* in which the system can be modeled, a *logical language* for the formulation of properties, and a deductive *calculus* or *algorithm* for the verification process. Often, the system to be verified is modeled as a state transition graph, and the properties are formulated in an appropriate temporal logic. If the state space of the system is finite, an automatic search procedure can be used to determine whether or not the state transition graph satisfies the temporal formulas. For infinite systems, interactive verification tools can be used to verify that each state of the system satisfies a certain invariant.

The first part of this book introduces various modelling formalisms for reactive systems, several specification languages for system properties, and shows some connections between modelling and specification. Concrete algorithms for the verification procedure, which make use of the formalisms given here, are presented in part II.

As noted in the introduction, a *system* is something consisting of a num-

ber of subcomponents. There are two fundamental ways of composing subsystems: sequentially (i.e., in time) or parallel (i.e., in space). Whereas sequential composition is well-understood in computer science and an essential part in virtually every existing programming language, parallelism is much less used in today's practice. Parallel processes are meant to be executed during the same time period, thus a need to synchronize their actions arises. If there is complex interaction between the processes, this synchronization can be quite difficult. Therefore, parallel programs are notoriously error-prone and hard to understand. As a consequence, many current programming languages do not allow or support user-defined parallelism. Nevertheless, it is often advantageous to *model* a reactive system as a set of processes running in parallel:

- usually the *functionality* suggests a certain decomposition into modules; sequentialization is not the primary issue in the design;
- certain subcomponents (e.g. hardware components) actually are independent of the rest of the system and, therefore, conceptually parallel, and
- the *environment* can be seen as a process running in parallel to the system

Hence, when defining modelling formalisms for reactive systems we have to consider models of *parallel processes*, and the synchronisation between these processes.

1.1 Parallel Programming Paradigms

A number of different paradigms have emerged in the modelling of parallel programs. When choosing a modelling formalism, one has to decide whether the focus is on the spatial distribution or on the common resources, how the interaction between the components is modelled, which model of time is underlying the system, and how the processes are executed in time.

1.1.1 Modelling of Communication

To regard a set of components as a system only makes sense if there is some kind of interaction between the components. Thus, a fundamental issue is how such interactions can be represented in a modelling formalism.

Message Passing vs. Shared Variables

There are two main paradigms of parallel systems: *distributed* systems, where the subcomponents are seen as spatially apart from each other, and *concurrent* systems, where the subcomponents use common resources such as processor time or memory cells.

Consequently, there are two main paradigms for interaction between parallel processes: via *message passing* (for distributed systems), and via *shared variables* (for concurrent systems).

Of course, there is no clear distinction between distributed and concurrent programs. On one hand, most concurrent systems are also distributed in space. It is not possible to formalise the concept of being spatially apart, since this is dependent on one's own point of view: from the United States, all computers in a local area network in Europe can be regarded as a single system. From the processor's viewpoint, a hard disk controller can be regarded as a remote subsystem. On the other hand, every component of a distributed system shares *some* resource with *some* other component; if it were totally unrelated it would not make sense to regard it as being part of one system.

Consequently, from a certain point of view, passing a message between process A and B can be seen as process A writing into a shared variable which is read by B. On the other side, writing a shared variable can be seen as sending to all other processes which might use this variable the message that its value has changed. In fact, this transition from the message passing paradigm to an implementation via shared variables occurs in every network controller; and the transition from the shared variables paradigm to an implementation via message passing occurs in every distributed cache.

However, different paradigms produce different techniques; many parallel programming languages and many verification systems support only one of these two paradigms.

Synchronous and Asynchronous Communication

A related topic is whether the interaction between parallel components is coordinated or independent. With *synchronous communication*, each partner wishing to interact is blocked until all required partners are willing to participate in the communication. The information is then broadcast to all communication partners. With *asynchronous communication* each process decides whether it wants to wait at a certain point or not; usually some kind of buffering mechanism is used for messages which are not needed immedi-

ately.

Synchronous communication can be seen as a special case of asynchronous communication where the length of each buffer queue is limited to one, and each process decides to wait after writing into or before reading from that queue until the queue is empty or full again, respectively.

Vice versa, a buffer can be seen as a separate process in a synchronous system which is always willing to communicate with other processes. If the size of the buffer is unbounded, the system is not finite state. Even if their size is bounded, the buffers can be the biggest part of the modelling of an asynchronously communicating system.

Examples of synchronous modelling formalisms are (parallel) transition systems, Petri nets, CCS, CSP, and its variants, semaphores and monitors, critical regions and so on. Examples of asynchronous formalisms are protocol specification languages such as SDL and Lotos.

1.1.2 Modelling of Execution

Another issue is the modelling of process execution in time. Of course, a modelling formalism should be able to model not only statical aspects, but also dynamic aspects of the system. Thus, a suitable intuition about the nature of time has to be chosen, and it has to be defined how processes evolve in time.

Flow of Time

In *discrete* processes a computation consists of well-bounded steps, whereas in *continuous* systems the value of state parameters changes gradually as time passes. *Hybrid systems* combine discrete and continuous components. Usually, the model of time which is used in verification is determined by the type of system under consideration.

For parallel systems of discrete processes, there are various ways to model their execution. For *relativistic* time, each process has its own time scale; therefore, an execution is a partial order of events reflecting the causal dependencies between the computation steps. All events belonging to a specific process are linearly ordered by the causality relation. In a *global* time approach, it is assumed that there is a common notion of time for all processes; hence an execution is modelled as a sequence or tree of steps. In such a model it may be necessary to assign a temporal relationship (earlier, later, or at the same time) even to causally independent events.

Alternatives (or nondeterministic choices) can be modelled either with

linear time or *branching* time: In a linear time approach, each moment in time has exactly one successor moment which is realized, whereas in a branching time approach, there can be several successors existing at the same ontological level.

Synchronous, Asynchronous and Interleaved Execution

Similar to the communication mode, also the execution of the processes in time can be coordinated or independent. *Synchronous* processing is characterized by the fact that in each step, every parallel component advances. For example, a circuit in which each gate switches at the pulse of a global clock can be seen as a synchronous system. In contrast, in an *asynchronous* execution in each step an arbitrary (nonempty) subset of all components proceeds. For example, a set of agents working independently and synchronizing via mailboxes is a typical asynchronous system. With synchronous processing, the transition relation of the system is the conjunction of the transition relation of the components, with asynchronous processing it is the disjunction.

If each process can perform an "idle" step at any time ("*stutter*"), then synchronous and asynchronous processing coincides. Both synchronous and asynchronous executions can be implemented by *interleaving*, where in each step at most one process is active. A typical example is a set of threads in a time-sharing operating system on a monoprocessor machine. With interleaving execution, usually some fairness constraints are imposed on the scheduling to ensure that all processes can progress.

1.2 Some Basic Formalisms

In this section, we introduce some concrete formal notations for the modelling of systems. These formalisms are general and flexible enough such that they can implement any of the above paradigms. They are the basis for all subsequent chapters, since the state spaces which are traversed in the verification are defined using the basic formalisms.

1.2.1 Transition Systems and Automata

Assume that Σ is a nonempty finite *alphabet*, i.e., set of *letters*. Often, we will use the special case that $\Sigma \triangleq 2^{\mathcal{P}}$ is the powerset of some finite basic set of *propositions*. An ω-word over Σ is a nonempty (finite or infinite) sequence of letters from Σ. An *ω-language* is any set of ω-words over Σ. Note

that for each nontrivial alphabet (i.e., with more than one letter) there are uncountably many ω-words and languages.
A *transition system* T is a tuple $T \triangleq (\Sigma, S, \Delta, S_0)$, where

- Σ is the *alphabet*,
- S is a nonempty set of *states*,
- $\Delta \subseteq S \times \Sigma \times S$ is the *transition relation*, and
- $S_0 \subseteq S$ is the set of *initial states*.

A *finite transition system (FTS)* is one where S (and, thus also S_0 and Δ) is finite.

Let $\sigma \triangleq (\sigma_1\sigma_2\sigma_3...)$ be an ω-word over Σ. We say that σ is *generated by* the transition system T, if there exists a sequence $s_0, s_1, s_2, ...$ of states in S such that

- $s_0 \in S_0$, and
- for each $i > 0$ it holds that $(s_{i-1}, \sigma_i, s_i) \in \Delta$.

The *generated language* of a transition system T is the set of all ω-words which are generated by T. In some sense, the generated language does not contain information about the internal structure of a transition system. For example, the three transition systems in Fig. 1.1 all generate the same language: The set of all words where each first position (modulo 3) is an a, and the second and third are b and c or c and b, respectively. It is said that by considering the generated language instead of the transition system itself one cannot distinguish between nondeterminism and choice. We will come back to this issue in Section 3.3.

If $\sigma = (\sigma_1, \sigma_2, \sigma_3, ...)$ is an ω-word, then each finite nonempty subsequence $\sigma^{..i} \triangleq (\sigma_1, ..., \sigma_i)$ is called a *prefix* of σ. It follows immediately from the definition that for each σ which is generated by the transition system T all prefixes $\sigma^{..i}$ of σ are also generated by T. In other words,

Lemma 1.1 *Transition systems generate prefix closed ω-languages.*

In some applications, this prefix-closure is an undesired property. A state s is called *terminal*, if it has no outgoing transition to other states, i.e., $\Delta \cap (\{s\} \times \Sigma \times S) = \emptyset$. We say that σ is *weakly fair generated* by the transition system T, if in addition to the two requirements above the following holds:

- if σ is finite with last letter σ_n, then s_n is terminal.

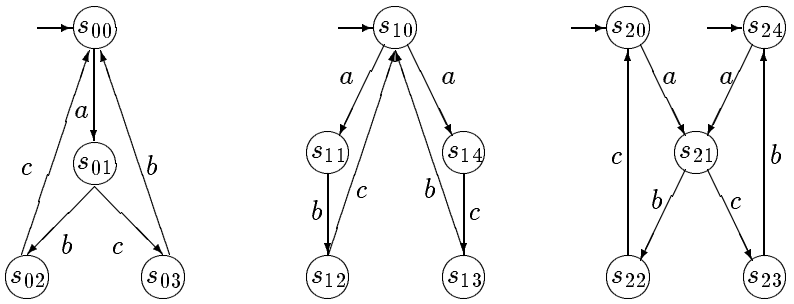


Figure 1.1: Three equivalent transition systems

This *weak fairness constraint* guarantees that weakly fair generated words represent maximal paths through the transition graph. The weakly fair generated language of a transition system T is the set of all ω-words which are weakly fair generated by T. In the examples from Fig. 1.1, the weakly fair language contains only infinite words, since there are no terminal states.

A generalization of the weak fairness condition leads to the notion of *ω-automaton* or *fair transition system* for the alphabet Σ. It is defined like a usual nondeterministic automaton (see any standard book, e.g., [HU79]) with an additional recurrence set ("fairness constraint") as a tuple $A \triangleq (S, \Delta, S_0, S_{acc}, S_{rec})$, where

- S is a set of states,
- $\Delta \subseteq S \times \Sigma \times S$ is the transition relation,
- $S_0 \subseteq S$ is the set of initial states,
- $S_{acc} \subseteq S$ is the set of accepting states (for finite words), and
- $S_{rec} \subseteq S$ is the set of recurring states (for infinite words).

A *Büchi-automaton* is a finite ω-automaton, that is, a fair transition system where the set S of states is finite. A (finite or infinite) nonempty word $\sigma \triangleq (\sigma_0, \sigma_1, ...)$ is *accepted* by an automaton $(S, \Delta, S_0, S_{acc}, S_{rec})$, if there is a function ρ assigning to any letter σ_i a state $\rho(\sigma_i) \in S$ of the automaton such that

- $\rho(\sigma_0) \in S_0$,
- for all $0 \leq i < n$, $(\rho(\sigma_i), \sigma_i, \rho(\sigma_{i+1})) \in \Delta$,

- if σ is finite with last letter w_n, then $(\rho(\sigma_n), \sigma_n, s) \in \Delta$ for some $s \in S_{acc}$, and
- if σ is infinite, then $\mathit{inf}(\rho) \cap S_{rec} \neq \emptyset$, where $\mathit{inf}(\rho)$ is the set of states that appear infinitely often in the range of ρ.

That is, reading finite sequences the automaton must end in an accepting state, and for infinite sequences at least one recurring state must be selected infinitely often. For alternative acceptance conditions, see [Tho90].

A transition system can be regarded as a special fair transition system where $S_{acc} = S_{rec} = S$, and a weakly fair transition system is one where $S_{rec} = S$ and $S_{acc} = \{s \mid \forall a, s'(s, a, s') \notin \Delta\}$. That is, in a weakly fair transition system all states are recurring, and states are accepting iff they are terminal. A classical *finite automaton* has $S_{rec} = \emptyset$; it does not accept any infinite word. If $S_{acc} = \emptyset$, then the automaton accepts only infinite words.

As an example of a Büchi-automaton, consider Figure 1.2. In the drawing, we mark initial states by ingoing arrows, accepting states by outgoing arrows and recurring states by double circles.

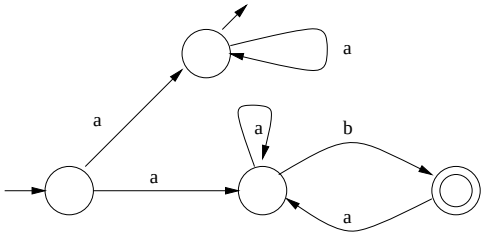


Figure 1.2: A Büchi automaton

This automaton accepts all finite words containing only the letter a, and all infinite words containing infinitely many b's (separated by a's).

To be able to model parallelism within state machines, it is advantageous to introduce special composition operators. A *parallel transition system* is a tuple $T \triangleq (T_1, ..., T_n)$ of transition systems, such that their sets of states are pairwise disjoint ($S_i \cap S_j = \emptyset$). We also write $(T_1 \parallel ... \parallel T_n)$ for the parallel composition of transition systems. The *global* transition system T *associated with* a parallel transition system $(T_1 \parallel ... \parallel T_n)$ is defined by $T \triangleq (\Sigma, S, \Delta, S_0)$, where

- $\Sigma \triangleq \bigcup \Sigma_i$

- $S \triangleq S_1 \times \cdots \times S_n$
- $S_0 \triangleq S_{10} \times \cdots \times S_{n0}$, and
- $((s_1, ..., s_n), a, (s'_1, ..., s'_n)) \in \Delta$ iff for all T_i
  - if $a \in \Sigma_i$, then $(s_i, a, s'_i) \in \Delta_i$, and
  - if $a \notin \Sigma_i$, then $s_i = s'_i$

Thus, in a parallel transition system synchronisation between components is by the common alphabet. In general, the size of the state space of the global transition system is the product of the sizes of all parallel components.

A popular slogan says that "composition should be conjunction". In our context, the slogan means that a composed system should accept a word if and only if each component accepts it. Since the composed system can have a larger alphabet than the components, this property can not be true in general for the above parallel composition operation. However, with an appropriate restriction the slogan holds. The *projection* $\sigma \restriction_\Sigma$ of the word $\sigma \triangleq (\sigma_1, \sigma_2, \sigma_3...)$ over some alphabet onto some other alphabet Σ is the word $(\xi_1, \xi_2, \xi_3...)$, where $\xi_k \triangleq \sigma_m$ and m is the minimal index of σ such that $\sigma^{..m}$ contains k occurrences of letters from Σ. As an example, if $\sigma \triangleq abaacaba...$ and $\Sigma \triangleq \{b, c\}$, then $\sigma^{..6}$ contains 2 occurrences of letters from Σ, and $\sigma^{..7}$ contains 3 such occurrences, hence $\xi_3 = \sigma_7$.

An equivalent recursive definition of this operation is

$$\sigma \restriction_\Sigma \triangleq \begin{cases} (\sigma_1, \sigma'_2, \sigma'_3, ...), & \text{if } \sigma_1 \in \Sigma \\ (\sigma'_2, \sigma'_3, ...), & \text{else} \end{cases}$$

where $(\sigma'_2, \sigma'_3, ...) = (\sigma_2, \sigma_3, ...) \restriction_\Sigma$. Note that $\sigma \restriction_\Sigma$ can be empty or finite even if σ is infinite. The extension of the notion of projection to sets of words is straightforward: $\mathcal{L} \restriction_\Sigma \triangleq \{\sigma \mid \exists \sigma' \in \mathcal{L}, \sigma = \sigma' \restriction_\Sigma\}$

Lemma 1.2 *Assume that $\sigma \restriction_{\Sigma_i}$ is nonempty for $1 \leq i \leq n$. Then it holds that σ is generated by $(T_1 \parallel ... \parallel T_n)$ iff $\sigma \restriction_{\Sigma_i}$ is generated by T_i for all $i \in \{1, ..., n\}$.*

PROOF: Even though the statement is rather obvious, the proof is somewhat tedious. For one direction, assume that $\sigma \restriction_{\Sigma_i} \triangleq (\xi_{i,1}, \xi_{i,2}, ...)$ is generated by T_i for all $1 \leq i \leq n$. That is, for each i there is a sequence $(s_{i,0}, s_{i,1}, ...)$ of states such that $(s_{i,m-1}, \xi_{i,m}, s_{i,m}) \in \Delta_i$. We construct a sequence $((s'_{i,m}))$ of tuples of states generating σ: $s'_{i,0} \triangleq s_{i,0}$, and if $\sigma_m \in \Sigma_i$ and $s'_{i,m-1} = s_{i,j}$

then $s'_{i,m} \triangleq s_{i,j+1}$, else $s'_{i,m} \triangleq s'_{i,m-1}$. The result is a sequence of states from $(T_1 \parallel ... \parallel T_n)$ generating σ.

For the other direction, assume that $\sigma \triangleq (\sigma_1, \sigma_2, ...)$ is generated by $(T_1 \parallel ... \parallel T_n)$. Then there is a sequence $((s_{j,k})) \triangleq ((s_{1,0}, ..., s_{n,0}), (s_{1,1}, s_{n,1}), ...)$ of tuples of states such that $((s_{j,k-1}), \sigma_k, (s_{j,k})) \in \Delta$. We fix some $i \leq n$ and show that $(\xi_1, \xi_2, ...) \triangleq \sigma \restriction_{\Sigma_i}$ is generated by T_i. Let $s_0 \triangleq s_{i,0}$ and for all $0 < k \leq |\sigma \restriction_{\Sigma_i}|$, let $s_k \triangleq s_{i,m}$, where again m is the smallest index for which $\sigma^{..m}$ contains k occurrences of letters from Σ. We now show (*): If $m > 0$ is *any* index such that $\sigma^{..m}$ contains k occurrences of letters from Σ, then $s_{i,m} = s_k$. If $\sigma_m \in \Sigma$, then m is minimal and (*) follows from the definition of s_k. If $\sigma_m \notin \Sigma_i$, then $\sigma^{..m-1}$ also contains k occurrences of letters from Σ_i, and we can inductively assume that $s_{i,m-1} = s_k$. According to the definition of Δ, in this case $s_{i,m-1} = s_{i,m}$. Hence $s_{i,m} = s_k$, which proves (*).

Assume now that ξ_k is the m-th letter of σ, i.e., $\sigma_m \in \Sigma_i$ and $\sigma^{..m-1}$ contains $k-1$ occurrences of letters from Σ_i. From (*) we know that $s_{i,m-1} = s_{k-1}$ and $s_{i,m} = s_k$. Since $\sigma_m \in \Sigma_i$, from the definition of Δ it follows that $(s_{i,m-1}, \sigma_m, s_{i,m}) \in \Delta_i$. Hence $(s_{k-1}, \xi_k, s_k) \in \Delta_i$, which was to be shown. □

The definition of a parallel composition of Büchi automata is not so obvious as for transition systems. We want to have a similar property as above, namely that the composed automaton accepts any word iff each individual automaton accepts the projection of this word onto its own alphabet. Consider the case of two Büchi automata A_1 and A_2 running in parallel. When processing an infinite word w, it may happen that each automaton is infinitely often in a recurring state, but A_1 and A_2 are never both in a recurring state at the same instant. Thus, the global automaton has to keep track of whether both components repeat to visit recurring states. It uses two additional memory bits x_1 and x_2 to record whether A_1 and A_2 have been in a recurring state. That is, x_1 is set whenever A_1 enters a recurring state, and vice versa for x_2 and A_2. The memory bits are reset whenever both are set. The automaton accepts if infinitely often a state is reached where both bits are set.

Another difficulty is that the projection of an infinite word onto a smaller alphabet may be finite. In such a case, some component of the global automaton would remain forever in the same state s. Assume that s is a terminal state (without outgoing transitions). If s is accepting, then being stuck at s is similar to visiting a recurring state infinitely often. If s is nonaccepting, then remaining forever at s means that no recurring state can be visited infinitely often. Thus, in this case s can be treated as if it belongs to the recurring states in the composition iff it is an accepting

state of the component. However, not every accepting state of a component automaton can be defined to be recurring in the composition: There may be runs with infinitely many accepting but only finitely many recurring states. For each automaton $A = (S, \Delta, S_0, S_{acc}, S_{rec})$, define the automaton $A' \triangleq (S', \Delta', S_0, S'_{acc}, S_{rec})$ by $S' \triangleq (S \cup S'_{acc})$, $S'_{acc} \triangleq \{s' \mid s \in S_{acc}\}$, and $\Delta' \triangleq \Delta \cup \{(s_1, a, s'_2) \mid (s_1, a, s_2 \in \Delta\}$. Then A and A' accept the same language, and in A' each accepting state is terminal. With this definition, the global Büchi-automaton associated with a parallel composition $(A_1 \parallel ... \parallel A_n)$ of Büchi-automata can be defined by:

- $\Sigma \triangleq \bigcup \Sigma_i$
- $S \triangleq S'_1 \times \cdots \times S'_n \times \{0,1\}^n$
- $S_0 \triangleq S_{10} \times \cdots \times S_{n0} \times \{(0, ..., 0)\}$,
- $S_{acc} \triangleq S_{1,acc} \times \cdots \times S_{n,acc} \times \{0,1\}^n$,
- $S_{rec} \triangleq S'_1 \times \cdots \times S'_n \times \{(1, ..., 1)\}$, and
- $((s_1, ..., s_n, (x_1, ..., x_n)), a, (s'_1, ..., s'_n, (y_1, ..., y_n))) \in \Delta$ iff for all i
  - if $a \in \Sigma_i$, then $(s_i, a, s'_i) \in \Delta'_i$,
  - if $a \notin \Sigma_i$, then $s_i = s'_i$, and
  - if $(x_1, ..., x_n) = (1, ..., 1)$ then $(y_1, ..., y_n) = (0, ..., 0)$ else
  - if $s'_i \in S'_{acc,i}$ or $a \in \Sigma_i$ and $s'_i \in S_{rec,i}$ then $y_i = 1$ else $y_i = x_i$.

Assume that $A_1, ..., A_n$ are Büchi automata and that each $\sigma \restriction_{\Sigma_i}$ is nonempty. Then again, σ is accepted by $(A_1 \parallel ... \parallel A_n)$ iff $\sigma \restriction_{\Sigma_i}$ is accepted by A_i for all $i \in \{1, ..., n\}$. The proof is left as an exercise.

1.2.2 ω-Regular Expressions and Process Algebras

Besides automata, *regular expressions* are widely used for the definition of formal languages. They provide a textual representation of transition systems which is more convenient for processing by a machine. Regular expressions were invented in the 1950's and had a comeback with the advent of process algebra for modelling of concurrency in the 1980's. Since then, they have become the basis for many modern specification languages.

ω-Regular Expressions

In the case of finite words, it is well known that regular expressions are as expressive as finite automata. The notion of regular expression can be easily extended to infinite words: an *ω-regular expression* can be built from the usual operators for regular expressions, with an additional operation denoting infinite repetition of a subexpression. Formally, the syntax of ω-regular expressions is given as follows:[1]

$$\omega\mathbf{R} ::= a \mid \varepsilon \mid (\omega\mathbf{R} + \omega\mathbf{R}) \mid (\omega\mathbf{R};\ \omega\mathbf{R}) \mid \omega\mathbf{R}^{+} \mid \omega\mathbf{R}^{\omega}$$

In other words,

- Every letter from the alphabet is an ω-regular expression.
- ε is an ω-regular expression.
- If t_1 and t_2 are ω-regular expressions, then so are $(t_1 + t_2)$, and $(t_1; t_2)$.
- If t is an ω-regular expression, then so is t_1^+ and t^ω, and
- nothing else is an ω-regular expression.

Every ω-regular expression defines an ω-language: the letter $a \in \Sigma$ defines $\{(a)\}$, i.e., a one-word language (one-element set) consisting of a one-letter word (one-element sequence). ε denotes the empty language, and $(t_1 + t_2)$, $(t_1; t_2)$ and t_1^+ denote union, sequential composition and finite iteration of languages. t^ω denotes the language of all words consisting of an infinite concatenation of words from t. A language is called ω-regular if it can be defined by an ω-regular expression.

As an example for an ω-regular expression, the language of the Büchi automaton in Figure 1.2 can be defined by $a^+ + (a^+; b)^\omega$. In general, for Büchi automatonwe can construct such an ω-regular expression and vice versa; Büchi-automata can define all and only ω-regular languages.

Theorem 1.3 *ω-regular expressions and Büchi-automata are of equal expressive power.*

[1] A note on parenthesis: To be uniquely parseable, formulas and expressions which contain nested binary infix operator require the use of parenthesis. However, if a binary infix operator is associative, for nested occurrences of this operator within itself parenthesis can be omitted. In expressions containing different binary operators we declare that ";" binds stronger than "+" and omit parenthesis whenever appropriate.

PROOF: The proof of this statement is similar as for automata on finite words: For one direction, we have to show that the Büchi acceptance condition can be captured by an appropriate regular expression. Let $L(s_i, s_j)$ be a regular expression for the language of finite nonempty words sending an automaton from state s_i into state s_j. Then the ω-regular expression associated with any Büchi-automaton is

$$\Sigma\{L(s_0, s) \mid s_0 \in S_0,\ s \in S_{acc}\} + \Sigma\{L(s_0, s); L(s, s)^\omega \mid s_0 \in S_0,\ s \in S_{rec}\}$$

For the other direction it must be shown that Büchi-automata are closed under single letters, the empty language, union, concatenation, and finite and infinite repetition. All of these constructions are straightforward extensions of the appropriate constructions for automata on finite words. □

The automaton resulting from this proof is highly nondeterministic. An automaton is called *deterministic*, if its transition relation is a function $\Delta : S \times \Sigma \to S$. A deterministic automaton has a unique run on any given word. For each nondeterministic finite automaton on finite words an equivalent deterministic one is given by the well known powerset construction of Rabin and Scott [HU79]. In contrast, for nondeterministic Büchi-automata it is not always possible to construct an equivalent deterministic one.

Lemma 1.4 *Nondeterministic Büchi automata are more expressive than deterministic ones.*

PROOF: For example, consider the language $\mathcal{L}$ of all infinite words over $\{a, b\}$ containing only finitely many letters a. This language is defined by the ω-regular expression $((a + b)^+; b^\omega)$. A nondeterministic automaton for this language is given in Figure 1.3. However, there is no determinictic Büchi-automaton defining $\mathcal{L}$: Assume for contradiction that $\mathcal{L}$ is the language of $\mathcal{A}$. Then $\mathcal{A}$ must accept $(\sigma; a; b^\omega)$ for any finite word σ. In particular, from any reachable state some recurring state is reached by an a-transition and then a finite number of b-transitions. Let m be the maximum of these numbers. Then, from any reachable state, some recurring state is visited when processing $(a; b^m)$. Therefore, in the run of $\mathcal{A}$ on the word $(a; b^m)^\omega$ infinitely often recurring states are visited. Since there are only finitely many states, some recurring state must be visited infinitely often. Thus, the word $(a; b^m)^\omega$ is also accepted by $\mathcal{A}$. This is a contradiction, since it is not in $\mathcal{L}$. □

Process Algebras

Parallel composition operators for regular expressions have been widely studied in the context of process algebras [Hoa85, Ros98, Sch00]. Let Σ be a

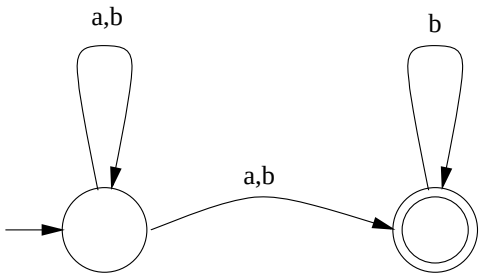


Figure 1.3: Büchi automaton for "finitely many a"

finite alphabet not containing the special letters ✓ and τ. Furthermore, assume that we are given a countable set $\mathcal{Q}$ of *process variables*. A *process algebraic term* (in a CSP-like language) is given by the following syntax:

$$\begin{aligned}\mathbf{PA} ::= {} & STOP \mid SKIP \mid \mathcal{Q} \mid (a \longrightarrow \mathbf{PA}) \mid \\ & (\mathbf{PA} \,[]\, \mathbf{PA}) \mid (\mathbf{PA};\ \mathbf{PA}) \mid (\mathbf{PA} \parallel_A \mathbf{PA}) \mid \nu\mathcal{Q}\ \mathbf{PA} \mid \mathbf{PA}\{R\}\end{aligned}$$

In this clause, $STOP$ and $SKIP$ are process constants, a is a metavariable for *process events* (letters from the alphabet Σ), A is a metavariable for a set of letters, and R is a metavariable for a binary relation between letters.

$STOP$ stands for the *terminated* process which cannot be continued (i.e., in a terminal state), $SKIP$ is a process which is *completed* and waiting for continuation, and $(a \longrightarrow t)$ is a process which can perform event a and then behave like process t. *External choice* is denoted by $(t_1 \,[]\, t_2)$: depending on the input, this process can either execute t_1 or t_2. The *sequential composition* of t_1 and t_2 is denoted by $(t_1;\ t_2)$. Parallel composition of processes $(t_1 \parallel_A t_2)$ is slightly more general than parallel composition of automata: whereas automata have to synchronize on all events in the common alphabet, here the set A of letters on which synchronization occurs can be explicitly defined. The ν operator[2] denotes *recursion*: The process $\nu q\ t$ behaves like t, where each occurence of q is replaced by the process $\nu q\ t$. Finally, $t\{R\}$ is the *renaming* of t by $R \subseteq \Sigma \times (\Sigma \cup \{\tau\})$: This is a process which behaves like t but where each event $a \in \Sigma$ in the execution is replaced by some event b such that $(a, b) \in R$.

[2] In the literature, the symbol μ has been used for this operator. Since we are working in a space of finite and infinite words, it denotes a greatest fixed point. Therefore we prefer to use the symbol ν rather than μ

- $(SKIP, \checkmark, STOP) \in \Delta$.
- If $a \in \Sigma$, then $((a \longrightarrow t), a, t) \in \Delta$.
- If $a \in \Sigma \cup \{\checkmark\}$ and $(t_1, a, t_1') \in \Delta$, then
  $((t_1 \,[]\, t_2), a, t_1') \in \Delta$ and $((t_2 \,[]\, t_1), a, t_1') \in \Delta$.

  If $(t_1, \tau, t_1') \in \Delta$, then
  $((t_1 \,[]\, t_2), \tau, (t_1' \,[]\, t_2)) \in \Delta$ and $((t_2 \,[]\, t_1), \tau, (t_2 \,[]\, t_1')) \in \Delta$.
- If $a \in \Sigma \cup \{\tau\}$ and $(t_1, a, t_1') \in \Delta$, then $((t_1;\ t_2), a, (t_1';\ t_2)) \in \Delta$.

  If $(t_1, \checkmark, t_1') \in \Delta$, then $((t_1;\ t_2), \tau, t_2) \in \Delta$.
- If $a \in A \cup \{\checkmark\}$ and $(t_1, a, t_1') \in \Delta$ and $(t_2, a, t_2') \in \Delta$, then
  $((t_1 \parallel_A t_2), a, (t_1' \parallel_A t_2')) \in \Delta$.

  If $a \notin A \cup \{\checkmark\}$ and $(t_1, a, t_1') \in \Delta$, then
  $((t_1 \parallel_A t_2), a, (t_1' \parallel_A t_2)) \in \Delta$ and $((t_2 \parallel_A t_1), a, (t_2 \parallel_A t_1')) \in \Delta$.
- If $a \in \Sigma \cup \{\tau, \checkmark\}$ and $(t, a, t') \in \Delta$, then
  $((\nu q\ t), a, (t'\{q := \nu q\ t\})) \in \Delta$.
- If $a \in \Sigma$, $(a, b) \in R$ and $(t, a, t') \in \Delta$, then $(t\{R\}, b, t'\{R\}) \in \Delta$.

  If $a \in \{\tau, \checkmark\}$ and $(t, a, t') \in \Delta$, then $(t\{R\}, a, t'\{R\}) \in \Delta$.

Table 1.1: Operational Semantics for Process Algebra

The semantics of process algebra is often defined in an *operational* style: With each term t_0, a transition system over the alphabet $\Sigma \cup \{\checkmark, \tau\}$ is associated as follows. The set of states is the set of all well-formed terms. The initial state is the term t_0 itself. The transition relation is defined by the set of rules given in Table 1.1. In this table, $a \in \Sigma \cup \{\tau, \checkmark\}$, t, t_1, t_2, t_1', t_2' are terms from **PA**, and $t\{q := t'\}$ denotes the term t where each free occurrence of variable $q \in \mathcal{Q}$ is replaced by the term t'.

Usually, in process algebras like CSP some more operators are provided which can be defined with the above set. *Hiding* $(t \setminus A)$ of a set of events in a process is just a special case of renaming, where all $a \in A$ are mapped to τ:

- $t \setminus A \triangleq t\{R\}$, where $(a,b) \in R$ iff $a \in A$ and $b = \tau$ or $a \notin A$ and $b = a$.

An operational characterization for hiding is that if $(t,a,t') \in \Delta$ and $a \in A$, then $(t \setminus A, \tau, t' \setminus A) \in \Delta$, and if $a \notin A$, then $(t \setminus A, a, t' \setminus A) \in \Delta$.

Slightly different from hiding is the *restriction* operator $(t|_A)$: Here the process can perform only actions from $A \cup \{\tau, \checkmark\}$, all other actions are blocked:

- $t|_A \triangleq t\{R\}$, where $(a,b) \in R$ iff $a \in A$ and $a = b$ (i.e., R is the restriction of the identity relation to A).

Operationally, $(t|_A, a, t'|_A) \in \Delta$ if $(t,a,t') \in \Delta$ and $a \in A \cup \{\tau, \checkmark\}$.

The *internal choice* operator $(t_1 \sqcap t_2)$ is characterized by the two operational axioms $((t_1 \sqcap t_2), \tau, t_1) \in \Delta$ and $((t_1 \sqcap t_2), \tau, t_2) \in \Delta$. It can be defined by

- $(t_1 \sqcap t_2) \triangleq ((a \longrightarrow t_1)\ [\,]\ (a \longrightarrow t_2)) \setminus \{a\}$ for some a which is not contained in t_1 and t_2.

The *interleaving* operator $(t_1 \,|||\, t_2)$ is just parallelism with no synchronization at all. It allows each process to operate independently, synchronizing only on $\checkmark$.

- $(t_1 \,|||\, t_2) \triangleq (t_1 \overset{\|}{_\emptyset}\, t_2)$

A more interesting operator is the *binary alphabetized parallel* $(t_1\ {}_A||_B\ t_2)$ with two alphabets A and B, such that the first process is restricted to A, the second is restricted to B, and both have to synchronize on the intersection of A and B. The definition of this operator is

- $(t_1\ {}_A||_B\ t_2) \triangleq (t_1|_A\ {}_{A\cap B}\overset{\|}{}\ t_2|_B)$

It follows the two operational rules

1. For $a \in (A \cup \{\tau\} \setminus B)$, if $(t_1, a, t_1') \in \Delta$ then $(t_1\ {}_A||_B\ t_2), a, (t_1'\ {}_A||_B\ t_2)) \in \Delta$ and $(t_2\ {}_A||_B\ t_1), a, (t_2\ {}_A||_B\ t_1')) \in \Delta$.
2. For $a \in ((A \cap B) \cup \{\checkmark\})$, if $(t_1, a, t_1') \in \Delta$ and $(t_2, a, t_2') \in \Delta$, then $((t_1\ {}_A||_B\ t_2), \tau, (t_1'\ {}_A||_B\ t_2')) \in \Delta$.

An ω-word σ over Σ is called a *trace* of a process algebraic term, if there exists a word σ' which is generated by the associated transition system such that $\sigma = \sigma' \restriction_\Sigma$. A *failure* of t is a tuple (σ, a), such that σ is a finite trace $(\sigma_0, ..., \sigma_n)$ of t, but σ extended by a, i.e., $(\sigma_0, ..., \sigma_n, a)$, is not amongst t's

traces. We will come back to this notion and to different alternatives in the context of real-time conformance checking in Chapter 9.

In process algebra, there is another frequently used notion: A *divergence* of t is a trace σ such that some prefix σ' of σ exists which is the projection of a generated word σ'' ($\sigma' = \sigma'' \restriction_\Sigma$), such that $(\sigma''; \tau^\omega)$ is also amongst t's generated words. That is, a divergence is a trace such that after some initial actions, the system could go into an infinite loop of internal actions. Since we will be using process algebraic terms mostly in the context of real time, the notion of divergence is of minor importance in this book.

1.2.3 Nets, Programs, I/O systems

Some more advanced basic modelling formalisms are Petri nets, programs using shared variables, and automata distinguishing between input and output. Each of these formalisms has its own specific advantage and will be used in particular contexts only.

Elementary Petri Nets

Whereas in parallel transition systems and automata the number of parallel components is statically fixed, in process algebras the possibility of arbitrarily nesting parallelism and recursion allows to define processes with infinitely many parallel subtasks. Hence, in general the state space of a process algebraic term is infinite. Sometimes it is desirable to model process forking and joining within a finite state formalism. An *elementary Petri net* is a tuple $N \triangleq (P, T, F, m_0)$, where

- P is a finite set of *places*,
- T is a finite set of *transitions* ($P \cap T = \emptyset$),
- $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*, and
- $m_0 \subseteq P$ is the *initial marking* of the net.

A *marking* m of the net is any subset of P. By $\bullet t \triangleq \{p \mid (p,t) \in F\}$ and $t\bullet \triangleq \{p \mid (t,p) \in F\}$ we denote the *preset* and the *postset* of transition t, respectively. A transition t is *enabled* at marking m if $\bullet t \subseteq m$ (all its input places are occupied at m) and $t\bullet \cap m \subseteq \bullet t$ (all its output places are empty at m, or they are also input places). Marking m' is the *result of firing* transition t from marking m, if t is enabled at m and $m' = (m \setminus \bullet t) \cup t\bullet$.

A *firing sequence* is a finite or infinite sequence of markings and transitions $(m_0, t_1, m_1, t_2, ...)$ such that m_0 is the initial marking and each m_{i+1} is

the result of firing t_{i+1} from m_i. A firing sequence is *maximal*, if it is either infinite, or ends in a marking where no transition is enabled. To define the *language* of an elementary Petri net we assume that $\mathcal{L}$ is a *labelling function* from transitions $t \in T$ into some finite alphabet Σ. An ω-word $(\sigma_1, \sigma_2, ...)$ is in the language of net N if there exists a firing sequence $(m_0, t_1, m_1, t_2, ...)$ such that $\mathcal{L}(t_i) = \sigma_i$. In the *weakly fair* language, a maximal firing sequence is required.

For every elementary Petri net there is an associated (weakly fair) finite transition system with the same (weakly fair) language: The alphabet is the set of transitions, the state set is the set of markings, the initial state is the initial marking, and $(m, a, m') \in \Delta$ iff m' is the result of firing t from m and $\mathcal{L}(t) = a$. The number of states in this FTS is exponential in the number of places of the net. In other words, elementary Petri nets can be exponentially more concise than finite transition systems. Vice versa, every finite parallel transition system is a special elementary Petri net with only one token in each reachable marking. In general, it is hard to find an elementary Petri net with a minimal number of places and transitions which is equivalent to a given transition system.

Shared Variables Programs

All of the above finite state modelling formalisms are centered around the idea of control flow. For the modelling of data objects, either the formalisms must be extended, or the data has to be coded into the control structure. In applications which are highly data-dependent, this is not convenient. In contrast, in a von-Neumann-computer the location of control is just the value of a special variable, the program counter. Thus, this architecture is very much data oriented. If all data items are from a finite domain, then object-oriented programs can be seen as yet another specification formalism.

A *shared variables program* is a tuple $P \triangleq (V, D, T, s_0)$, where

- $V = (v_1, ..., v_n)$ is a set of *program variables*,
- $D = (D_1, ..., D_n)$ is a tuple of corresponding finite *domains* $D_i = \{d_{i1}, ..., d_{im_i}\}$
- $T \subseteq D \times D$ is a *transition relation*, and
- $s_0 = (d_{11}, ..., d_{n1})$ is the *initial state*.

A *state* of a shared variables program is a tuple $(d_1, ..., d_n)$, where each $d_i \in D_i$. Thus the number of states in a shared variables program is the product of the size of all domains.

A common way to define the transition relation T is by a propositional formula φ_T with the set of atomic proposition $\mathcal{P} = \{(x = y) \mid x, y \in (V \cup V' \cup \bigcup D_i)\}$, where $V' = \{v'_1, ..., v'_n\}$. (For a formal definition of propositional formulas, see Section 2.1.) If $s = (d_1, ..., d_n)$ and $s' = (d'_1, ..., d'_n)$, then $(s, s') \in T$ iff the formula φ_T is true with v_i interpreted as d_i and v'_i interpreted as d'_i.

For every elementary Petri net or finite parallel transition system there is an equivalent shared variables program of the same order of size. The translation in the other direction requires the evaluation of propositional formulas and thus can involve an exponential blowup. Using relational semantics, a shared variables program can be obtained for almost all other models for concurrency. Therefore, shared variable programs are among the most flexible from all basic formalisms introduced so far.

Input-Output Systems

For several purposes, it is of advantage to distinguish between input signals a system receives from the environment, and output signals, which it can communicate to the environment. Such a distinction allows a hierarchical composition of modules, and it is the basis for automated testing approaches.

I/O-automaton were first defined in [Lyn88, LT89] and used in [GL00] for the development of distributed systems. The ideas developed in these papers, however, can be applied to any of the basic formalisms from above.

An *action alphabet* is an alphabet Σ which is partitioned into Σ^{in}, Σ^{out}, and Σ^{int} of *input*, *output* and *internal* actions. Elements of $\Sigma^{in} \cup \Sigma^{out}$ are called *external actions*, and elements of $\Sigma^{out} \cup \Sigma^{int}$ are called *locally controlled actions*. The idea is that a system has control over its output and internal actions and no control over the inputs which are provided. An automaton or transition system over an action alphabet is *input-enabled*, if for every $s \in S$ and every $\sigma \in \Sigma^{in}$ there exists and s' such that $(s, \sigma, s') \in \Delta$. An *I/O-automaton* is an input-enabled automaton together with a special fairness condition. Again, since we will be using I/O-automata only in the context of real time, the fairness condition is not important in this book.

A *trace* of an I/O-automaton is the projection of a generated word onto the external actions. More generally, an *I/O-module* or *canonical trace structure* is a tuple $M \triangleq (\Sigma^{in}, \Sigma^{out}, \mathcal{T})$, where Σ^{in} is the set of *input actions*, Σ^{out} is the set of *output actions* ($\Sigma^{in} \cap \Sigma^{out} = \emptyset$), and $\mathcal{T}$ is a set of words (traces) over $\Sigma \triangleq \Sigma^{in} \cup \Sigma^{out}$, such that for each $\sigma \in \mathcal{T}$, $k \geq 1$ and $\sigma_i \in \Sigma^{in}$ it holds that $(\sigma^{\cdot\cdot k}; \sigma_i) \in \mathcal{T}$. An I/O-module in some sense represents the externally observable behavior of a system.

We now consider the hierarchical composition of I/O-automata and I/O-modules. Composition is done by identifying actions of different components by name. Thus, whenever one components outputs the action a, all other components having a in its input action must also perform an a. This is similar to the parallel composition of automata. However, since each component can determine when one of its locally controlled actions occurs, there is a side condition that these do not interfere. In particular, no action may appear as an output of two different components, and no internal action of one component can be in the alphabet of another one. Intuitively, the value of a locally controlled action in a composed system must be determined by one and only one of its components. It is allowed that I/O-modules have common inputs, and that an output of one I/O-module appears as input of (several) other I/O-modules. Formally, a set $\{\Sigma_i\}$ of action alphabets is called *compatible*, if for all $i \neq j$

- $\Sigma_i^{out} \cap \Sigma_j^{out} = \emptyset$, and
- $\Sigma_i^{int} \cap \Sigma_j = \emptyset$.

Trivially, by appropriate renaming of symbols any set of I/O-modules can be made compatible. Subsequently, whenever we consider sets of I/O-modules, we assume that they are compatible. If $M_C = \{M_1, \cdots, M_n\}$ is a (compatible) set of I/O-modules, then we define the *composed* action alphabet Σ by

- $\Sigma_C^{out} \triangleq \bigcup_i \Sigma_i^{out}$,
- $\Sigma_C^{in} \triangleq \bigcup_i \Sigma_i^{in} \setminus \bigcup_i \Sigma_i^{out}$,
- $\Sigma_C^{int} \triangleq \bigcup_i \Sigma_i^{int}$.

That is, the inputs of the composed system are all those inputs which are not at the same time output by another component, and each output of an individual I/O-module is an output of the composed system. Signals internal to any component are also internal in the composition.

The traces σ of a composed system M_C are all those traces such that the projection of σ onto each individual alphabet Σ_i is a trace of the system $\mathcal{M}_i$. In the context of real time, this definition will be modified slightly.

Chapter 2

Logical Specification Languages

Whereas formal modelling techniques can be used to specify the operational behaviour of a system, they are not well-suited to describe properties. Often, properties are documented as natural language constraints on the system under consideration. However, natural language is not fit for automated reasoning. One of the major concerns of philosophical logic is to find an appropriate language for the formalization of natural language constraints.

The first and probably most successful of these languages is first order logic. Almost all mathematical statements and proofs can be formulated in this language. However, certain concepts important for computer science like well-foundedness and transitive closure require more expressive languages.

Temporal logic was invented to formalize natural language sentences about events in time, which use temporal adverbs like "eventually" and "constantly". Temporal logics have proved to be useful for specifying concurrent systems, because they can describe the ordering of events without introducing time explicitly. There have been many variants of temporal logic proposed in the literature. Temporal logics can be classified as

- point- or interval-based, depending on whether the formulated properties focus on states or transitions,
- linear, branching or partially ordered, depending on the modelling of the flow of time,
- propositional, first- or higher order, depending on the cardinality of the temporal and nontemporal domains.

In this book, we concentrate on propositional modal logic, linear temporal logic, computation tree logic, and relational μ-calculus. Restrictions and extensions of these logics are introduced whenever appropriate.

2.1 Propositional and First Order Logic

We assume a set $\mathcal{P} = \{\mathtt{p}, \mathtt{q}, \mathtt{p}_1, ...\}$ of (atomic) propositionswhich can be either true or false.

For example, the proposition `stack_is_empty` denotes the fact that "the stack is empty". The *propositional logic* **PL** is built from $\mathcal{P}$ with the following syntax:

$$\mathbf{PL} \quad ::= \quad \mathcal{P} \quad | \quad \bot \quad | \quad (\mathbf{PL} \to \mathbf{PL})$$

That is,

- Every $\mathtt{p} \in \mathcal{P}$ is a well-formed formula of propositional logic,
- $\bot$ is a well-formed formula ("the falsum"),
- if φ and ψ are well-formed formulae, then so is $(\varphi \to \psi)$, and
- nothing else is a formula.

$\mathcal{P}$ is a parameter of the logic; the special case $\mathcal{P} = \emptyset$ is allowed. Other connectives can be defined as usual: $\neg\varphi \triangleq (\varphi \to \bot)$, $\top \triangleq \neg\bot$, $(\varphi \vee \psi) \triangleq (\neg\varphi \to \psi)$, $(\varphi \wedge \psi) \triangleq \neg(\neg\varphi \vee \neg\psi)$, and $(\varphi \leftrightarrow \psi) \triangleq ((\varphi \to \psi) \wedge (\psi \to \varphi))$. The precedence of these operators is fixed by $(\neg, \wedge, \vee, \to, \leftrightarrow)$, and parentheses are omitted in formulas whenever appropriate. Atomic propositions and negated propositions are called *literals*.

Assume an *interpretation* $\mathcal{I}$ for the propositions is given, which assigns a truth value from $\{\mathbf{true}, \mathbf{false}\}$ to every proposition. (For example, the proposition `stack_is_empty` is interpreted differently on a farm, in a library, or in front of a computer terminal.)

$\mathcal{I}$ can be extended to the set of all propositional formulas as follows:

- $\mathcal{I}(\bot) = \mathbf{false}$, and
- $\mathcal{I}((\varphi \to \psi)) = \mathbf{true}$ iff $\mathcal{I}(\varphi) = \mathbf{true}$ implies $\mathcal{I}(\psi) = \mathbf{true}$.

We write $\mathcal{I} \models \varphi$ iff $\mathcal{I}(\varphi) = \mathbf{true}$. Thus the above is equivalent to:

- $\mathcal{I} \models \mathtt{p}$ iff $\mathcal{I}(\mathtt{p}) = \mathbf{true}$,
- $\mathcal{I} \not\models \bot$, and

- $\mathcal{I} \models (\varphi \to \psi)$ iff $\mathcal{I} \models \varphi$ implies $\mathcal{I} \models \psi$.

Propositional logic is not well-suited to formalise statements about events in time. Even though the interpretation of a statement can be fixed, its truth value may vary in time.

To express such temporal dependencies, first order logic can be used. The set $\mathcal{P}$ is redefined to be a set of *monadic predicates*. That is, each $\mathtt{p} \in \mathcal{P}$ is augmented with an additional parameter denoting time, for example, $\mathtt{stack\_is\_empty}(t)$.

For sake of simplicity, we do not include function symbols (or constants) in the first-order language. Assume in addition to the set $\mathcal{P}$ of unary predicates a fixed set $\mathcal{R} \triangleq \{R, a, b, ...\}$ of *accessibility relations*, and let $\mathcal{R}^+ \triangleq \mathcal{R} \cup \{\prec, <, =\}$. Furthermore, let $\mathcal{T}$ be a set of *first-order variables* $\mathcal{T} \triangleq \{t, t_0, ...\}$ for points in time (which is assumed to be infinite unless stated otherwise).

$$\textbf{FOL} \quad ::= \quad \mathcal{P}(\mathcal{T}) \;\mid\; \bot \;\mid\; (\textbf{FOL} \to \textbf{FOL}) \;\mid\; \mathcal{R}^+(\mathcal{T}, \mathcal{T}) \;\mid\; \exists \mathcal{T}\ \textbf{FOL}$$

When writing formulas, we often use infix notation for relational terms: $t_1 R t_2 \triangleq R(t_1, t_2)$. The notation $\forall t\ \varphi$ is an abbreviation for $\neg\exists t\ \neg\varphi$, the string $x > y$ stands for $y < x$, and $x \leq y$ for $(x < y \vee x = y)$, etc.

To assign a truth value to a formula containing (free) variables, we assume that we are given a nonempty universe U of points in time, and that the *interpretation* $\mathcal{I}$ assigns to every proposition $\mathtt{p} \in \mathcal{P}$ a subset of points $\mathcal{I}(\mathtt{p}) \subseteq U$, and to every relation symbol $R \in \mathcal{R}$ a binary relation $\mathcal{I}(R) \subseteq U \times U$. For the special relation signs $=$, $\prec$, and $<$ we require that $\mathcal{I}(=) \triangleq \{(w, w) \mid w \in U\}$ is the *equality relation*, $\mathcal{I}(\prec) \triangleq \bigcup\{\mathcal{I}(R) \mid R \in \mathcal{R}\}$ is the *transition relation*, and $\mathcal{I}(<)$ is the transitive closure of $\mathcal{I}(\prec)$, the *reachability relation*. A *variable valuation* $\mathbf{v}$ assigns to any variable $t \in \mathcal{T}$ a point $w \in U$. A first-order model $\mathcal{M} \triangleq (U, \mathcal{I}, \mathbf{v})$ consists of a universe U, an interpretation $\mathcal{I}$, and a variable valuation $\mathbf{v}$. As in the propositional case, we define when a formula holds in a model:

- $\mathcal{M} \models \mathtt{p}(t)$ iff $\mathbf{v}(t) \in \mathcal{I}(\mathtt{p})$;
- $\mathcal{M} \not\models \bot$, and
- $\mathcal{M} \models (\varphi \to \psi)$ iff $\mathcal{M} \models \varphi$ implies $\mathcal{M} \models \psi$;
- $\mathcal{M} \models R(t_0, t_1)$ iff $(\mathbf{v}(t_0), \mathbf{v}(t_1)) \in \mathcal{I}(R)$;
- $\mathcal{M} \models \exists t\ \varphi$ iff $(U, \mathcal{I}, \mathbf{v}') \models \varphi$ for some $\mathbf{v}'$ which differs from $\mathbf{v}$ at most in t.

This language is rather expressive: Consider the following example formulas.

(1) $(\texttt{stack\_is\_empty}(t_0) \rightarrow \exists t_1(put(t_0, t_1) \wedge \neg\texttt{stack\_is\_empty}(t_1)))$
If `stack_is_empty`, then it is possible to perform a *put* such that not `stack_is_empty` holds.

(2) $\forall t_1((t_0 \leq t_1 \wedge \texttt{req}(t_1)) \rightarrow \exists t_2(t_1 < t_2 \wedge \texttt{ack}(t_2)))$
Every request is eventually acknowledged.

(3) $\forall t_1((t_0 \leq t_1 \wedge \texttt{req}(t_1)) \rightarrow \exists t_2((t_1 < t_2 \wedge \texttt{ack}(t_2)) \wedge \forall t_3((t_1 < t_3 \wedge t_3 < t_2) \rightarrow \texttt{req}(t_3))))$
No request is withdrawn before it is acknowledged.

2.2 Multimodal and Temporal Logic

First order logic has been criticised for not being intuitive. Except from text in mathematical books, one can hardly find English sentences which explicitly use variables to refer to objects. Natural language statements use modal adverbs like "possibly" and "necessarily" to refer to an alternative state of affairs. Temporal phrases in natural language use the adverbs "eventually" and "constantly" (or "sometime" and "always") to refer to future points in time. Modal logic was invented to formalise these modal and temporal adverbs [Lew12, Pri57]. The idea is to suppress first-order variables $t \in \mathcal{T}$; propositions $\texttt{p} \in \mathcal{P}$ are nullary again. The meaning of a proposition like `stack_is_empty` is "the stack is empty *now*". Thus, in a temporal interpretation, every formula describes a certain state of affairs *at a given point*.

To be able to describe properties depending on the relations between points, in multimodal logic for every $R \in \mathcal{R}$ a new operator $\langle R\rangle\ \varphi$ is introduced. The meaning of $\langle R\rangle\ \varphi$ is "possibly φ", i.e., "there exists some t accessible via R such that φ holds at t". Dually, $[R]\ \varphi \triangleq \neg\ \langle R\rangle\ \neg\varphi$ means "necessarily φ"; "for all t accessible via R, it is the case that φ holds at t".

$$\mathbf{ML} \quad ::= \quad \mathcal{P} \ \mid\ \bot \ \mid\ (\mathbf{ML} \rightarrow \mathbf{ML}) \ \mid\ \langle\mathcal{R}\rangle\,\mathbf{ML}.$$

Intuitively, the above example (1) could be written

$$(\texttt{stack\_is\_empty} \rightarrow\langle put\rangle\ \neg\texttt{stack\_is\_empty}).$$

A (Kripke-) *model* (introduced in [Kri63, Kri75]) $\mathcal{M} \triangleq (U, \mathcal{I}, w_0)$ for multimodal logic consists of a universe U of points, an interpretation $\mathcal{I}$ assigning to every $\texttt{p} \in \mathcal{P}$ and $R \in \mathcal{R}$ a subset $\mathcal{I}(\texttt{p}) \subseteq U$ and a relation $\mathcal{I}(R) \subseteq U \times U$,

respectively. Instead of a valuation for free variables, a model designates a *current* $w_0 \in U$.

- $\mathcal{M} \models \mathtt{p}$ iff $w_0 \in \mathcal{I}(\mathtt{p})$;
- $\mathcal{M} \not\models \bot$, and
- $\mathcal{M} \models (\varphi \to \psi)$ iff $\mathcal{M} \models \varphi$ implies $\mathcal{M} \models \psi$.
- $\mathcal{M} \models \langle R \rangle\ \varphi$ iff there exists $w_1 \in U$ with $(w_0, w_1) \in \mathcal{I}(R)$ and $(U, \mathcal{I}, w_1) \models \varphi$.

We write $w \models \varphi$ instead of $(U, \mathcal{I}, w) \models \varphi$ whenever the context U and $\mathcal{I}$ are given. A formula φ is *universally* valid in $(U, \mathcal{I})$, if for all $w \in U$ it holds that $w \models \varphi$.

As defined above, $\prec$ is interpreted as the *transition relation*, i.e., the union of all accessibility relations, $<$ is interpreted as the transitive closure of $\prec$, and $\leq$ as the reflexive transitive closure (the *reachability relation*). We introduce the special operators $\mathbf{X}$, $\mathbf{F}^+$ and $\mathbf{F}^*$:

- $w_0 \models \mathbf{X}\ \varphi$ iff there exists $w_1 \in U$ such that $(w_0, w_1) \in \mathcal{I}(\prec)$ and $w_1 \models \varphi$,
- $w_0 \models \mathbf{F}^+\ \varphi$ iff there exists $w_1 \in U$ such that $(w_0, w_1) \in \mathcal{I}(<)$ and $w_1 \models \varphi$, and
- $w_0 \models \mathbf{F}^*\ \varphi$ iff there exists $w_1 \in U$ such that $(w_0, w_1) \in \mathcal{I}(\leq)$ and $w_1 \models \varphi$.

For the dual operators, we use the symbols $\not\mathbf{X}\ \varphi \triangleq \neg\ \mathbf{X}\ \neg\varphi$, and $\mathbf{G}^+\ \varphi \triangleq \neg\ \mathbf{F}^+\ \neg\varphi$, and $\mathbf{G}^*\ \varphi \triangleq \neg\ \mathbf{F}^*\ \neg\varphi$. Traditionally, $\mathbf{X}$, $\mathbf{F}$, and $\mathbf{G}$ have been used to indicate ne$\mathbf{X}$t time, $\mathbf{F}$uture and $\mathbf{G}$lobal operators[1]. Alternatively, $\mathbf{F}^+$ and $\mathbf{G}^+$ are called *sometime*- and *always*-operators. $\not\mathbf{X}$ is referred to as *weak next*- operator. [Bur74] suggested the use of a modal logic built upon $\mathbf{F}^+$ and $\mathbf{G}^+$ to describe program properties. [Krö78] was the first to use $\mathbf{X}$ and $\mathbf{F}^+$ in program verification. [Pnu77] extended this framework for parallel programs. The combination of $\langle R \rangle$- and $\mathbf{F}^+$-operators originates from *dynamic logic* (for an overview on dynamic logics, see [Har84, KT90]).

Intuitively, $\mathbf{X}\ \varphi$ indicates that φ holds at some point accessible via a single transition, $\mathbf{F}^+\ \varphi$ specifies that φ must hold in some point which can

[1] A note on notation: With the above convention, the $\mathbf{X}$, $\not\mathbf{X}$, $\mathbf{F}^+$, $\mathbf{F}^*$, $\mathbf{G}^+$ and $\mathbf{G}^*$ operators could be written as $\langle \prec \rangle$, $[\prec]$, $\langle < \rangle$, $\langle \leq \rangle$, $[<]$ and $[\leq]$, respectively. In the literature, some authors use the symbols $\odot$, $\circ$, $\Diamond$, and $\Box$.

be reached by a nonempty sequence of transitions, and $\mathbf{F}^* \varphi$ means that φ holds at some reachable point (possibly now). Dually, $\not\mathbf{X} \varphi$ holds if all successors satisfy φ, and $\mathbf{G}^* \varphi$ and $\mathbf{G}^+ \varphi$ determine that all reachable points (except maybe the current point) must validate φ. With these operators, example (2) could be written

$$\mathbf{G}^* (\mathtt{req} \rightarrow \mathbf{F}^+ \mathtt{ack}).$$

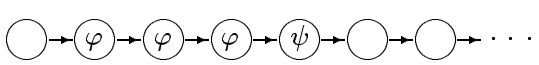
From the definition, $w_0 \models \not\mathbf{X} \varphi$ iff $w_1 \models \varphi$ for all $w_1 \in U$ such that $(w_0, w_1) \in \mathcal{I}(\prec)$. Similarly, $w_0 \models \mathbf{G}^+ \varphi$ iff $w_1 \models \varphi$ for all $w_1 \in U$ such that $(w_0, w_1) \in \mathcal{I}(<)$. Recall that a point $w \in U$ is *terminal*, if $\{w' \mid (w, w') \in \mathcal{I}(\prec)\} = \{\}$. A terminal point represents a final state of a terminating computation. Terminal points satisfy all $\not\mathbf{X}$- and $\mathbf{G}^+$-formulas vacuously: If w_0 has no accessible successors, then $w_0 \models \not\mathbf{X} \varphi$ and $w_0 \models \mathbf{G}^+ \varphi$ for any formula φ.

The difference between $\mathbf{F}^+$ and $\mathbf{F}^*$ is that in the latter "the future includes the present". Clearly, $(\mathbf{F}^* \varphi \leftrightarrow \varphi \vee \mathbf{F}^+ \varphi)$. Therefore, the $\mathbf{F}^*$-operator can be defined by $\mathbf{F}^+$. Using the identity $(\mathbf{F}^+ \varphi \leftrightarrow \mathbf{X}\mathbf{F}^* \varphi)$, $\mathbf{F}^+$ can be defined by $\mathbf{X}$ and $\mathbf{F}^*$ with linear increase in formula length. It is not possible to define the $\mathbf{F}^+$-operator only by $\mathbf{F}^*$: Consider two models $\mathcal{M}_1$ and $\mathcal{M}_2$, where $U_1 \triangleq U_2 \triangleq \{w\}$, $\mathcal{I}_1(\prec) \triangleq \{\}$, $\mathcal{I}_2(\prec) \triangleq \{(w, w)\}$ and $\mathcal{I}_1(\mathtt{p}) = \mathcal{I}_2(\mathtt{p})$ for all $\mathtt{p} \in \mathcal{P}$. Then $\mathcal{M}_1 \not\models \mathbf{F}^+ \top$ and $\mathcal{M}_2 \models \mathbf{F}^+ \top$. However, $w \models \mathbf{F}^* \varphi$ iff $w \models \varphi$ in both $\mathcal{M}_1$ and $\mathcal{M}_2$. Therefore, for all formulas φ which involve only propositions, boolean operators and $\mathbf{F}^*$ it holds that $\mathcal{M}_1 \models \varphi$ iff $\mathcal{M}_2 \models \varphi$. (Formally, the proof of this statement is by induction on the construction of such formulas). Thus $\mathbf{F}^+ \top$ is not expressible in this language. □

A similar proof shows that modal operators cannot express statements about intervals. For example, there is no formula equivalent to example (3) of the above. *Temporal logic* is based on a binary operator $(\varphi \ \mathbf{U}^+ \ \psi)$ meaning "φ holds until ψ holds". This operator was introduced in [Kam68] and used to describe properties of concurrent programs in [GPSS80]. The semantics of $\mathbf{U}^+$ is defined as follows[2] :

- $\mathcal{M} \models (\varphi \ \mathbf{U}^+ \ \psi)$ iff there exists $w_1 \in U$ with $w_0 < w_1$ and $w_1 \models \psi$, and for all $w_2 \in U$ with $w_0 < w_2$ and $w_2 < w_1$, we have $w_2 \models \varphi$.

This situation can be illustrated with a picture.



[2] For the relations $\sim \in \{\prec, <, =, \leq\}$, which have a fixed interpretation, we henceforth write $v \sim w$ for $(v, w) \in \mathcal{I}(\sim)$, whenever no confusion can arise.

As an example, the above formula (3) can be expressed with an until-operator as

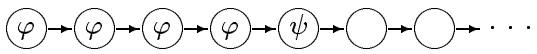
$$\mathbf{G}^* (\mathtt{req} \rightarrow (\mathtt{req}\, \mathbf{U}^+ \,\mathtt{ack})).$$

Various other operators can be defined via $\mathbf{U}^+$. Sometime-operator and nexttime operators are obtained as follows:

- $\mathbf{X}\,\varphi \leftrightarrow (\bot\ \mathbf{U}^+\ \varphi)$
- $\mathbf{F}^+\,\varphi \leftrightarrow (\top\ \mathbf{U}^+\ \varphi)$

The proof of these equivalences is immediate from the definition: $w_0 \models (\bot\ \mathbf{U}^+\ \psi)$ iff there exists $w_1 \in U$ with $w_0 < w_1$ and $w_1 \models \psi$, and for all $w_2 \in U$ with $w_0 < w_2 < w_1$ it holds that $w_2 \models \bot$, which is impossible. In other words, $w_0 < w_1$, but there is no w_2 that satisfies $w_0 < w_2$ and $w_2 < w_1$. Therefore w_1 must be an immediate successor of w_0, i.e., $w_0 \prec w_1$. Consequently, $w_0 \models \mathbf{X}\,\varphi$. The second equivalence is obtained in a similar way.

The *reflexive until*-operator is defined as $(\varphi\ \mathbf{U}^*\ \psi) \triangleq (\psi \vee \varphi \wedge (\varphi\ \mathbf{U}^+\ \psi))$.



As above, $\mathbf{F}^*\,\varphi \leftrightarrow (\top\ \mathbf{U}^*\ \varphi)$ and $(\varphi\ \mathbf{U}^+\ \psi) \leftrightarrow \mathbf{X}\ (\varphi\ \mathbf{U}^*\ \psi)$. Without $\mathbf{X}$ it is not possible to define $\mathbf{U}^+$ or $\mathbf{F}^+$ from $\mathbf{U}^*$. Hence, $\mathbf{X}$ cannot be defined by $\mathbf{U}^*$.

The *unless* or *weak until*-operator is defined as

$$(\varphi\ \mathbf{W}^+\ \psi) \triangleq \neg(\neg\psi\ \mathbf{U}^+\ \neg(\varphi \vee \psi)).$$

Intuitively, it says that φ holds at least up to the next point where ψ holds. This can be seen as follows: Assume that $w_0 \models \neg(\neg\psi\ \mathbf{U}^+\ \neg(\varphi \vee \psi))$. By definition, it is not the case that for some $w_1 > w_0$ both $w_1 \models \neg(\varphi \vee \psi)$ and $w_2 \models \neg\psi$ for all $w_0 < w_2 < w_1$. Thus, for all $w_1 > w_0$ it holds that $w_1 \models (\varphi \vee \psi)$, or $w_2 \models \psi$ for some $w_0 < w_2 < w_1$. In other words, if $w_1 > w_0$ then either $w_1 \models \varphi$ or there is some $w_0 < w_2 \leq w_1$ such that $w_2 \models \psi$. Therefore, if $w_2 \not\models \psi$ for all $w_0 < w_2 \leq w_1$, i.e. if w_1 is before the next point where ψ holds, then $w_1 \models \varphi$. □

Note that by definition $(\varphi\ \mathbf{W}^+\ \bot) = \neg(\top\ \mathbf{U}^+\ \neg\varphi) = \mathbf{G}^+\,\varphi$. Some texts define the unless operator by

$$(\varphi\ \mathbf{W}^+\ \psi) \triangleq ((\varphi\ \mathbf{U}^+\ \psi) \vee\ \mathbf{G}^+\,\varphi).$$

In models which consist of a sequence of points, these two definitions are equivalent. To prove this, we must show (i) $\mathcal{M} \models ((\varphi\ \mathbf{W}^+\ \psi) \rightarrow ((\varphi\ \mathbf{U}^+\ \psi) \vee\ \mathbf{G}^+\ \varphi))$, (ii) $\mathcal{M} \models (\mathbf{G}^+\ \varphi \rightarrow (\varphi\ \mathbf{W}^+\ \psi))$ and (iii) $\mathcal{M} \models ((\varphi\ \mathbf{U}^+\ \psi) \rightarrow (\varphi\ \mathbf{W}^+\ \psi))$ for all such models $\mathcal{M}$. For (i), assume that $w_0 \models (\varphi\ \mathbf{W}^+\ \psi)$ and $w_0 \not\models \mathbf{G}^+\ \varphi$. Then $w_1 \not\models \varphi$ for some $w_1 > w_0$. According to above, there is some $w_0 < w_2 \leq w_1$ such that $w_2 \models \psi$. Since the model is well-founded, this means that there is a smallest w_2 with this property; i.e. $w_0 < w_2 \leq w_1$, $w_2 \models \psi$, and $w_3 \not\models \psi$ for all $w_0 < w_3 < w_2$. Again, according to the above, if $w_0 < w_3 < w_2$ then $w_3 \models \varphi$. Therefore $w_0 \models (\varphi\ \mathbf{U}^+\ \psi)$. Formula (ii) follows immediately from the definition: if $w_0 \models \mathbf{G}^+\ \varphi$, then $w_1 \models \varphi$ for all $w_1 > w_0$. Therefore, it is not the case that some $w_1 > w_0$ exists which satisfies $w_1 \models \neg(\varphi \vee \psi)$. This implies $w_0 \not\models (\neg\psi\ \mathbf{U}^+\ \neg(\varphi \vee \psi))$, i.e., $w_0 \models (\varphi\ \mathbf{W}^+\ \psi)$. For implication (iii), we need the property that the model is linear: If $w_0 \models (\varphi\ \mathbf{U}^+\ \psi)$, then there exists $w_1 > w_0$ such that $w_1 \models \psi$ and $w_2 \models \varphi$ for all $w_0 < w_2 < w_1$. Assume any point $w > w_0$. Then $w < w_1$ or $w \geq w_1$. In the first case, $w \models \varphi$. In the second case, there exists $w' = w_1$ such that $w' \models \psi$. Thus, for all $w > w_0$ it holds that $w \models \varphi$, or there exists $w_0 < w' \leq w$ such that $w' \models \psi$. This shows that $w_0 \models (\varphi\ \mathbf{W}^+\ \psi)$. □

An immediate consequence of the above equivalence is that in models consisting of sequences $\mathbf{U}^+$ is definable by $\mathbf{W}^+$ and $\mathbf{F}^+$:

$$(\varphi\ \mathbf{U}^+\ \psi) \leftrightarrow ((\varphi\ \mathbf{W}^+\ \psi) \wedge\ \mathbf{F}^+\ \psi).$$

First order logic is more expressive than temporal logic since it can use *reverse* relations: $x > y$ iff $y < x$. Therefore, we introduce the temporal *past* operator $\mathbf{U}^-$, with the following semantics:

- $\mathcal{M} \models (\varphi\ \mathbf{U}^-\ \psi)$ iff $\exists w_1 (w_1 < w_0 \wedge \psi(w_1) \wedge \forall w_2 (w_1 < w_2 < w_0 \rightarrow \varphi(w_2)))$.

The syntax of *linear temporal logic* (**LTL**) is defined as follows:

$$\mathbf{LTL} \quad ::= \quad \mathcal{P} \mid \bot \mid (\mathbf{LTL} \rightarrow \mathbf{LTL}) \mid (\mathbf{LTL}\ \mathbf{U}^+\ \mathbf{LTL}) \mid (\mathbf{LTL}\ \mathbf{U}^-\ \mathbf{LTL}).$$

We write $\mathbf{F}^-\ \varphi$ and $\mathbf{G}^-\ \varphi$ for $(\top\ \mathbf{U}^-\ \varphi)$ and $\neg\ \mathbf{F}^-\ \neg\varphi$, respectively. Intuitively, these operators refer to "sometime in the past" and "always in the past". Moreover, $\mathbf{F}^\pm\ \varphi$ and $\mathbf{G}^\pm\ \varphi$ are abbreviations for $(\mathbf{F}^-\ \varphi \vee \varphi \vee\ \mathbf{F}^+\ \varphi)$ and $\neg\ \mathbf{F}^\pm\ \neg\varphi$, respectively.

The semantic clauses induce a translation **FOL** from temporal to first order logic, where $\mathbf{FOL}(\varphi)$ has exactly one free variable t_0.

- $\mathbf{FOL}(\mathtt{p}) \triangleq \mathtt{p}(t_0)$
- $\mathbf{FOL}(\bot) \triangleq (t_0 \neq t_0)$
- $\mathbf{FOL}((\varphi \to \psi)) \triangleq (\mathbf{FOL}(\varphi) \to \mathbf{FOL}(\psi))$
- $\mathbf{FOL}((\varphi\ \mathbf{U}^+\ \psi)) \triangleq \exists t'(t_0 < t' \wedge \mathbf{FOL}(\psi)\{t_0 := t'\} \wedge \forall t''(t_0 < t'' < t' \to \mathbf{FOL}(\varphi)\{t_0 := t''\}))$.
- $\mathbf{FOL}((\varphi\ \mathbf{U}^-\ \psi)) \triangleq \exists t'(t' < t_0 \wedge \mathbf{FOL}(\psi)\{t_0 := t'\} \wedge \forall t''(t' < t'' < t_0 \to \mathbf{FOL}(\varphi)\{t_0 := t''\}))$.

In the translation of $(\varphi\ \mathbf{U}^+\ \psi)$ and $(\varphi\ \mathbf{U}^-\ \psi)$, the symbols t' and t'' denote arbitrary variables which do not occur in $\mathbf{FOL}(\varphi)$ or $\mathbf{FOL}(\psi)$. The formula $\mathbf{FOL}(\psi)\{t_0 := t'\}$ denotes the formula $\mathbf{FOL}(\psi)$, where every (free) occurrence of the variable t_0 is replaced by the variable which is denoted by t'. The following example demonstrates this translation.

$\mathbf{FOL}(((\neg\mathtt{ack}\ \mathbf{U}^-\ \mathtt{req})\ \mathbf{U}^+\ \mathtt{ack}))$

$= \exists t_1(t_0 < t_1 \wedge \mathtt{ack}(t_1) \wedge \forall t_2(t_0 < t_2 < t_1 \to \mathbf{FOL}((\neg\mathtt{ack}\ \mathbf{U}^-\ \mathtt{req}))\{t_0 := t_2\}))$

$= \exists t_1(t_0 < t_1 \wedge \mathtt{ack}(t_1) \wedge \forall t_2(t_0 < t_2 < t_1 \to \exists t_3(t_3 < t_2 \wedge \mathtt{req}(t_3) \wedge \forall t_4(t_3 < t_4 < t_2 \to \neg\mathtt{ack}(t_4)))))$.

The translation of a temporal formula is a first-order formula with exactly one free variable t_0. Correctness of this translation is stated formally as follows:

Lemma 2.1 *For every $\varphi \in \mathbf{LTL}$ there exists a first order formula $\mathbf{FOL}(\varphi)$ such that for every model $\mathcal{M} \triangleq (U, \mathcal{I}, w_0)$ and valuation $\mathbf{v}$ for which $\mathbf{v}(t_0) = w_0$ it holds that $(U, \mathcal{I}, w_0) \models \varphi$ iff $(U, \mathcal{I}, \mathbf{v}) \models \mathbf{FOL}(\varphi)$.*

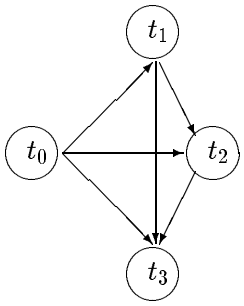
In other words, **FOL** is at least as expressive as **LTL**. For the translation of any given temporal formula into first order logic only three variables are really needed. Other variables can be reused; for example, $\mathbf{FOL}(((\neg\mathtt{ack}\ \mathbf{U}^-\ \mathtt{req})\ \mathbf{U}^+\ \mathtt{ack}))$ is equivalent to

$\exists t_1(t_0 < t_1 \wedge \mathtt{ack}(t_1) \wedge \forall t_2(t_0 < t_2 < t_1 \to \exists t_0(t_0 < t_2 \wedge \mathtt{req}(t_0) \wedge \forall t_1(t_0 < t_1 < t_2 \to \neg\mathtt{ack}(t_1)))))$.

As an immediate consequence, **LTL** cannot express properties which "inherently" use four variables. For example, the statement "there are three different connected points reachable from the current point" is not expressible in temporal logic.

$$\exists t_1, t_2, t_3(t_0 < t_1 \wedge t_0 < t_2 \wedge t_0 < t_3 \wedge t_1 < t_2 \wedge t_1 < t_3 \wedge t_2 < t_3)$$

A minimal model satisfying this formula is e.g. the following:



In case that $<$ is a linear order (antisymmetric and total) this is equivalent to

$$\exists t_1(t_0 < t_1 \wedge \exists t_2(t_1 < t_2 \wedge \exists t_3(t_2 < t_3)))$$

in which we can rename t_3 by t_0 to get the equivalent

$$\exists t_1(t_0 < t_1 \wedge \exists t_2(t_1 < t_2 \wedge \exists t_0(t_2 < t_0)))$$

which in turn can be expressed temporally as $\mathbf{F}^+\mathbf{F}^+\mathbf{F}^+ \top$.

Therefore, attention is restricted to certain classes of structures, like complete linear orders, or finitely-branching trees, etc. A *natural* model $\mathcal{M} \triangleq (U, \mathcal{I}, w_0)$ is a Kripke-model with only one accessibility relation, such that U is isomorphic to the natural numbers or an initial segment of the natural numbers[3], with $\prec$ interpreted as the usual successor relation.

For natural models, a converse to the above lemma holds: Given any $\varphi \in \mathbf{FOL}$ with at most one free variable w_0, there a temporal formula $\mathbf{LTL}(\varphi)$ such that for every natural model $\mathcal{M} \triangleq (U, \mathcal{I}, w_0)$ and valuation $\mathbf{v}$ for which $\mathbf{v}(t_0) = w_0$ it holds that $(U, \mathcal{I}, \mathbf{v}) \models \varphi$ iff $(U, \mathcal{I}, w_0) \models \mathbf{LTL}(\varphi)$. In other words,

Theorem 2.2 (Kamp, Gabbay) *Temporal logic is expressivly complete for natural models.*

The proof can be found in [CS01].

2.3 Linear and Branching Time Logics

Linear temporal logic is expressively complete for natural models. The same (with minor modifications) can be proved for finitely branching

[3]Some textbooks restrict attention to infinite models. Terminating computations are then modelled with an idle loop. In this book, we use ω-words, that is, computation sequences which can be finite or infinite.

trees [Sch92a, Sch92b]. In computer science, the set of executions of a program can be modelled as a set of execution sequences or as an execution tree, where branches denote nondeterministic decisions.

Statements about correctness of program can involve assertions about *all maximal paths* in a tree. A *path* in a model is a (finite or infinite) nonempty sequence of points $\sigma = (w_0, w_1, ...)$, where for each i with $0 \leq i < |\sigma|$ there exists an $R_i \in \mathcal{R}$ such that $(w_i, w_{i+1}) \in \mathcal{I}(R_i)$. A path is *maximal*, if each of its points which has a successor in the model also has a successor in the path. In other words, a maximal path is either infinite, or its final point w_n is terminal (there is no w such that $w_n \prec w$). Computation tree logic (**CTL**) [CE81] has the following syntax:

$$\mathbf{CTL} \quad ::= \quad \mathcal{P} \mid \bot \mid (\mathbf{CTL} \rightarrow \mathbf{CTL}) \mid \mathbf{E}(\mathbf{CTL}\ \mathbf{U}^+\ \mathbf{CTL}) \mid \mathbf{A}(\mathbf{CTL}\ \mathbf{U}^+\ \mathbf{CTL}).$$

CTL is interpreted on *tree models*. A tree is defined as usual: It has a single root w_0, and every node w_n can be reached from w_0 by exactly one finite path. The transitive closure "$<$" of the successor relation "$\prec$" then denotes the usual tree-order: $(w_1, w_2) \in \mathcal{I}(<)$ iff w_1 is on the (unique) path from the root w_0 up to w_2.

- $w_0 \models \mathbf{E}(\varphi\ \mathbf{U}^+\ \psi)$ iff there exists $w_1 > w_0$ such that $w_1 \models \psi$, and for all $w_2 \in U$, if $w_0 < w_2 < w_1$ then $w_2 \models \varphi$.
- $w_0 \models \mathbf{A}(\varphi\ \mathbf{U}^+\ \psi)$ iff *for all maximal paths* p from w_0 there exists $w_1 > w_0$ *on path* p such that $w_1 \models \psi$, and for all $w_0 < w_2 < w_1$, $w_2 \models \varphi$.

Thus, the $\mathbf{E}\ \mathbf{U}^+$-operator is defined similar to the **LTL** Until-operator. Since the intended models for **CTL** are trees, whereas **LTL** usually is interpreted on natural models, it makes sense to use a different notation. In **CTL** weak and derived operators can also be defined as abbreviations. However, in branching time, there are two variants of each derived operator.

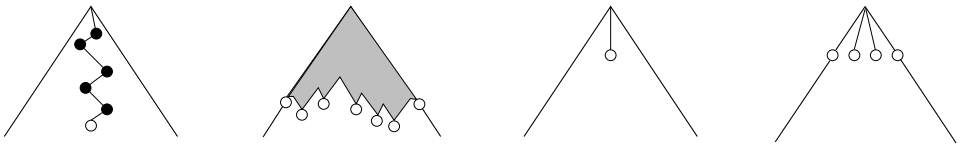
$$\mathbf{E}\ \mathbf{X}\ \psi \triangleq \mathbf{E}(\bot\ \mathbf{U}^+\ \psi), \qquad \mathbf{A}\ \mathbf{X}\ \psi \triangleq \mathbf{A}(\bot\ \mathbf{U}^+\ \psi),$$
$$\mathbf{E}\ \tilde{\mathbf{X}}\ \psi \triangleq \neg\mathbf{A}\ \mathbf{X}\ \neg\psi, \qquad \mathbf{A}\ \tilde{\mathbf{X}}\ \psi \triangleq \neg\mathbf{E}\ \mathbf{X}\ \neg\psi,$$
$$\mathbf{E}\ \mathbf{F}^+\ \psi \triangleq \mathbf{E}(\top\ \mathbf{U}^+\ \psi), \qquad \mathbf{A}\ \mathbf{F}^+\ \psi \triangleq \mathbf{A}(\top\ \mathbf{U}^+\ \psi),$$
$$\mathbf{E}\ \mathbf{G}^+\ \psi \triangleq \neg\mathbf{A}\ \mathbf{F}^+\ \neg\psi, \qquad \mathbf{A}\ \mathbf{G}^+\ \psi \triangleq \neg\mathbf{E}\ \mathbf{F}^+\ \neg\psi,$$
$$\mathbf{E}(\varphi\ \mathbf{U}^*\ \psi) \triangleq (\psi \vee \varphi \wedge \mathbf{E}(\varphi\ \mathbf{U}^+\ \psi)), \qquad \mathbf{A}(\varphi\ \mathbf{U}^*\ \psi) \triangleq (\psi \vee \varphi \wedge \mathbf{A}(\varphi\ \mathbf{U}^+\ \psi)),$$
$$\mathbf{E}\ \mathbf{F}^*\ \psi \triangleq (\psi \vee \mathbf{E}\ \mathbf{F}^+\ \psi), \qquad \mathbf{A}\ \mathbf{F}^*\ \psi \triangleq (\psi \vee \mathbf{A}\ \mathbf{F}^+\ \psi),$$

$$\mathbf{E}\ \mathbf{G}^*\ \psi \triangleq (\psi \wedge \mathbf{E}\ \mathbf{G}^+\ \psi), \qquad \mathbf{A}\ \mathbf{G}^*\ \psi \triangleq (\psi \wedge \mathbf{A}\ \mathbf{G}^+\ \psi)$$

$$\mathbf{E}(\varphi\ \mathbf{W}^+\ \psi) \triangleq \neg\mathbf{A}(\neg\psi\ \mathbf{U}^+\ \neg(\varphi \vee \psi)), \quad \mathbf{A}(\varphi\ \mathbf{W}^+\ \psi) \triangleq \neg\mathbf{E}(\neg\psi\ \mathbf{U}^+\ \neg(\varphi \vee \psi))$$

Informally, $\mathbf{E}\ \mathbf{X}\ \psi$ means that some successor node satisfies ψ, and $\mathbf{A}\ \not\mathbf{X}\ \psi$ holds if all successors are ψ. In a terminal point, $\mathbf{A}\ \not\mathbf{X}\ \bot$ is true, but $\mathbf{A}\ \mathbf{X}\ \bot$ not: If w_0 has no successors, then the only maximal path p from w_0 is the one-element sequence $\sigma = (w_0)$. On this unique path σ there is no $w_1 > w_0$, therefore each formula $\mathbf{A}(\varphi\ \mathbf{U}^+\ \psi)$ and $\mathbf{E}(\varphi\ \mathbf{U}^+\ \psi)$ must be false. As a special case, in such a point $\mathbf{E}\ \mathbf{X}\ \top$ is false, but $\mathbf{E}\ \not\mathbf{X}\ \top$ and $\mathbf{E}\ \not\mathbf{X}\ \bot$ are true. In a nonterminal point, $(\mathbf{E}\ \mathbf{X}\ \varphi \leftrightarrow \mathbf{E}\ \not\mathbf{X}\ \varphi)$ and $(\mathbf{A}\ \mathbf{X}\ \varphi \leftrightarrow \mathbf{A}\ \not\mathbf{X}\ \varphi)$. Thus, if we restrict attention to models without terminal points, these operators coincide. The operators $\mathbf{A}\ \mathbf{X}$ and $\mathbf{E}\ \not\mathbf{X}$ can be expressed by $\mathbf{E}\ \mathbf{X}$ and $\mathbf{A}\ \not\mathbf{X}$ (with linear increase of formula length) via $(\mathbf{A}\ \mathbf{X}\ \varphi \leftrightarrow \mathbf{A}\ \not\mathbf{X}\ \varphi \wedge \mathbf{E}\ \mathbf{X}\ \top)$ and $(\mathbf{E}\ \not\mathbf{X}\ \varphi \leftrightarrow \mathbf{E}\ \mathbf{X}\ \varphi \vee \mathbf{A}\ \not\mathbf{X}\ \bot) \leftrightarrow (\mathbf{E}\ \mathbf{X}\ \top \rightarrow \mathbf{E}\ \mathbf{X}\ \varphi))$. Thus, all **CTL** nexttime-operators can be expressed in terms of $\mathbf{E}\ \mathbf{X}$.

The formula $\mathbf{E}\ \mathbf{F}^*\ \psi$ means that some node in the computation tree satisfies ψ, and $\mathbf{A}\ \mathbf{F}^*\ \psi$ specifies that ψ must hold somewhere along every computation path. Dually, $\mathbf{A}\ \mathbf{G}^*\ \psi$ means that every node in the (sub-) tree satisfies ψ, whereas $\mathbf{E}\ \mathbf{G}^*\ \psi$ indicates that ψ is globally valid along some path.



$\mathbf{E}(\varphi\ \mathbf{U}^+\ \psi)$ $\mathbf{A}(\varphi\ \mathbf{U}^*\ \psi)$ $\mathbf{E}\ \mathbf{X}\ \psi$ $\mathbf{A}\ \mathbf{X}\ \psi$

In the above picture, nodes satisfying φ are shown solid (or as a shaded area), whereas ψ nodes are indicated by a circle.

The operator $\mathbf{A}\ \mathbf{U}^+$ can be expressed by $\mathbf{E}\ \mathbf{U}^+$ and $\mathbf{A}\ \mathbf{F}^+$. This characterization is similar to the definition of the unless-operator in linear temporal logic, cf. page 40:

$$\mathbf{A}(\varphi\ \mathbf{U}^+\ \psi) \leftrightarrow (\mathbf{A}(\varphi\ \mathbf{W}^+\ \psi) \wedge \mathbf{A}\ \mathbf{F}^+\ \psi) = (\neg\mathbf{E}(\neg\psi\ \mathbf{U}^+\ \neg(\varphi \vee \psi)) \wedge \mathbf{A}\ \mathbf{F}^+\ \psi).$$

Therefore, it is sufficient to consider only the two basic operators $\mathbf{E}\ \mathbf{U}^+$ and $\mathbf{A}\ \mathbf{F}^+$ in formal proofs and algorithms. Similarly, the formula $\mathbf{E}(\varphi\ \mathbf{W}^+\ \psi)$ can be replaced by $(\mathbf{E}(\varphi\ \mathbf{U}^+\ \psi) \vee \mathbf{E}\ \mathbf{G}^+\ \varphi)$. However, there is no negation-free "dual" characterization of $\mathbf{A}\ \mathbf{W}^+$ and $\mathbf{E}\ \mathbf{U}^+$.

Some typical formulas that might arise in verifying a finite state concurrent program are given below:

— $\mathbf{E}\,\mathbf{F}^+$ ($\mathtt{started} \wedge \neg\mathtt{ready}$): It is possible to get to a state where `started` holds but `ready` does not hold.

— $\mathbf{A}\,\mathbf{G}^*$ ($\mathtt{req} \rightarrow \mathbf{A}\,\mathbf{F}^+\,\mathtt{ack}$): If a request occurs, then it will be eventually acknowledged

— $\mathbf{A}\,\mathbf{G}^*\;\mathbf{A}\,\mathbf{F}^*\;\mathtt{stack\_is\_empty}$: The proposition `stack_is_empty` holds infinitely often on every computation path

— $\mathbf{A}\,\mathbf{G}^*\;\mathbf{E}\,\mathbf{F}^*\;\mathtt{restart}$: From any state it is possible to get to a `restart` state.

For many **CTL** formulas it is possible to formulate similar correctness properties in **LTL**. *Possibility properties* like the last one mentioned above can not be formulated in **LTL**. On the other hand, certain *fairness properties* cannot be formulated in **CTL**.

How can we compare the expressivity of **CTL** with (the future fragment of) **LTL**? Direct comparison is difficult, since models are different: On natural models, which are special tree models with branching degree one, $\mathbf{A}\,\mathbf{U}^+$ and $\mathbf{E}\,\mathbf{U}^+$-operators coincide. On tree models with higher branching degree, **LTL** obviously cannot express $\mathbf{A}(\varphi\;\mathbf{U}^+\;\psi)$.

Therefore, one considers **LTL** and **CTL** on (nonlinear, nontree) Kripke-models $(U, \mathcal{I}, w_0)$. In contrast to natural or tree models, Kripke-models can contain reflexive points, loops or even dense relations. We call an **LTL**-formula *sequence-valid* in a Kripke-model, if it is valid for all natural models *generated* from the model, that is, for all maximal paths in U starting from w_0. (A formal definition of this notion will be given in Chapter 3.) Similarly, a **CTL**-formula is called *tree-valid* in a Kripke-model, if it is valid for the unique maximal tree generated from it.

With this definition, the expressivity of **LTL** and **CTL** are incomparable. For example, the **LTL**-formula $\varphi \triangleq \mathbf{F}^+\mathbf{G}^+\,\mathtt{p}$ is not expressible in **CTL** (it is *not* the same property as $\mathbf{A}\,\mathbf{F}^+\,\mathbf{A}\,\mathbf{G}^+\,\mathtt{p}$). That is, there is no **CTL**-formula ψ such that ψ is tree-valid in exactly the same Kripke-models in which φ is sequence-valid. Similarly, $\mathbf{A}\,\mathbf{G}^+\,\mathbf{E}\,\mathbf{F}^+\,\mathtt{p}$ is not expressible in **LTL** (it is *not* the same as $\mathbf{G}^+\mathbf{F}^+\,\mathtt{p}$). For more information on the expressiveness of linear versus branching time see [EL85, EH86, CD88, Eme90].

On Kripke-models, the logic $\mathbf{CTL}^*$ (see [EL85, EH86]) subsumes **CTL** and **LTL** by separating path quantification (**E**) from temporal quantification ($\mathbf{U}^+$). Thus it is possible to write e.g. $\mathbf{E}\,\mathbf{G}^*\mathbf{F}^*\,\mathtt{p}$. The logic $\mathbf{CTL}^*$ is strictly

more expressive than both **CTL** and **LTL**. On binary trees, the expressiveness of $\mathbf{CTL}^*$ can be compared to first order logic with additional (second order) quantification on paths. For more information on the expressiveness and complexity of various sublogics of $\mathbf{CTL}^*$, see [Eme90].

2.4 Propositionally Quantified Logics

Quantification over paths is not a first-order notion. For linear time, this quantifier is not very useful. But why should second-order quantification be restricted to paths? Wolper remarked that "temporal logic can be more expressive"[Wol83]. In temporal or first-order logic, it is not possible to specify that a certain proposition $\mathtt{p}$ holds on every *second* point of an execution sequence, without constraining the values of $\mathtt{p}$ in intermediate points. Formally, for a natural model where $U = (w_0, w_1, ...)$, define the new operator $\mathbf{G}^{2n}$ by

$$w_i \models \mathbf{G}^{2n}\varphi \quad \text{iff} \quad w_{i+2n} \models \varphi \text{ for all } n \geq 0$$

We will show that this operator can not be expressed in **LTL** or **FOL**. First, note that the following operator defines a stronger property than required:

$$\mathbf{G}^{2n}_{\mathbf{LTL}}\varphi \triangleq \varphi \wedge \mathbf{G}^*(\varphi \rightarrow \mathbf{XX}\varphi)$$

$$(\mathbf{G}^{2n}_{\mathbf{FOL}}\varphi)(t_0) \triangleq \varphi(t_0) \wedge \forall t \geq t_0(\varphi(t) \rightarrow \forall t_1, t_2(tRt_1Rt_2 \rightarrow \varphi(t_2)))$$

These formulas are no adequate formulation of $\mathbf{G}^{2n}\varphi$. They imply that if φ holds twice in a row, it must hold always. Therefore, $\models (\mathbf{G}^{2n}_{\mathbf{LTL}}\varphi \rightarrow \mathbf{G}^{2n}\varphi)$. The reverse implication does not hold: there are models satisfying $\mathbf{G}^{2n}\varphi$ but not $\mathbf{G}^{2n}_{\mathbf{LTL}}\varphi$ or $\mathbf{G}^{2n}_{\mathbf{FOL}}\varphi(t_0)$, respectively.

Theorem 2.3 (Wolper) *Let* $\mathtt{p}$ *be any atomic proposition. There is no* **LTL***-formula* φ *such that* $\models \varphi \leftrightarrow \mathbf{G}^{2n}\mathtt{p}$.

Thus the $\mathbf{G}^{2n}$ operator cannot be defined in the basic temporal or first order language. However, it can be defined if additional propositions are allowed. To assert that $\mathbf{G}^{2n}\varphi$ holds, it suffices to provide a "new" proposition $\mathtt{q}$ (not occurring in φ) such that $\mathbf{G}^{2n}_{\mathbf{LTL}}\mathtt{q}$ holds, and that φ is true wherever $\mathtt{q}$ is true. This puts an additional constraint on the "auxiliary variable" $\mathtt{q}$, which can be considered as an "implementation detail" in the context of φ. If we neglect the value of $\mathtt{q}$, then the models satisfying $(\mathbf{G}^{2n}_{\mathbf{LTL}}\mathtt{q} \wedge \mathbf{G}^*(\mathtt{q} \rightarrow \varphi))$ are exactly those satisfying $\mathbf{G}^{2n}\varphi$. That is, for any model $\mathcal{M}$ such that $\mathcal{M} \models (\mathbf{G}^{2n}_{\mathbf{LTL}}\mathtt{q} \wedge \mathbf{G}^*(\mathtt{q} \rightarrow \varphi))$ it holds that $\mathcal{M} \models \mathbf{G}^{2n}\varphi$, and for every model

$\mathcal{M}$ such that $\mathcal{M} \models \mathbf{G}^{2n}\,\varphi$ it holds that $\mathcal{M}' \models (\mathbf{G}^{2n}_{\mathbf{LTL}}\,\mathtt{q} \wedge\, \mathbf{G}^{*}\,(\mathtt{q} \rightarrow \varphi))$, where $\mathcal{M}'$ differs from $\mathcal{M}$ only in the fact that $\mathcal{I}(\mathtt{q}) = \{w_0, w_2, w_4, ...\}$. Logically, this projection operation amounts to existential quantification on temporal propositions or sets of points:

$$\mathbf{G}^{2n}\,\varphi \leftrightarrow \exists q(\mathbf{G}^{2n}_{\mathbf{LTL}}\,q \wedge\, \mathbf{G}^{*}\,(q \rightarrow \varphi))$$

$$(\mathbf{G}^{2n}\,\varphi)(t_0) \leftrightarrow \exists q((\mathbf{G}^{2n}_{\mathbf{FOL}}\,q)(t_0) \wedge \forall t \geq t_0(q(t) \rightarrow \varphi(t))))$$

The language used in the first item is called quantified temporal logic **qTL** [Sis83], the language of the second item is *monadic second order logic* **MSOL**.

$$\begin{aligned}\mathbf{qTL} \;::=\; & \mathcal{P} \mid \mathcal{Q} \mid \bot \mid (\mathbf{qTL} \rightarrow \mathbf{qTL}) \mid \\ & (\mathbf{qTL}\ \mathbf{U}^{+}\ \mathbf{qTL}) \mid (\mathbf{qTL}\ \mathbf{U}^{-}\ \mathbf{qTL}) \mid \exists\mathcal{Q}\ \mathbf{qTL}.\end{aligned}$$

$$\begin{aligned}\mathbf{MSOL} \;::=\; & \mathcal{P}(\mathcal{T}) \mid \mathcal{Q}(\mathcal{T}) \mid \bot \mid (\mathbf{MSOL} \rightarrow \mathbf{MSOL}) \mid \\ & \mathcal{R}^{+}(\mathcal{T}, \mathcal{T}) \mid \exists\mathcal{T}\ \mathbf{MSOL} \mid \exists\mathcal{Q}\ \mathbf{MSOL}\end{aligned}$$

To define this syntax, we used another syntactic category $\mathcal{Q} = \{q, q_0, ...\}$ of *proposition variables*. Any valuation in a model assigns a subset of U to each of these (second order) variables. The formula $\exists q\ \varphi$ is true in a model $\mathcal{M}$ if it is true in some model which differs from $\mathcal{M}$ at most in the valuation of the proposition variable q.

It is easy to lift the expressive completeness theorem 2.2 to second order.

Lemma 2.4 *On natural models,* **qTL** *has the same expressiveness as* **MSOL**.

Lemma 2.5 *On natural models, the* $\mathbf{U}^{+}$*-operator in* **qTL** *is definable by* $\mathbf{G}^{\pm}$ *and* $\mathbf{X}$*:*

$$(\varphi\ \mathbf{U}^{+}\ \psi) \leftrightarrow \forall q(\mathbf{G}^{\pm}\,(\mathbf{X}\,(\psi \vee \varphi \wedge q) \rightarrow q) \rightarrow q).$$

Proof: Since this lemma is used several times in subsequent chapters, we give a detailed proof. For one direction, assume that $(\varphi\ \mathbf{U}^{+}\ \psi)$ is true in $\mathcal{M} \triangleq (U, \mathcal{I}, w_0)$. To prove that $\mathcal{M} \models \forall q(\mathbf{G}^{\pm}\,(\mathbf{X}\,(\psi \vee \varphi \wedge q) \rightarrow q) \rightarrow q)$, let $\mathcal{I}'(q)$ be an arbitrary set of points, and show that $(U, \mathcal{I}', w_0) \models (\mathbf{G}^{\pm}\,(\mathbf{X}\,(\psi \vee \varphi \wedge q) \rightarrow q) \rightarrow q)$. In other words, from the assumption $w_0 \models \mathbf{G}^{\pm}\,(\mathbf{X}\,(\psi \vee \varphi \wedge q) \rightarrow q)$ we have to show that $w_0 \models q$. In any natural model satisfying $w_0 \models (\varphi\ \mathbf{U}^{+}\ \psi)$, there are w_1, ..., $w_n \in U$ such that $w_i \prec w_{i+1}$ for all $0 \leq i < n$, and $\varphi(w_i)$ for all $0 < i < n$, and $w_n \models \psi$. If $w_0 \models \mathbf{G}^{\pm}\,(\mathbf{X}\,(\psi \vee \varphi \wedge q) \rightarrow q)$, then $w_i \models (\mathbf{X}\,(\psi \vee \varphi \wedge q) \rightarrow q)$ for all

$i \geq 0$. Hence, $w_i \models (\mathbf{X}\ \psi \rightarrow q)$ and $w_i \models (\mathbf{X}\ (\varphi \wedge q) \rightarrow q)$ for all $i \geq 0$. From $w_n \models \psi$ it follows that $w_{n-1} \models \mathbf{X}\ \psi$. Since $w_{n-1} \models (\mathbf{X}\ \psi \rightarrow q)$, we have $w_{n-1} \models q$. Therefore $w_{n-1} \models (\varphi \wedge q)$, and $w_{n-2} \models \mathbf{X}\ (\varphi \wedge q)$. Since $w_{n-2} \models (\mathbf{X}\ (\varphi \wedge q) \rightarrow q)$, it follows that $w_{n-2} \models q$. Continuing inductively, we find that $w_i \models q$ for all $0 \leq i < n$. Therefore, $w_0 \models q$.

For the other direction, assume that $w_0 \models \forall q(\mathbf{G}^{\pm}\ (\mathbf{X}\ (\psi \vee \varphi \wedge q) \rightarrow q) \rightarrow q)$ and show that $w_0 \models (\varphi\ \mathbf{U}^{+}\ \psi)$. First, we show that there must be some $w > w_0$ satisfying $w \models \psi$. Assume for contradiction that this is not the case. Choose $\mathcal{I}(q) \triangleq \{w \mid \text{not } w \geq w_0\}$. In natural models, this is the set $\{w \mid w < w_0\}$. It follows that (i) $w \models q$ for all w such that not $w \geq w_0$, (ii) $w_0 \not\models q$, and (iii) $w \not\models q$ for all $w > w_0$. We show that (*): $w \models (\mathbf{X}\ (\psi \vee \varphi \wedge q) \rightarrow q)$ for all $w \in U$. According to the contradiction assumption, $w \not\models \psi$ for all $w > w_0$. With (iii), it follows that $w \not\models (\psi \vee \varphi \wedge q)$ for all $w > w_0$. Hence, $w \not\models \mathbf{X}\ (\psi \vee \varphi \wedge q)$ for all $w \geq w_0$. As a consequence, (*) holds for all $w \geq w_0$. If not $w \geq w_0$, then (*) is an immediate consequence of (i). From (*), we infer that $w_0 \models \mathbf{G}^{\pm}\ (\mathbf{X}\ (\psi \vee \varphi \wedge q) \rightarrow q)$. Therefore, $w_0 \models q$, which is a contradiction to (ii).

Let $w_1, ..., w_n$ be a set of points such that $w_i \prec w_{i+1}$ for all $0 \leq i < n$, and w_n is the smallest point satisfying ψ (i.e., $w_n \models \psi$ and $w_i \models \neg\psi$ for all $w_0 < w_i < w_n$). If $n = 1$, we are done: In this case $w_0 \models \mathbf{X}\ \psi$, which implies that $w_0 \models (\varphi\ \mathbf{U}^{+}\ \psi)$. If $n > 1$, to prove $w_0 \models (\varphi\ \mathbf{U}^{+}\ \psi)$ we additionally have to show that $w_i \models \varphi$ for any $0 < i < n$. Substitution of q with $\neg q$ in the assumption yields the following equivalent version: $w_0 \models \forall q(q \rightarrow \mathbf{F}^{\pm}\ (q \wedge\ \mathbf{X}\ (\psi \vee \varphi \wedge \neg q)))$. Choose $\mathcal{I}(q) \triangleq \{w \mid w_0 \leq w < w_i\}$. It follows that $w_0 \models \mathbf{F}^{\pm}\ (q \wedge\ \mathbf{X}\ (\psi \vee \varphi \wedge \neg q))$. That is, there is some $w \in U$ such that $w \models (q \wedge\ \mathbf{X}\ (\psi \vee \varphi \wedge \neg q))$. Since n is minimal, there is no $w \in \mathcal{I}(q)$ which satisfies $w \models \mathbf{X}\ \psi$. Therefore, it follows that there is a $w \geq w_0$ such that $w \models (q \wedge\ \mathbf{X}\ (\varphi \wedge \neg q))$. Since w_{i-1} is the only point with $w_{i-1} \models (q \wedge\ \mathbf{X}\ \neg q)$ we can conclude that $w_{i-1} \models \mathbf{X}\ \varphi$, i.e., $w_i \models \varphi$. □

This characterization of the $\mathbf{U}^{+}$-operator with second order quantification is a special case of the general scheme $\forall q(\mathbf{G}^{\pm}\ (\xi \rightarrow q) \rightarrow q)$, where $\xi \triangleq \mathbf{X}\ (\psi \vee \varphi \wedge q)$. Dually, the operator $(\varphi\ \mathbf{W}^{+}\ \psi) \triangleq \neg(\neg\psi\ \mathbf{U}^{+}\ \neg(\varphi \vee \psi))$ is characterized by

$$
\begin{aligned}
(\varphi\ \mathbf{W}^{+}\ \psi) &\leftrightarrow \neg\forall q(\mathbf{G}^{\pm}\ (\mathbf{X}\ (\neg(\varphi \vee \psi) \vee (\neg\psi \wedge q)) \rightarrow q) \rightarrow q) \\
&\leftrightarrow \exists q(\neg q \wedge\ \mathbf{G}^{\pm}\ (\mathbf{X}\ ((\neg\psi \wedge \neg\varphi) \vee (\neg\psi \wedge q)) \rightarrow q)) \\
&\leftrightarrow \exists q(\neg q \wedge\ \mathbf{G}^{\pm}\ (\neg q \rightarrow \neg\ \mathbf{X}\ (\neg\psi \wedge (\neg\varphi \vee q)))) \\
&\leftrightarrow \exists q(\neg q \wedge\ \mathbf{G}^{\pm}\ (\neg q \rightarrow \widetilde{\mathbf{X}}\ (\psi \vee (\varphi \wedge \neg q)))) \\
&\leftrightarrow \exists q(q \wedge\ \mathbf{G}^{\pm}\ (q \rightarrow \widetilde{\mathbf{X}}\ (\psi \vee \varphi \wedge q)))
\end{aligned}
$$

This is an instance of the dual scheme $\exists q(q \wedge \mathbf{G}^{\pm}\ (q \rightarrow \xi))$ with $\xi \triangleq \mathbf{X}\ (\psi \vee \varphi \wedge q)$.

For complexity reasons, it is not always advisable to allow quantifiers on arbitrary subsets of the universe U. Therefore, we introduce *fixpoint quantification*: quantification on sets which follows these schemes. This results in the *propositional μ-calculus* $\mu\mathbf{TL}$ [EC80, Pra81, Koz83, KP83]:

$$\mu\mathbf{TL} \quad ::= \quad \mathcal{P} \ \mid\ \mathcal{Q} \ \mid\ \bot \ \mid\ (\mu\mathbf{TL} \rightarrow \mu\mathbf{TL}) \ \mid\ \langle\mathcal{R}\rangle\ \mu\mathbf{TL} \ \mid\ \nu\,\mathcal{Q}\ \mu\mathbf{TL}.$$

The semantics of $\mu\mathbf{TL}$ can be defined by a translation into **MSOL**.

- $\mathbf{MSOL}(\varphi) \triangleq \mathbf{FOL}(\varphi)$, if $\varphi \in \mathbf{ML}$
- $\mathbf{MSOL}(q) \triangleq q(t_0)$, if $q \in \mathcal{Q}$
- $\mathbf{MSOL}(\nu q\ \varphi) \triangleq \exists q(q(t_0) \wedge \forall t(q(t) \rightarrow \mathbf{MSOL}(\varphi)\{t_0 := t\}))$.

We write $\varphi\{q := \psi\}$ for the formula which results from φ by replacing every free occurrence of q with ψ. The formula $\mu q\ \varphi$ is short for $\neg\nu q\ \neg(\varphi\{q := \neg q\})$. Thus, the translation of $\mu q\ \varphi$ evaluates to

- $\begin{aligned}\mathbf{MSOL}(\mu q\ \varphi) &= \neg\exists q(\neg q(t_0) \wedge \forall t(\neg q(t) \rightarrow \neg\mathbf{MSOL}(\varphi)\{t_0 := t\})) \\ &= \forall q(q(t_0) \vee \neg\forall t(\neg q(t) \rightarrow \neg\mathbf{MSOL}(\varphi)\{t_0 := t\})) \\ &= \forall q(\forall t(\mathbf{MSOL}(\varphi)\{t_0 := t\} \rightarrow q(t)) \rightarrow q(t_0)).\end{aligned}$

When interpreting $\mu\mathbf{TL}$ on natural models, we use the operator $\mathbf{X}$ for the unique diamond operator $\langle R\rangle$. With this notation, Lemma 2.5 can be reformulated:

Corollary 2.6 *For any natural model $\mathcal{M}$,*

$$\mathcal{M} \models (\varphi\ \mathbf{U}^{+}\ \psi) \quad \textit{iff} \quad \mathcal{M} \models \mu q\ \ \mathbf{X}\ (\psi \vee \varphi \wedge q)$$

PROOF: The equivalence follows almost immediately from the definitions.

$$\begin{aligned}&\mathbf{MSOL}(\mu q\ \mathbf{X}\ (\psi \vee \varphi \wedge q)) \\ &\quad = \forall q(\forall t(\mathbf{MSOL}(\mathbf{X}\ (\psi \vee \varphi \wedge q))\{t_0 := t\} \rightarrow q(t)) \rightarrow q(t_0)) \\ &\quad = \mathbf{MSOL}(\forall q(\mathbf{G}^{\pm}\ (\mathbf{X}\ (\psi \vee \varphi \wedge q) \rightarrow q) \rightarrow q)) \\ &\quad \leftrightarrow \mathbf{FOL}((\varphi\ \mathbf{U}^{+}\ \psi)) \quad \text{(acc. to Lemma 2.5)} \qquad \Box\end{aligned}$$

Corollary 2.6 does not hold for more general Kripke models. In natural models, other operators are characterized by similar $\mu\mathbf{TL}$ formulas:

$$\begin{aligned}\mathcal{M} \models \mathbf{F}^{+}\ \psi \quad &\text{iff} \quad \mathcal{M} \models \mu q\ \ \mathbf{X}\ (\psi \vee q) \\ \mathcal{M} \models (\varphi\ \mathbf{W}^{+}\ \psi) \quad &\text{iff} \quad \mathcal{M} \models \nu q\ \ \mathbf{X}\ (\psi \vee \varphi \wedge q) \\ \mathcal{M} \models \mathbf{G}^{*}\ \psi \quad &\text{iff} \quad \mathcal{M} \models \nu q\ (\psi \wedge \mathbf{X}\ q) \\ \mathcal{M} \models (\varphi\ \mathbf{U}^{*}\ \psi) \quad &\text{iff} \quad \mathcal{M} \models \mu q\ (\psi \vee \varphi \wedge \mathbf{X}\ q)\end{aligned}$$

For certain formulas, an alternative semantical description of the ν and μ quantifiers in terms of greatest and least fixed points can be given. A function $f : 2^U \to 2^U$ is called *monotonic*, if $P \subseteq Q$ implies that $f(P) \subseteq f(Q)$. A set $Q \subseteq U$ is called a *fixed point* of f, if $Q = f(Q)$.

Let $gfp(f) = \bigcup\{Q \mid Q \subseteq f(Q)\}$ and $lfp(f) = \bigcap\{Q \mid f(Q) \subseteq Q\}$. The Knaster-Tarski fixpoint theorem states that if f is monotonic, then $gfp(f)$ and $lfp(f)$ are the *greatest* and *least fixed point* of f.

Theorem 2.7 (Knaster-Tarski) *Let $f : 2^U \to 2^U$ be monotonic. Then*

- $gfp(f) = f(gfp(f))$ *and* $lfp(f) = f(lfp(f))$, *and*
- *If $Q = f(Q)$, then $Q \subseteq gfp(f)$ and $lfp(f) \subseteq Q$.*

PROOF: Since *gfp* and *gfp* are dual, it suffices to prove the theorem for *gfp*.

If $Q = f(Q)$, then $Q \subseteq f(Q)$. Therefore $Q \in \{Q \mid Q \subseteq f(Q)\}$, that is, $Q \subseteq \bigcup\{Q \mid Q \subseteq f(Q)\}$. If $Q \subseteq f(Q)$, then $Q \subseteq \bigcup\{Q \mid Q \subseteq f(Q)\}$. Monotonicity of f implies that in this case $f(Q) \subseteq f(\bigcup\{Q \mid Q \subseteq f(Q)\})$. Hence for all Q, if $Q \subseteq f(Q)$ then $Q \subseteq f(\bigcup\{Q \mid Q \subseteq f(Q)\})$ by transitivity of set inclusion. This means that $\bigcup\{Q \mid Q \subseteq f(Q)\} \subseteq f(\bigcup\{Q \mid Q \subseteq f(Q)\})$. For the other direction, observe that since f is monotonic, $f(\bigcup\{Q \mid Q \subseteq f(Q)\}) \subseteq f(f(\bigcup\{Q \mid Q \subseteq f(Q)\}))$. Thus, $f(\bigcup\{Q \mid Q \subseteq f(Q)\}) \in \{Q \mid Q \subseteq f(Q)\}$, which means $f(\bigcup\{Q \mid Q \subseteq f(Q)\}) \subseteq \bigcup\{Q \mid Q \subseteq f(Q)\}$. □

In fact, this proof shows that the second part of the theorem can be strengthened.

- $Q \subseteq f(Q)$ implies $Q \subseteq gfp(f)$, and $f(Q) \subseteq Q$ implies $lfp(f) \subseteq Q$.

For a more detailed discussion of other fixpoint theorems, see [DP90, GS90b].

In a model $\mathcal{M} = (U, \mathcal{I})$, any formula φ defines a set $\varphi^{\mathcal{M}} \subseteq U$ of points in the model, namely $\varphi^{\mathcal{M}} \triangleq \{w \mid (U, \mathcal{I}, w) \models \varphi\}$. Likewise, a formula φ with a free proposition variable q defines a function $\varphi_q^{\mathcal{M}} : U \to U$ from sets of points to sets of points (a *predicate transformer*): If $Q \subseteq U$, then $\varphi_q^{\mathcal{M}}(Q) \triangleq \{w \mid (U, \mathcal{I}', w) \models \varphi\}$, where $\mathcal{I}'$ differs from $\mathcal{I}$ only in $\mathcal{I}'(q) \triangleq Q$.

Lemma 2.8 $(\nu q\ \varphi)^{\mathcal{M}} = gfp(\varphi_q^{\mathcal{M}})$ *and* $(\mu q\ \varphi)^{\mathcal{M}} = lfp(\varphi_q^{\mathcal{M}})$.

PROOF: According to the definitions, $w \in gfp(\varphi_q^{\mathcal{M}})$ iff $w \in \bigcup\{Q \mid Q \subseteq \varphi_q^{\mathcal{M}}(Q)\}$, that is, if there is some $Q \subseteq U$ such that $w \in Q$ and $Q \subseteq \varphi_q^{\mathcal{M}}(Q)$. In **MSOL** this condition can be denoted as $w \models \exists q(q(t_0) \wedge \forall t(q(t) \to \mathbf{MSOL}(\varphi)\{t_0 := t\}))$. This clause is exactly the semantical translation of $w \models \nu q\ \varphi$. For $lfp(\varphi_q^{\mathcal{M}})$, the dual proof holds. □

Call an occurrence of a proposition variable q in a formula φ *positive* or *negative*, if it is under an even or odd number of negations. Formally, this notion is defined recursively: q is positive in the formula q. An occurrence of q in the formula $(\varphi \to \psi)$ is positive, if it is a negative occurrence in φ or a positive occurrence in ψ, and negative, if it is a positive occurrence in φ or a negative occurrence in ψ. An occurrence of q in $\langle R \rangle\ \varphi$ and $\nu q'\ \varphi$ is positive or negative, if it is positive or negative in φ, respectively. A formula φ is called *syntactically monotonic in* q, if every free occurrence of q in φ is positive. It is *syntactically monotonic*, if each sub-formula $\nu q\ \psi$ is syntactically monotonic in q.

Lemma 2.9 *If φ is syntactically monotonic in q, then $\varphi^{\mathcal{M}}_q$ is a monotonic predicate transformer.*

PROOF: This statement can be proved by induction on the structure of φ. The induction basis, namely formulas which are atomic propositions, proposition variables or boolean constants, is immediate. For the inductive step, assume that $P \subseteq Q$. If $(\varphi \to \psi)$ is positive in q, then ψ must be positive and φ must be negative in q. Therefore, $\neg\varphi$ is positive in q. The induction hypothesis is that $\psi^{\mathcal{M}}_q(P) \subseteq \psi^{\mathcal{M}}_q(Q)$ and $\neg\varphi^{\mathcal{M}}_q(P) \subseteq \neg\varphi^{\mathcal{M}}_q(Q)$. From this we can infer that $\varphi^{\mathcal{M}}_q(Q) \subseteq \varphi^{\mathcal{M}}_q(P)$. Therefore, if $\varphi^{\mathcal{M}}_q(P) \subseteq \psi^{\mathcal{M}}_q(P)$ then $\varphi^{\mathcal{M}}_q(Q) \subseteq \psi^{\mathcal{M}}_q(Q)$. This follows from $\varphi^{\mathcal{M}}_q(Q) \subseteq \varphi^{\mathcal{M}}_q(P) \subseteq \psi^{\mathcal{M}}_q(P) \subseteq \psi^{\mathcal{M}}_q(Q)$. In other words, $(\varphi \to \psi)^{\mathcal{M}}_q(P) \subseteq (\varphi \to \psi)^{\mathcal{M}}_q(Q)$. For the case $\langle R \rangle\ \varphi$, the induction hypothesis is that $\varphi^{\mathcal{M}}_q(P) \subseteq \varphi^{\mathcal{M}}_q(Q)$. Then, $\{w \mid \exists w'(w, w') \in \mathcal{I}(R) \wedge w' \in \varphi^{\mathcal{M}}_q(P)\} \subseteq \{w \mid \exists w'(w, w') \in \mathcal{I}(R) \wedge w' \in \varphi^{\mathcal{M}}_q(Q)\}$. In other words, $(\langle R \rangle\ \varphi)^{\mathcal{M}}_q(P) \subseteq (\langle R \rangle\ \varphi)^{\mathcal{M}}_q(Q)$. Similarly, for formulas $\nu q'\varphi$, where q and q' are different variables, the induction hypothesis is that $\varphi^{\mathcal{M}}_{q,q'}(P, X) \subseteq \varphi^{\mathcal{M}}_{q,q'}(Q, X)$ for all X. Therefore, $X \subseteq (\varphi)^{\mathcal{M}}_{q,q'}(P, X)$ implies $X \subseteq (\varphi)^{\mathcal{M}}_{q,q'}(Q, X)$ for all X. Consequently, $\{w \mid \text{for some } X,\ w \in X \text{ and } X \subseteq \varphi^{\mathcal{M}}_{q,q'}(P, X)\} \subseteq \{w \mid \text{for some } X,\ w \in X \text{ and } X \subseteq \varphi^{\mathcal{M}}_{q,q'}(Q, X)\}$. According to the definition, this is the semantics of $(\nu q'\varphi)^{\mathcal{M}}_q(P) \subseteq (\nu q'\varphi)^{\mathcal{M}}_q(Q)$. The last case is $\nu q\varphi$. Since this formula has no free occurrence of variable q, its denotation $(\nu q\varphi)^{\mathcal{M}}_q$ is a constant function. Trivially, constant functions are monotonic. □

Corollary 2.10 *If φ is syntactically monotonic, then*

- $\models (\nu q\ \varphi \leftrightarrow \varphi\{q := \nu q\ \varphi\})$ *and* $\models (\mu q\ \varphi \leftrightarrow \varphi\{q := \mu q\ \varphi\})$.
- *If* $(U, \mathcal{I}) \models (\chi \leftrightarrow \varphi\{q := \chi\})$ *then both* $(U, \mathcal{I}) \models (\chi \to \nu q\ \varphi)$ *and* $(U, \mathcal{I}) \models (\mu q\ \varphi \to \chi)$.

- $(U,\mathcal{I}) \models (\chi \to \varphi\{q := \chi\})$ *implies* $(U,\mathcal{I}) \models (\chi \to \nu q\ \varphi)$, *and*
  $(U,\mathcal{I}) \models (\varphi\{q := \chi\} \to \chi)$ *implies* $(U,\mathcal{I}) \models (\mu q\ \varphi \to \chi)$

PROOF: If φ is syntactically monotonic, then $\varphi_q^{\mathcal{M}}$ is monotonic according to Lemma 2.9. Theorem 2.7 asserts that $gfp(\varphi_q^{\mathcal{M}}) = \varphi_q^{\mathcal{M}}(gfp(\varphi_q^{\mathcal{M}}))$. In the notation of Lemma 2.8, this means $(\nu q\ \varphi)^{\mathcal{M}} = \varphi_q^{\mathcal{M}}((\nu q\ \varphi)^{\mathcal{M}})$. Moreover, $\varphi_q^{\mathcal{M}}((\nu q\ \varphi)^{\mathcal{M}}) = (\varphi_q\{q := \nu q\ \varphi\})^{\mathcal{M}}$. Therefore, $\mathcal{M} \models (\nu q\ \varphi \leftrightarrow \varphi_q\{q := \nu q\ \varphi\})$. The other statements are shown similarly. □

In particular, according to Corollary 2.6, $(\varphi\ \mathbf{U}^+\ \psi)$ and $(\varphi\ \mathbf{W}^+\ \psi)$ in natural models are least and greatest fixed points of $\mathbf{X}\ (\psi \vee \varphi \wedge q)$ and $\widetilde{\mathbf{X}}\ (\psi \vee \varphi \wedge q)$, respectively. Therefore, the following *recursion* and *induction* axioms hold:

- $\models (\varphi\ \mathbf{U}^+\ \psi) \leftrightarrow \mathbf{X}\ (\psi \vee \varphi \wedge (\varphi\ \mathbf{U}^+\ \psi))$ and
  $\models (\varphi\ \mathbf{W}^+\ \psi) \leftrightarrow \widetilde{\mathbf{X}}\ (\psi \vee \varphi \wedge (\varphi\ \mathbf{W}^+\ \psi))$.
- $\models \mathbf{F}^*\ \psi \leftrightarrow (\psi \vee \mathbf{X}\mathbf{F}^*\ \psi)$ and $\models \mathbf{G}^*\ \varphi \leftrightarrow (\varphi \wedge \widetilde{\mathbf{X}}\mathbf{G}^*\ \psi)$.
- $(U,\mathcal{I}) \models (\mathbf{X}\ (\psi \vee \varphi \wedge \chi) \to \chi)$ implies $(U,\mathcal{I}) \models ((\varphi\ \mathbf{U}^+\ \psi) \to \chi)$, and
  $(U,\mathcal{I}) \models (\chi \to \widetilde{\mathbf{X}}\ (\psi \vee \varphi \wedge \chi))$ implies $(U,\mathcal{I}) \models (\chi \to (\varphi\ \mathbf{W}^+\ \psi))$.

As we have shown, monotonic $\mu\mathbf{TL}$ formulas denote greatest or least fixed points of predicate transformers. For nonmonotonic formulas, the existence of fixed points is not granted. For example, there is no $Q \subseteq U$ satisfying $Q = U \setminus Q$; thus, there is no fixed point of $(\neg q)_q^{\mathcal{M}}$. However, the **MSOL** semantics of $\nu q\ \neg q$ is $\exists q(q(t_0) \wedge \forall t(q(t) \to \neg q(t)))$, which is equivalent to the well-defined value $\bot$. On general Kripke-models, monotonic $\mu\mathbf{TL}$ is stricly weaker in expressiveness than unrestricted $\mu\mathbf{TL}$. Even unrestricted $\mu\mathbf{TL}$ can, in turn, express fewer properties of Kripke models than monadic second order logic:

Lemma 2.11 *Consider the class of all Kripke models.*
(a) There is no monotonic $\mu\mathbf{TL}$ *formula equivalent to* $\nu q(\langle R\rangle\ \neg q)$.
(b) There is no $\mu\mathbf{TL}$ *formula which is equivalent to* $\forall t \mathtt{p}(t)$

PROOF: For (a), consider $\varphi \triangleq \nu q(\langle R\rangle\ \neg q)$. Then $\mathbf{MSOL}(\varphi) = \exists q(q(t_0) \wedge \forall t(q(t) \to \exists t'(tRt' \wedge \neg q(t'))))$. Choosing $\mathcal{I}(q) \triangleq \{\mathcal{I}(t_0)\}$ shows that $w \models \varphi$ iff $w \models \exists t(t_0 R t \wedge t_0 \neq t)$. There is no monotonic formula which can express this property: Consider the model $\mathcal{M} \triangleq (U,\mathcal{I})$, where $U \triangleq \{w_0, w_1\}$, $\mathcal{I}(R) \triangleq \{(w_0,w_0),(w_0,w_1),(w_1,w_1)\}$ and $\mathcal{I}(\mathtt{p}) = \mathcal{I}(q) = \emptyset$ for all $\mathtt{p} \in \mathcal{P}$ and $q \in \mathcal{Q}$. Then $w_0 \models \varphi$ and $w_1 \not\models \varphi$. For each monotonic formula ψ, however, it holds that $w_0 \models \psi$ iff $w_1 \models \psi$. To prove this, we show by induction on the

structure of ψ that $\psi^{\mathcal{M}} = \emptyset$ or $\psi^{\mathcal{M}} = U$. For propositional formulas, this is immediate; the case $\langle R \rangle\ \psi$ follows from the definition of $\mathcal{M}$. The only remaining case are formulas $\nu q \psi$. According to the induction hypothesis, either $(\psi\{q := \top\})^{\mathcal{M}} = \emptyset$ or $(\psi\{q := \top\})^{\mathcal{M}} = U$. In the first case, from the fact that $(\nu q \psi)^{\mathcal{M}} \subseteq \top^{\mathcal{M}}$ and monotonicity of ψ we infer that $(\psi\{q := \nu q \psi\})^{\mathcal{M}} \subseteq (\psi\{q := \top\})^{\mathcal{M}} = \emptyset$. The first part of Theorem 2.7 implies that $(\nu q \psi)^{\mathcal{M}} \subseteq (\psi\{q := \nu q \psi\})^{\mathcal{M}}$; therefore, $(\nu q \psi)^{\mathcal{M}} = \emptyset$. In the second case, $U = \top^{\mathcal{M}} = (\psi\{q := \top\})^{\mathcal{M}}$. With the second part of Theorem 2.7, it follows that $\top^{\mathcal{M}} \subseteq (\nu q \psi)^{\mathcal{M}}$, i.e., $(\nu q \psi)^{\mathcal{M}} = U$.

Statement (b) holds since the truth of $\mu\mathbf{TL}$ formulas is preserved under disjoint unions of models, whereas $\varphi \triangleq \forall t \mathtt{p}(t)$ can be invalidated by adding an isolated point w with $w \not\models \mathtt{p}$. Formally, consider the models $\mathcal{M}_0 \triangleq (U_0, \mathcal{I}, w_0)$ and $\mathcal{M}_1 \triangleq (U_1, \mathcal{I}, w_0)$, where $U_0 \triangleq \{w_0\}$, $U_1 \triangleq \{w_0, w\}$, $\mathcal{I}(R) \triangleq \emptyset$ and $\mathcal{I}(\mathtt{p}) \triangleq \mathcal{I}(q) \triangleq \{w_0\}$. Then $\mathcal{M}_0 \models \varphi$ and $\mathcal{M}_1 \not\models \varphi$, whereas for each $\mu\mathbf{TL}$ formula ψ it holds that $\mathcal{M}_0 \models \varphi$ iff $\mathcal{M}_1 \models \varphi$. □

If the model is *connected* (that is, $\forall w, w'(w < w' \vee w = w' \vee w > w')$), then every point is reachable from the current point. In this case, the operator $\mathbf{G}^{\pm}$ can replace the first-order universal quantifier: $\mathcal{M} \models \forall t \mathtt{p}(t)$ iff $\mathcal{M} \models \mathbf{G}^{\pm}\ \mathtt{p}$. In this case,

$$\begin{array}{lll} \mathcal{M} \models \nu q\ \varphi & \text{iff} & \mathcal{M} \models \exists q(q \wedge \mathbf{G}^{\pm}\ (q \rightarrow \varphi)), \\ \mathcal{M} \models \mu q\ \varphi & \text{iff} & \mathcal{M} \models \forall q(\mathbf{G}^{\pm}\ (\varphi \rightarrow q) \rightarrow q). \end{array}$$

Hence, on natural models $\mu\mathbf{TL}$ is at most as expressive as **qTL** (and **MSOL**). Since $\mu\mathbf{TL}$ does not contain any past-operators, there is no $\mu\mathbf{TL}$ formula which is equivalent to $\mathbf{F}^{-}\ \top$. Subsequently, however, we will show that for initial validity in natural models a converse translation from **qTL** (or **MSOL**) into monotonic $\mu\mathbf{TL}$ exists. Since the proof uses *ω-regular languages* and *ω-automata*, it is postponed.

2.5 Automata and Logics

Let $(U, \mathcal{I})$ be universe and interpretation of a natural model. Formula φ is *initially valid* in $(U, \mathcal{I})$, if $(U, \mathcal{I}, w_0) \models \varphi$, where w_0 is the unique initial point of U (which has no predecessors). For any $(U, \mathcal{I})$ it holds that φ is universally valid iff $\mathbf{G}^{*}\ \varphi$ is initially valid, and φ is initially valid iff $(\mathbf{G}^{-}\ \bot \rightarrow \varphi)$ is universally valid.

Given a (finite or infinite) natural model $\mathcal{M} \triangleq (U, \mathcal{I}, w_0)$, the interpretation $\mathcal{I}$ defines a mapping $\mathcal{I} : \mathcal{P} \rightarrow 2^U$ from propositions into subsets of

the universe. Define a *labelling function* $\mathcal{L} : U \to 2^{\mathcal{P}}$ by

$$\mathtt{p} \in \mathcal{L}(w) \qquad \text{iff} \qquad w \in \mathcal{I}(\mathtt{p})$$

That is, $\mathcal{L}(w) \triangleq \{\mathtt{p} \mid w \in \mathcal{I}(\mathtt{p})\}$ is the *label* of point $w \in U$. If $U = (w_0, w_1, w_2, ...)$, then the sequence $\sigma = (\mathcal{L}(w_0), \mathcal{L}(w_1), \mathcal{L}(w_2), ...)$ is called the *ω-word* of $\mathcal{M}$ over the *alphabet* $\Sigma \triangleq 2^{\mathcal{P}}$. A set of ω-words is called an *ω-language*.

We say that a linear-time logic formula *defines* the set of all natural models in which it is initially valid. Thus every such formula defines the ω-language given by these models. In order to define languages by formulas it suffices to restrict attention to the future fragment of temporal logic:

Lemma 2.12 *For any* **LTL** *or* **qTL** *formula φ there exists an* **LTL** *or* **qTL** *future formula (without $\mathbf{U}^-$-operators) defining the same language.*

PROOF: It can be shown that any **LTL**-formula can be separated into a boolean combination of pure future, pure present and pure past formulas (for a proof of this statement, see [Gab89, GHR94, CS01]). This can be extended to **qTL**: Note that $\exists q(\varphi \vee \psi) \leftrightarrow (\exists q\ \varphi \vee \exists q\ \psi)$. Moreover, if φ_1,...,φ_n are pure past, and ψ_1,...,ψ_m are pure present or future, then $\exists q(\varphi_1 \wedge ... \wedge \varphi_n \wedge \psi_1 \wedge ... \wedge \psi_m)$ is equivalent to $\exists q(\varphi_1 \wedge ... \wedge \varphi_n) \wedge \exists q(\psi_1 \wedge ... \wedge \psi_m)$. If the past-formulas are true in a model $(U, \mathcal{I}, w_0)$ where $\mathcal{I}(q) \triangleq Q_1$ and the present- and future-formulas are true if $\mathcal{I}(q) \triangleq Q_2$, then the conjunction of past, present and future part is true if $\mathcal{I}(q) \triangleq \{w \mid w < w_0\} \cap Q_1 \cup \{w \mid w \geq w_0\} \cap Q_2$. Thus, using disjunctive normal form, propositionally quantified formulas can be separated. Given a separated formula φ, let φ^+ be the formula φ where every sub-formula $(\varphi\ \mathbf{U}^-\ \psi)$ is replaced by $\bot$. Then φ is initially valid in any natural model $\mathcal{M}$ iff ψ^+ is initially valid in $\mathcal{M}$. Thus φ and φ^+ define the same language. □

Languages can also be defined by *(ω-)regular expressions* and by *finite (ω-) automata*.

Trivially, ω-automata are closed under projection onto a smaller alphabet. Büchi [Büc62] showed that his automata are closed under complement; this is a highly nontrivial proof.

Closure under complement can be used to show that Büchi-automata are at least as expressive as **qTL**.

Lemma 2.13 *For every* **qTL** *formula there is a Büchi-automaton defining the same language.*

PROOF: According to Lemma 2.12 it suffices to give a translation for formulas without $\mathbf{U}^-$. Automata for propositions are trivial two-state machines (with $2^{|\mathcal{P}|-1}$ transitions between this two states). An automaton for $\bot$ is one which never accepts. From an automaton for φ, an automaton for $\mathbf{X}\ \varphi$ and $\mathbf{F}^+\ \varphi$ can be built by an appropriate prefixing with a single step or loop on the initial states. According to Lemma 2.5, $\mathbf{U}^+$ can be expressed with $\mathbf{X}$, $\mathbf{F}^+$ and second order quantification. Implications $(\varphi \rightarrow \psi)$ can be written as $(\neg\varphi \vee \psi)$ and thus be reduced to unions and complements. Finally, existential second order quantification amounts to the projection of the automaton onto a smaller alphabet. □

In particular, since **LTL** is a sublanguage of **qTL**, for every **LTL** formula there is a corresponding Büchi-automaton. In Section 4.2, we will decribe a tableaux decision procedure, which can be seen as an efficient algorithm to construct a Büchi-automaton from a formula.

ω-regular expressions are at most as expressive as $\mu\mathbf{TL}$:

Lemma 2.14 *For every ω-regular expression there exists a $\mu\mathbf{TL}$-formula describing the same language.*

PROOF: The proof associates with every ω-regular expression φ a $\mu\mathbf{TL}$-formula $\mu\mathbf{TL}(\varphi)$ with at most one free proposition variable q indicating the end of the sequence.

- $\mu\mathbf{TL}(P) \triangleq (\bigwedge_{\mathtt{p}\in P} \mathtt{p} \wedge \bigwedge_{\mathtt{p}\notin P} \neg\mathtt{p} \wedge q), \quad \text{if } P \in 2^{\mathcal{P}}$
- $\mu\mathbf{TL}(\epsilon) \triangleq \bot$
- $\mu\mathbf{TL}(\varphi + \psi) \triangleq (\mu\mathbf{TL}(\varphi) \vee \mu\mathbf{TL}(\psi))$
- $\mu\mathbf{TL}(\varphi;\psi) \triangleq \mu\mathbf{TL}(\varphi)\{q :=\mathbf{X}\ \mu\mathbf{TL}(\psi)\}$
- $\mu\mathbf{TL}(\varphi^+) \triangleq \mu q_1\ (\mu\mathbf{TL}(\varphi)\{q := q \vee\ \mathbf{X}\ q_1\})$
- $\mu\mathbf{TL}(\varphi^\omega) \triangleq \nu q_1\ (\mu\mathbf{TL}(\varphi)\{q :=\mathbf{X}\ q_1\})$

To be able to deal with finite strings as well, the $\mu\mathbf{TL}$-formula corresponding to an ω-regular expression φ is defined as $\mu\mathbf{TL}(\varphi)\{q :=\mathbf{X} \bot\}$. It can be shown that this formula defines the same language as the original ω-regular expression φ. □

As an example, consider the expression $(\neg\mathtt{p1})^\omega + (\top^+;\mathtt{p2})^\omega$.

$$\mu\mathbf{TL}((\neg\mathtt{p1})^\omega + (\top^+;\mathtt{p2})^\omega)$$
$$= \nu q_1\ (\mu\mathbf{TL}(\neg\mathtt{p1})\{q :=\mathbf{X}\ q_1\}) \vee \nu q_2(\mu\mathbf{TL}(\top^+;\mathtt{p2})\{q :=\mathbf{X}\ q_2\})$$

$$
\begin{aligned}
&= \nu q_1\ ((\neg \mathtt{p1} \wedge q)\{q := \mathbf{X}\ q_1\}) \vee \\
&\quad \nu q_2((\mu \mathbf{TL}(\top^+)\{q := \mathbf{X}\ \mu \mathbf{TL}(\mathtt{p2})\})\{q := \mathbf{X}\ q_2\}) \\
&= \nu q_1\ (\neg \mathtt{p1} \wedge \mathbf{X}\ q_1) \vee \\
&\quad \nu q_2(\mu q_3(\top \wedge q)\{q := q \vee \mathbf{X}\ q_3\}\{q := \mathbf{X}\ (\mathtt{p2} \wedge q)\}\{q := \mathbf{X}\ q_2\}) \\
&= \nu q_1\ (\neg \mathtt{p1} \wedge \mathbf{X}\ q_1) \vee \nu q_2(\mu q_3(\mathbf{X}\ (\mathtt{p2} \wedge q) \vee \mathbf{X}\ q_3)\{q := \mathbf{X}\ q_2\}) \\
&= \nu q_1\ (\neg \mathtt{p1} \wedge \mathbf{X}\ q_1) \vee \nu q_2(\mu q_3\ \mathbf{X}\ (\mathtt{p2} \wedge \mathbf{X}\ q_2 \vee q_3)) \\
&\leftrightarrow \nu q_1\ (\neg \mathtt{p1} \wedge \mathbf{X}\ \top \wedge \mathbf{X} q_1) \vee \nu q_2(\mu q_3\ \mathbf{X}\ (\mathtt{p2} \vee q_3) \wedge \mathbf{X} q_2) \\
&= \mathbf{G}^*\ (\neg \mathtt{p1} \wedge \mathbf{X}\ \top) \vee \mathbf{G}^* \mathbf{F}^+\ \mathtt{p2}
\end{aligned}
$$

This lemma closes the circle in the expressiveness results of second order languages.

Theorem 2.15 (Büchi, Wolper, Sistla) *To define ω-languages, the following formalisms are of equal expressive power:*

$$\mu\mathbf{TL} = \mathbf{qTL} = \mathbf{MSOL} = \text{Büchi-automata} = \omega\text{-regular expressions}$$

PROOF: For every $\mu\mathbf{TL}$-formula there exists an equivalent $\mathbf{qTL}$-formula by definition; on natural models $\mathbf{qTL}$ is equal in expressiveness to $\mathbf{MSOL}$ by Lemma 2.4; according to Lemma 2.13, for every $\mathbf{qTL}$ (or $\mathbf{MSOL}$) formula there is a Büchi-automaton defining the set of its models; by Lemma 1.3, Büchi-automata are equivalent to ω-regular expressions; and these in turn can be described by $\mu\mathbf{TL}$-formulas as shown in Lemma 2.14. □

Similar results can be proved about logics with past operators on integer models (bi-infinite words) and two-way automata, and about branching time logics ($\mu\mathbf{TL}/\mathbf{qTL}$ on tree models) and tree automata ($\Delta \subseteq S \times 2^{\mathcal{P}} \times (\mathcal{R} \times S)^n$) (see [Niw88, Tho90, Sch92b]).

2.6 Relational μ-calculus

In this section we introduce yet another, richer logical language which is closer to functional and logic programming paradigm. This is the relational μ-calculus defined in [Par74]. Informally, this μ-calculus can be seen as first order predicate logic with an additional recursion operator.

A (typed) *structure* S consists of a collection of disjoint sets called *domains*, and a collection of functions and relations over these domains. (In some textbooks, structures are called *algebras*.) Elements of the domains are called *objects*. Models for propositional temporal logics can be regarded to be special structures with a single domain U, no functions, unary predicates $P \subseteq U$ and binary relations $R \subseteq U \times U$ on this domain.

A *signature* $\Sigma = (\mathcal{D}, \mathcal{F}, \mathcal{R})$ consists of a finite set $\mathcal{D}$ of *domain names*, a finite set $\mathcal{F}$ of *function symbols*, and a finite set $\mathcal{R}$ of *relation symbols*. Associated with each function and relation symbol is its *type* τ, which is a sequence of domain names (nonempty in the case of functions). Unary relation symbols are called *predicate symbols*, nullary function symbols are called *constant symbols*. A *constant term* of type D is either a constant symbol of type D, or of the form $ft_0...t_{n-1}$, where f is a function symbol of type $(D_0, ..., D_{n-1}, D)$ and each t_i is a constant term of type D_i for $i < n$.

An *interpretation* $\mathcal{I}$ for a signature Σ on a structure S is a mapping $\mathcal{I} : \Sigma \to S$ assigning a nonempty domain $D^{\mathcal{S}} \triangleq \mathcal{I}(D)$ for each domain name D, and a function $f^S \triangleq \mathcal{I}(f)$ and relation $R^S \triangleq \mathcal{I}(R)$ of appropriate type for each function and relation symbol, respectively. That is, if $\tau(f) = (D_0, ..., D_n)$, then $f^S \in (D_0^S \times \cdots \times D_{n-1}^S \to D_n^S)$, and if $\tau(R) = (D_1, ..., D_n)$, then $R^S \subseteq (D_1^S \times \cdots \times D_n^S)$.

It is important to distinguish between signatures (syntax) and structures (semantics). Given a signature, the set of structures for this signature forms the set of possible models. For example, each signature can both be interpreted in a structure consisting of a single one-element domain, and in a structure containing a unique object for each constant term, the so-called *term-algebra* or *Herbrand structure*. Vice versa, given a specific structure, then each signature determines which parts of this structure are "visible" in a specification formalism. As an example, consider the structure consisting of all functions and relations over the domain of natural numbers. Truth of logical formulas in the signature $\Sigma_1 \triangleq (\mathbf{N}, \{\}, \{\prec, <, =\})$ is decidable, whereas the set of valid formulas in $\Sigma_2 \triangleq (\mathbf{N}, \{+, *\}, \{=\})$ is not recursively enumerable.

Given a signature Σ, let $\mathcal{V}$ be a set of variables, each of which is either an *individual variable* or a *relation variable*. Again, we assume that each variable has an appropriate type. An *object term* t of type D is

- x, where x is an individual variable of type D, or
- $ft_0...t_{n-1}$, where f is a function symbol of type $(D_0, ..., D_{n-1}, D)$ and each t_i is an object term of type D_i for $i < n$.

As a special case, each constant symbol of type (D) is an object term of type D. In the relational μ-calculus, there are two more syntactic categories: *well-formed formulas* and *relation terms of type* τ. Assuming that the symbols $(,)$, $\bot$, $\to$, $=$, $\exists$, μ and λ are not in the signature, a *well formed formula* φ is built according to the following syntax:

- $\bot$, $(\varphi \to \psi)$, where φ and ψ are well formed formulas,

- $t_1 = t_2$, where t_1 and t_2 are object terms of the same type,
- $\exists x\ \varphi$, where φ is a well formed formula, and x is an individual variable, or
- $\rho t_1 ... t_n$, where ρ is a relation term of type $(D_1, ..., D_n)$ (see below), and t_i is an object term of type D_i for all $i \leq n$.

In first order logic, a *relation term* is just a relation symbol from the signature. In second order logic, a *relation term* can either be a relation symbol or a relation variable $q \in \mathcal{Q}$. In the relational μ-calculus, more complex relations can be specified via *λ-abstraction* and *μ-recursion*. In this calculus, a *relation term* ρ of type $(D_1, ..., D_n)$ is

- a relation symbol R or relation variable X of type $(D_1, ..., D_n)$,
- $\lambda x_1 ... x_n(\varphi)$, where φ is a well formed formula and each x_i is an individual variable of type D_i, or
- $\mu X(\rho)$, where X is a relation variable of type $(D_1, ..., D_n)$, and ρ a relation term of the same type which is syntactically monotone in X.

As above, in this definition ρ is defined to be *syntactically monotone* in X, if every occurrence of X is under an even number of negation signs. Syntactical monotonicity ensures that the functional defined by ρ is monotone in the lattice of values for X and thus the least fixpoint of this functional exists.

A *variable valuation* $\mathbf{v}$ is a mapping assigning an object $\mathbf{v}(x) \in D$ to every individual variable x of type D, and a set $\mathbf{v}(X) \subseteq D_1 \times \cdots \times D_n$ to every relation variable X of type $(D_1, ..., D_n)$. A *relational model* $\mathcal{M} \triangleq (S, \mathcal{I}, \mathbf{v})$ for the signature Σ consists of a structure S, an interpretation $\mathcal{I}$, and a variable valuation $\mathbf{v}$. Similar to first order and temporal logics, we say that the model $\mathcal{M} \triangleq (S, \mathcal{I}, \mathbf{v})$ is *based* on the *frame* $\mathcal{F} \triangleq (S, \mathcal{I})$.

Any relational model $\mathcal{M} \triangleq (S, \mathcal{I}, \mathbf{v})$ determines a unique object $t^{\mathcal{M}}$ for every object term t, a relation $\rho^{\mathcal{M}}$ of appropriate type for each relation term ρ, and a unique truth value $\varphi^{\mathcal{M}} \in \{\mathbf{true}, \mathbf{false}\}$ for any formula φ. This *denotation* of terms and formulas is defined in the usual way:

- $x^{\mathcal{M}} \triangleq \mathbf{v}(x)$, if $x \in \mathcal{V}$ is an individual variable, and
- $(f t_1 ... t_n)^{\mathcal{M}} \triangleq \mathcal{I}(f)(t_1^{\mathcal{M}} ... t_n^{\mathcal{M}})$, i.e., the value of function f^S at $(t_1^{\mathcal{M}} ... t_n^{\mathcal{M}})$.
- $\bot^{\mathcal{M}} \triangleq \mathbf{false}$,

- $(\varphi \rightarrow \psi)^{\mathcal{M}} = \mathbf{true}$ iff $\varphi^{\mathcal{M}} = \mathbf{true}$ implies $\psi^{\mathcal{M}} = \mathbf{true}$,
- $(t_1 = t_2)^{\mathcal{M}} = \mathbf{true}$ iff $t_1^{\mathcal{M}} = t_2^{\mathcal{M}}$; i.e., iff t_1 and t_2 denote the same object in S,
- $(\exists x\ \varphi)^{\mathcal{M}} = \mathbf{true}$ iff $\varphi^{(S,\mathcal{I},\mathbf{v}')} = \mathbf{true}$ for some valuation $\mathbf{v}'$ which differs from $\mathbf{v}$ at most in x,
- $(\rho t_1 ... t_n)^{\mathcal{M}} = \mathbf{true}$ iff $(t_1^{\mathcal{M}}, ..., t_n^{\mathcal{M}}) \in \rho^{\mathcal{M}}$,
- $R^{\mathcal{M}} \triangleq \mathcal{I}(R)$, if R is a relation symbol,
- $X^{\mathcal{M}} \triangleq \mathbf{v}(X)$, if X is a relation variable,
- $(\lambda x_1 ... x_n\ \varphi)^{\mathcal{M}} \triangleq \{(d_1, ..., d_n) \mid \varphi^{\mathcal{F}}(d_1, ..., d_n) = \mathbf{true}\}$, where $\varphi^{\mathcal{F}}(d_1, ..., d_n) \triangleq \varphi^{(S,\mathcal{I},\mathbf{v}')}$ and $\mathbf{v}'$ differs from $\mathbf{v}$ only in the assignment of d_i to x_i for $1 \leq i \leq n$; i.e., $(\lambda x_1 ... x_n(\varphi))^{\mathcal{M}}$ is the relation consisting of all tuples of objects for which φ is **true**, and
- $(\mu X\ \rho)^{\mathcal{M}} \triangleq \bigcap\{Q \mid \rho^{\mathcal{F}}(Q) \subseteq Q\}$, where $\rho^{\mathcal{F}}(Q) \triangleq \rho^{(S,\mathcal{I},\mathbf{v}')}$, and $\mathbf{v}'$ differs from $\mathbf{v}$ only in $\mathbf{v}'(X) = Q$;
  i.e., $\mu X(\rho)^{\mathcal{M}}$ is the least fixpoint of the relation functional ρ.

For closed terms t (not containing any free variables), the denotation $t^{\mathcal{M}}$ is independent of the variable valuation $\mathbf{v}$. Assume that the object term t of type D contains free individual variables $x_1, ..., x_n$ of type $D_1, ..., D_n$. For any model $\mathcal{M}$, the denotation $t^{\mathcal{M}}$ is an object in the domain $\mathcal{I}(D)$. For any frame $\mathcal{F} \triangleq (S,\mathcal{I})$, however, we define $t^{\mathcal{F}}$ to be a function $t^{\mathcal{F}} : \mathcal{I}(D_1) \times \cdots \times \mathcal{I}(D_n) \rightarrow \mathcal{I}(D)$. The function value is given by $t^{\mathcal{F}}(d_1, \ldots, d_n) = t^{(S,\mathcal{I},\mathbf{v})}$, where $\mathbf{v}$ is any valuation assigning d_i to x_i for $1 \leq i \leq n$. Similarly, if t is a relation term which contains a relation variable X of type $(D_1, ..., D_n)$, then $t^{\mathcal{F}}$ is a functional which maps subsets of $\mathcal{I}(D_1) \times \cdots \times \mathcal{I}(D_n)$ into $\mathcal{I}(D)$.

The relational operators λ and μ are similar to the operators used in λ-calculus and in denotational semantics. In fact, we could define well formed formulas to be object terms of a special type **boolean**. Relation terms could then be defined as function terms with boolean result, and the λ abstraction builds such a function term from a boolean object term.

The relational μ-calculus extends first order logic in a similar way as the propositional μ calculus extends modal logic. In fact, the standard translation from modal into first order logic can be trivially extended into a standard translation from propositional into relational μ calculus. In addition, the relational μ calculus offers some restricted form of non-monadic second order quantification. It contains classical first-order logic as a sublanguage.

Note, however, that in the relational μ-calculus there is no λ-abstraction on function or relation variables. This would result in a second-order calculus. In contrast to second order logic, there is no μ-calculus formula expressing that domain D is finite [Par74]. On the other hand, the minimization operator can be expressed in second order logic:

$$\mu X(\rho)\vec{x} \leftrightarrow \forall X(\forall\vec{x}(\rho\vec{x} \rightarrow X\vec{x}) \rightarrow X\vec{x})$$

Since the induction axiom for arithmetic can be formulated as a least fixpoint formula, the natural numbers have a categorical theory in the relational μ-calculus (for details, see also [Par74]). Therefore, the set of valid formulas is not recursively enumerable, and its expressiveness lies properly in between first and second order logic.

The μ-recursion operator can be used to give recursive definitions of boolean functions, similar to the use of recursion in functional and logic programming. As an example, Park defines the addition-relation on natural numbers from the constant 0 and the predecessor relation $\prec$ recursively as follows:

$$a + b = c \Leftrightarrow ((a = 0 \wedge b = c) \vee \exists uv(u \prec a \wedge v \prec c \wedge u + b = v)).$$

In the relational μ-calculus, this definition can be written as

$$a + b = c \Leftrightarrow \mu X(\lambda xyz(x = 0 \wedge y = z) \vee \exists uv(u \prec x \wedge v \prec z \wedge Xuyv))abc.$$

Using the fixpoint unfolding $\mu X\rho \leftrightarrow \rho\{X := \mu X\rho\}$ we can calculate the truth value of $1 + 2 = 3$ as follows. Let $Add \triangleq \mu X\rho$, where ρ is the relation term

$$\lambda xyz((x = 0 \wedge y = z) \vee \exists uv(u \prec x \wedge v \prec z \wedge Xuyv)).$$

The calculation is as follows.

$$\begin{aligned}
&Add\ 123\\
&\leftrightarrow \rho\{X := Add\}\ 123\\
&\leftrightarrow \lambda xyz((x = 0 \wedge y = z) \vee \exists uv(u \prec x \wedge v \prec z \wedge Add\ uyv))123\\
&\leftrightarrow ((1 = 0 \wedge 2 = 3) \vee \exists uv(u \prec 1 \wedge v \prec 3 \wedge Add\ u2v))\\
&\leftrightarrow Add\ 022\\
&\leftrightarrow \lambda xyz((x = 0 \wedge y = z) \vee \exists uv(u \prec x \wedge v \prec z \wedge Add\ uyv))022\\
&\leftrightarrow ((0 = 0 \wedge 2 = 2) \vee \exists uv(u \prec 0 \wedge v \prec 2 \wedge Add\ u2v))\\
&\leftrightarrow \top.
\end{aligned}$$

As another example, consider the following PROLOG program.

```
parent(john, mary).
parent(jane, john).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
?- ancestor(jane, mary).
```

This PROLOG query can be translated into a formula of the relational μ-calculus. The signature consists of a single domain $\mathcal{D} \triangleq \{\texttt{person}\}$, constant symbols $\mathcal{F} \triangleq \{\texttt{john}, \texttt{mary}, \texttt{jane}\}$ of type $\texttt{person}$, and relation symbols $\mathcal{R} \triangleq \{\texttt{parent}, \texttt{ancestor}\}$ of type $(\texttt{person}, \texttt{person})$. The "closed world assumption" in PROLOG guarantees that these relations are completely described by the above clauses; therefore `parent` and `ancestor` are the minimal relations satisfying these clauses. In the relational μ-calculus, this is expressed by

$$\begin{aligned}&\texttt{parent} \equiv \lambda xy\,(x = \texttt{john} \wedge y = \texttt{mary} \vee x = \texttt{jane} \wedge y = \texttt{john})\\&\texttt{ancestor} \equiv \mu X\,(\lambda xy\,(\texttt{parent}\ xy \vee \exists z\,(\texttt{parent}\ xz \wedge Xzy))).\end{aligned}$$

Here, $R \equiv S$ abbreviates the condition $\forall xy\,(Rxy \leftrightarrow Sxy)$. Using this as a rewrite rule, the formula $\texttt{ancestor jane mary}$ evaluates as follows:

$$\begin{aligned}&\texttt{ancestor jane mary}\\&\leftrightarrow \mu X\,(\lambda xy\,(\texttt{parent}\ xy \vee \exists z\,(\texttt{parent}\ xz \wedge Xzy)))\ \texttt{jane mary}\\&\leftrightarrow \lambda xy\,(\texttt{parent}\ xy \vee \exists z\,(\texttt{parent}\ xz \wedge\ \texttt{ancestor}\ zy))\ \texttt{jane mary}\\&\leftrightarrow (\texttt{parent jane mary} \vee \exists z\,(\texttt{parent jane}\ z \wedge\ \texttt{ancestor}\ z\ \texttt{mary}))\\&\leftrightarrow ((\texttt{jane} = \texttt{john} \wedge \texttt{mary} = \texttt{mary} \vee \texttt{jane} = \texttt{jane} \wedge \texttt{mary} = \texttt{john}) \vee\\&\qquad\qquad \exists z\,((\texttt{jane} = \texttt{john} \wedge z = \texttt{mary} \vee \texttt{jane} = \texttt{jane} \wedge z = \texttt{john}) \wedge \texttt{ancestor}\ z\ \texttt{mary})\\&\leftrightarrow \texttt{ancestor john mary}\\&\leftrightarrow (\texttt{parent john mary} \vee \exists z\,(\texttt{parent john}\ z \wedge \texttt{ancestor}\ z\ \texttt{mary}))\\&\leftrightarrow \top\end{aligned}$$

Similarly, the term $\lambda x(\texttt{ancestor jane}\ x)$ evaluates to $\{\texttt{john}, \texttt{mary}\}$. More complicated properties like $\forall x \exists y\ (x \neq y \wedge \neg\texttt{ancestor}\ xy)$ can be formulated in the relational μ-calculus. In general, such formulas need more powerful proof principles than λ-substitution and μ-unfolding. In the above example, all domains are finite. Therefore, the result to this and similar queries can be calculated completely automatic by a model checker. The respective algorithm will be given in section 6.5.

Chapter 3

Model Transformations

As we have seen, linear temporal formulas and ω-automata both can be used to describe sets of infinite sequences. The practical difference is, that logic tends to be more "descriptive", specifying *what* a system should do, whereas automata tend to be more "machine-oriented", indicating *how* it should be done. Logical formulas are "global", they are interpreted on the whole structure, whereas automata are "local", describing single states and transitions.

Therefore, traditionally automata or related models are used to give an abstract account of the *system* to be verified, whereas formulas are used to specify *properties* of these systems. But, since it is possible to translate between automata and formulas and back, the choice is a matter of complexity, of available algorithms and of taste. We could equally well define both system and properties in temporal logic; in this case we would have to prove an implication formula (Chapter 4.2 will explain how to do this). Another alternative is that both the implementation and the specification are given as automata, where the latter is more "abstract" than the former. Then we have to prove that one can *simulate* the other.

In the next sections, we describe various transformations between models such as simulations, refinements and collapses, and investigate the preservation of logical properties under these transformations.

3.1 Models, Automata and Transition Systems

The previous chapter related ω-automata and linear temporal formulas via the ω-language accepted by the automaton and the set of natural models in which the formula is initially valid. There is, however, a more direct

connection on the structural level. Let $\mathcal{M} = (U, \mathcal{I}, w_0)$ be a Kripke-model with predicates from $\mathcal{P}$ and accessibility relations from $\mathcal{R}$. Consider the alphabet $\Sigma = 2^{\mathcal{P}} \times \mathcal{R}$, and let $\sigma = (\sigma_0\sigma_1\sigma_2...)$ be an ω-word, where $\sigma_i = (P_i, R_i)$. We say that σ is *generated by* $\mathcal{M}$ if there exists a mapping ρ from letters of σ into points of U, such that

1. $\rho(\sigma_0) = w_0$,
2. if $\rho(\sigma_i) = w$, then $P_i = \mathcal{L}(w)$,
3. if $\rho(\sigma_i) = w$ and $\rho(\sigma_{i+1}) = w'$, then $(w, w') \in \mathcal{I}(R_i)$, and
4. if σ is finite with last letter σ_n, and $\rho(\sigma_n) = w$, then w is terminal (i.e., there is no w' such that $w \prec w'$).

(Recall that $\mathcal{L}(w) \triangleq \{\mathtt{p} \mid \mathtt{p} \in \mathcal{I}(w)\}$ is the label of point w.) Condition 4. guarantees that generated words represent maximal paths in the model[1]. Define the *language generated by* $\mathcal{M}$ to be the set of all ω-words generated by $\mathcal{M}$. With these definitions, Kripke-models can be regarded as weakly fair transition systems for the alphabet $\Sigma = 2^{\mathcal{P}} \times \mathcal{R}$. (Recall that a weakly fair transition system is a fair transition system where all states are recurring, and all terminal states are accepting.)

Lemma 3.1 *For any Kripke-model $\mathcal{M} = (U, \mathcal{I}, w_0)$ there exists a weakly fair transition system $\mathcal{M}_{\mathcal{A}} = (S, \Delta, S_0)$, such that the language generated by $\mathcal{M}$ is equal to the language accepted by $\mathcal{M}_{\mathcal{A}}$.*

PROOF: To prove this lemma, there are several alternative constructions. One possibility is to define $S \triangleq U \cup \{stop\}$, where *stop* is a special accepting state for finite paths. Furthermore, $S_0 \triangleq \{w_0\}$, and $(w, (P, R), s) \in \Delta$ iff $w \in U$, $\mathcal{L}(w) = P$, and either $(w, s) \in \mathcal{I}(R)$ or w is terminal and $s = stop$. Then, $\mathcal{M}_{\mathcal{A}}$ accepts exactly the set of all natural models which are generated by $\mathcal{M}$. □

Thus, models can be seen as automata. Likewise, formulas can be seen as automata: In the previous section we observed that for every **LTL** formula there exists an equivalent Büchi-automaton. Since this proof is constructive, it yields a method to obtain such an automaton. However, a much more concise way of constructing it is the tableau construction sketched in Chapter 4.2 below.

[1] Some texts omit this condition, with the consequence that all prefixes of a generated word are also generated. Other authors impose the even stronger condition that all generated words must be infinite; this implies that all points in a model should be nonterminal.

Let $\mathcal{M}$ be a Kripke-model with a single accessibility relation, and φ be an **LTL**-formula. Then φ is sequence-valid in $\mathcal{M}$ iff the language generated by $\mathcal{M}$ (i.e., the language accepted by the weakly fair transition system $\mathcal{M}_\mathcal{A}$ for $\mathcal{M}$) is a subset of the language accepted by the Büchi-automaton $\mathcal{M}_\varphi$ for φ. That is,

$$\mathcal{M} \models \varphi \quad \text{iff} \quad L(\mathcal{M}_\mathcal{A}) \subseteq L(\mathcal{M}_\varphi).$$

The latter condition is equivalent to $L(\mathcal{M}_\mathcal{A}) \cap \overline{L(\mathcal{M}_\varphi)} = \emptyset$, or $L(\mathcal{M}_\mathcal{A} \times \mathcal{M}_{\neg\varphi}) = \emptyset$. Therefore, model checking of **LTL** sequence-validity in finite models reduces to the nonemptyness problem of Büchi-automata: a feasible way to check whether $\mathcal{M} \models \varphi$ is to construct the Büchi-automata $\mathcal{M}_\mathcal{A}$ for the model and $\mathcal{M}_{\neg\varphi}$ for $\neg\varphi$, and to check whether the language of the product automaton $\mathcal{M}_\mathcal{A} \times \mathcal{M}_{\neg\varphi}$ is empty.

If both system $\mathcal{M}$ and property φ are given as automata, then "specification" φ can be regarded as a "more abstract version" of the "implementation" $\mathcal{M}$. We write $\mathcal{M}_I \models \mathcal{M}_S$ if $L(\mathcal{M}_I) \subseteq L(\mathcal{M}_S)$, i.e., if (the language of) $\mathcal{M}_I$ is a subset of (the language of) $\mathcal{M}_S$. A *property* φ is defined to be just any ω-language $\varphi \subseteq \Sigma^\omega$, where $\Sigma = 2^\mathcal{P} \times \mathcal{R}$.

Fact 3.2 *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be Büchi-automata. Then*

1. *$\mathcal{M}_1 \models \mathcal{M}_2$ iff for all properties φ, if $\mathcal{M}_2 \models \varphi$ then $\mathcal{M}_1 \models \varphi$.*
2. *$\mathcal{M}_1 \models \mathcal{M}_2$ iff for all ω-regular φ, if $\mathcal{M}_2 \models \varphi$ then $\mathcal{M}_1 \models \varphi$.*

PROOF: One direction is immediate by transitivity of the subset relation: If $L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$ and $L(\mathcal{M}_2) \subseteq L(\varphi)$, then $L(\mathcal{M}_1) \subseteq L(\varphi)$. The other direction follows from instantiating φ with $L(\mathcal{M}_2)$ and, in the strong form, from the fact that the Büchi-automaton $\mathcal{M}_2$ defines a regular language. □

This fact can help to reduce the complexity of checking whether a model satisfies a formula. In order to prove $\mathcal{M}_1 \models \varphi$, it can be helpful to look for a "small" model $\mathcal{M}_2$ such that $\mathcal{M}_1 \models \mathcal{M}_2$ and $\mathcal{M}_2 \models \varphi$.

3.2 Safety and Liveness Properties

A similar characterization result holds for finite transition systems and a special class of ω-languages called *safety-properties*. For natural models $\mathcal{M}$ and $\mathcal{M}'$, let $\mathcal{M}^{[..i]}$ be the model consisting of the first i points of $\mathcal{M}$, and $\mathcal{M} \circ \mathcal{M}'$ be the concatenation of the two models $\mathcal{M}$ and $\mathcal{M}'$.

- φ is a *safety property*, iff for every natural model $\mathcal{M}$,

$$\mathcal{M} \models \varphi \;\text{ if }\; \forall i \exists \mathcal{M}' : \; \mathcal{M}^{[..i]} \circ \mathcal{M}' \models \varphi$$

This definition is from [AS85]. An ω-language φ is a safety property if for every model *not* satisfying φ there is a finite prefix $\mathcal{M}^{[..i]}$ which can not be completed by any continuation $\mathcal{M}'$ such that $\mathcal{M}^{[..i]} \circ \mathcal{M}' \models \varphi$. In other words, for every model dissatisfying φ something "bad" must have happened after some finite number of steps which cannot be remedied by any future good behavior. Hence, in Lamport's popular characterization, safety properties express that "something bad never happens" [Lam83].

- φ is a *liveness property*, iff for every natural model $\mathcal{M}$,

$$\forall i \exists \mathcal{M}' : \ \mathcal{M}^{[..i]} \circ \mathcal{M}' \models \varphi$$

A liveness property φ, on the other hand, can never be refuted by observing only a finite prefix of some run. It holds, if and only if every finite sequence can be completed to a model satisfying φ, hence φ states that "something good eventually happens". Notice, however, that in contrast to the "bad thing" referred to above, the occurrence of the "good thing" does not have to be observable in any fixed time interval. Thus, liveness failures cannot be detected by testing.

Without proof we state some facts about safety and liveness from [AS85]:

1. Safety properties are closed under finite unions and arbitrary intersections.

2. Liveness properties are closed under arbitrary unions, but not under intersections.

3. $\top$ is the only property which is both a safety and a liveness property.

4. For any property φ there exists a safety property φ_S and a liveness property φ_L such that $\varphi = (\varphi_S \cap \varphi_L)$.

The last of these facts is known as the *decomposition theorem* and can be proved by topological arguments. The safety-part of a property φ is the topological closure of φ, that is, the least safety property containing φ. As an example, on natural models the **LTL**-formula ($\mathtt{p}\ \mathbf{U}^+\ \mathtt{q}$) is equivalent to (($\mathtt{p}\ \mathbf{W}^+\ \mathtt{q}$)$\wedge\ \mathbf{F}^+\ \mathtt{q}$), where the language defined by ($\mathtt{p}\ \mathbf{W}^+\ \mathtt{q}$) is a safety property and the language defined by $\mathbf{F}^+\ \mathtt{q}$ is a liveness property. Similarly, total correctness statements about programs can be decomposed into invariance (safety) and termination (liveness).

In linear temporal logic, every formula built from literals with $\bot$, $\top$, $\wedge$, $\vee$ and $\mathbf{W}^+$ defines a safety property. In the proof, the only interesting case

is $(\varphi\ \mathbf{W}^+\ \psi)$. Assume that any model $\mathcal{M}$ falsifying both φ and ψ has a finite prefix $\mathcal{M}^{[..i]}$ such that any extension of $\mathcal{M}^{[..i]}$ falsifies these formulas. If $\mathcal{M} \not\models (\varphi\ \mathbf{W}^+\ \psi)$, then there is a $w_j > w_0$ such that $w_j \models (\neg\varphi \wedge \neg\psi)$, and $w_k \models \neg\psi$ for $w_0 < w_k < w_j$. Therefore, in any model $\mathcal{M}^{[..j+i]} \circ \mathcal{M}'$, the formula $(\varphi\ \mathbf{W}^+\ \psi)$ must be false. □

Another syntactical characterization of safety in linear temporal logic is with past-operators. Any **LTL** formula $\mathbf{G}^*\ \psi$, where ψ is a past formula, defines a safety property. Moreover, any **LTL**-definable safety property can be defined by a formula of this form [LPZ85].

Recall that a binary relation $\Delta \subseteq U \times U$ is called *image finite*, if for any $x \in U$ the set $\{y \in U \mid (x, y) \in \Delta\}$ is finite. In particular, any finite relation is image finite. We call a transition system (Σ, S, Δ, S_0) *finitary*, if S_0 is finite and Δ is image finite. Of course, any finite transition system is finitary. Intuitively, finitary transition systems allow only "finite nondeterminism".

Lemma 3.3 *Any finitary transition system defines a safety property.*

PROOF: Consider the language L of a finitary transition system. We have to show that for every sequence σ, if $\forall i \exists \sigma' :\ \sigma^{[..i]} \circ \sigma' \in L$ then $\sigma \in L$. In other words, assume that any finite prefix of σ can be extended to a string in L and show $\sigma \in L$. If σ is finite, then it is a finite prefix of itself; thus there exists some σ' such that $\sigma \circ \sigma' \in L$. Since every state of a transition system is accepting, it follows that $\sigma \in L$. If σ is infinite, consider the following computation tree: each node is marked by $(s, \sigma^{[..i]})$, where s is a state of the transition system and $\sigma^{[..i]}$ is a finite prefix of σ. The root is marked $(s, ())$, where s is any state. For any initial state $s_0 \in S_0$ of the transition system there is a child of the root in the computation tree which is marked (s_0, σ_0), where $\sigma_0 = \sigma^{[..0]}$ is the first letter of σ. Given a node marked $(s, \sigma^{[..i-1]})$ (where $i > 0$), for any s' such that $(s, \sigma_{i-1}, s') \in \Delta$ there is a child node in the tree marked $(s', (\sigma_0, ..., \sigma_i))$. Thus there exists a node marked $(s, \sigma^{[..i]})$ iff there is a path from some initial state to state s which is labelled by $(\sigma_0, ..., \sigma_{i-1})$. Since S_0 is finite and Δ is image finite, the computation tree is finitely branching. Since every prefix of σ can be extended to a string which is accepted by the transition system, the tree contains infinitely many nodes. Thus, by Königs lemma, it must contain an infinite branch. Therefore, there is a path in the transition system labelled by σ. Since all states in a transition system are recurring, it accepts σ. □

Without the finitary restriction, Lemma 3.3 does not hold: Consider the infinite transition system $\mathcal{M}$ of Figure 3.1. It shows a tree, such that for every natural number i a path of length i starts from the root. This transition system defines the set of all finite strings $(\mathbf{F}^*\mathbf{X}\ \bot)$, which is not a safety

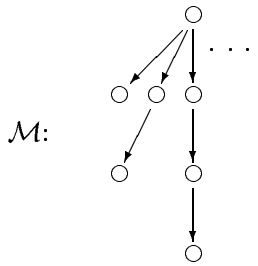


Figure 3.1: A non-finitary Kripke-model

property. Similarly, the same language can be defined by an image finite transition system with infinitely many starting states. However, Lemma 3.3 implies in particular, that any finite transition system defines an ω-regular safety property. A reverse statement also holds:

Lemma 3.4 *For every ω-regular safety property there is a finite transition system defining this property.*

PROOF: Assume that a Büchi-automaton defining a certain safety property φ is given. We transform this automaton into a suitable normal form. Any nonaccepting state can be deleted: Since safety properties are prefix-closed languages, if there is an accepted path which passes through nonaccepting states, then there must be an equivalent path passing only through accepting states. Similarly, *nonaccepting SCCs* can be deleted: These are nontrivial strongly connected components in the automaton which do not contain a recurring state. Since φ is a safety property, for any accepted path ρ passing through states in a nonaccepting SCC there must be an equivalent path which avoids this SCC. Otherwise, assume that $\rho = \rho_1 \circ \rho_2$, where ρ_1 leads into the nonaccepting SCC. Consider the (nonaccepted) path $\rho_1 \circ \sigma^\omega$ which passes infinitely often through the nodes of this nonaccepting SCC. Any finite prefix $\rho_1 \circ \sigma^n$ of this path can be extended to the accepted path $\rho_1 \circ \sigma^n \circ \rho_2$; hence the whole path would have to be accepted. After the deletion of nonaccepting SCCs, each nontrivial SCC contains a recurring state. Therefore, the automaton accepts all finite and infinite paths through its state graph. Consider the transition system with the same state set and transition relation, where all states are accepting and recurring.

The language of this transition system is the same as that of the (reduced) automaton. □

For **LTL** safety properties φ, a deterministic transition system $\mathcal{M}_\varphi$ corresponding to φ can be obtained directly by a tableau procedure; see chapter 4.2.

Given a finite Kripke model $\mathcal{M}$ and an ω-regular safety property φ, checking whether $\mathcal{M}$ sequence-validates φ is especially easy. Let $\mathcal{M}_\mathcal{A}$ be the weakly fair transition system corresponding to $\mathcal{M}$, and let $\mathcal{M}_\varphi$ be a deterministic finite transition system defining the same language as φ. As above, $\mathcal{M} \models \varphi$ iff $L(\mathcal{M}_\mathcal{A}) \subseteq L(\mathcal{M}_\varphi)$. Language containment can be decided by executing $\mathcal{M}_\mathcal{A}$ (program) and $\mathcal{M}_\varphi$ (specification) in parallel and checking that for every step in $\mathcal{M}_\mathcal{A}$ the corresponding step in $\mathcal{M}_\varphi$ exists. This approach is also used in *specification-based testing*, where a number of *test runs* $\sigma \in L(\mathcal{M}_\mathcal{A})$ is checked whether they conform to the specification, that is, $\sigma \in L(\mathcal{M}_\varphi)$. The test runs are either determined by the system under test, or selected by the specification according to some coverage strategy.

Safety properties can be used to characterize language containment for finitary transition systems just as as ω-regular properties for Büchi-automata (cf. Fact 3.2). For finitary transition systems, it is sufficient to check whether $\mathcal{M}_2 \models \varphi$ implies $\mathcal{M}_1 \models \varphi$ for all *safety* properties φ in order to establish $\mathcal{M}_1 \models \mathcal{M}_2$:

Theorem 3.5 *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be finitary transition systems. Then $\mathcal{M}_1 \models \mathcal{M}_2$ iff for all safety properties φ, if $\mathcal{M}_2 \models \varphi$ then $\mathcal{M}_1 \models \varphi$.*

PROOF: Assume that $\mathcal{M}_1 \models \mathcal{M}_2$, and that $\mathcal{M}_1 \not\models \varphi$. Then there exists a word σ accepted by $\mathcal{M}_1$ such that $\sigma \notin \varphi$. Since $L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$, this countermodel is also in the language of $\mathcal{M}_2$, hence $\mathcal{M}_2 \not\models \varphi$. For the other direction, since the set of all natural models generated from a finitary transition system is a safety property and by the fact that $\mathcal{M}_2 \models \mathcal{M}_2$ the assumption immediately reduces to $\mathcal{M}_1 \models \mathcal{M}_2$. □

3.3 Simulation Relations

The above characterization results concentrate on containment between the ω-languages generated by models and (linear time) formulas. However, there are two reasons to consider also weaker preorders between models: Firstly, for large nondeterministic transition systems language containment may not be easy to check. Secondly, sometimes it is desirable to formulate properties which depend on the *structure* of the system under consideration rather

than on its *behavior*. Such properties may not be preserved even for systems generating the same language. For example, reconsider the example from Section 1.2: The two models $\mathcal{M}_1$ and $\mathcal{M}_2$ of Figure 3.2 over $\mathcal{P} = \{\}$ and $\mathcal{R} = \{a, b, c\}$.

Figure 3.2: Two sequence-equivalent but branching-inequivalent Kripke-models

Clearly, $L(\mathcal{M}_1) = L(\mathcal{M}_2)$, and therefore $\mathcal{M}_1 \models \mathcal{M}_2$. That is, if we observe sequences of transitions, then every possible behavior of $\mathcal{M}_1$ is also a possible behavior of $\mathcal{M}_2$. However, if we observe not only transitions which *are* taken, but also transitions which *could be* taken, then the behavior of $\mathcal{M}_1$ and $\mathcal{M}_2$ differs: If "possible continuations" are indicated by small light bulbs, then in the first system after performing a both the b and c lights will be lit, whereas in the second system only one of both is on. Formally, for every **LTL**-formula ψ it holds that ψ is sequence-valid in $\mathcal{M}_1$ iff ψ is sequence-valid in $\mathcal{M}_2$. For $\varphi \triangleq [a]\,([b]\,\bot \vee [c]\,\bot)$, it holds that $\mathcal{M}_2 \models \varphi$, but $\mathcal{M}_1 \not\models \varphi$.

Given two models $\mathcal{M}_1 = (U_1, \mathcal{I}_1, w_1)$ and $\mathcal{M}_2 = (U_2, \mathcal{I}_2, w_2)$, we say that $\mathcal{M}_1$ is a *submodel* of $\mathcal{M}_2$ (denoted by $\mathcal{M}_1 \sqsubseteq \mathcal{M}_2$), if $U_1 \subseteq U_2$, $\mathcal{I}_1 = \mathcal{I}_2 \downarrow U_1$, and $w_1 = w_2$. Intuitively, a submodel consists of some parts of the original model. In the proof of Lemma 3.4 we constructed a special submodel which preserves all execution sequences. Generally, all temporal properties are preserved when a model is replaced by the *generated submodel*, i.e., the submodel consisting of all points reachable from the current point. However, usually properties are not preserved when a model is replaced by an arbitrary submodel. Instead of simply omitting parts of a model, it is better to collapse several points into a single point. Given two models $\mathcal{M}_1 = (U_1, \mathcal{I}_1, w_1)$ and $\mathcal{M}_2 = (U_2, \mathcal{I}_2, w_2)$, a relation $H \subseteq U_1 \times U_2$ is called a *simulation relation* between $\mathcal{M}_1$ and $\mathcal{M}_2$ if

- $(w_1, w_2) \in H$,
- For all $\mathtt{p} \in \mathcal{P}$, $u \in U_1$, and $v \in U_2$, if $(u, v) \in H$ then $u \in \mathcal{I}_1(\mathtt{p})$ iff

$v \in \mathcal{I}_2(\mathtt{p})$.

- For all u and v such that $(u, v) \in H$ and all R and u' such that $(u, u') \in \mathcal{I}_1(R)$ there is a v' with the property that $(v, v') \in \mathcal{I}_2(R)$ and $(u', v') \in H$.

Figure 3.3 illustrates the third condition.

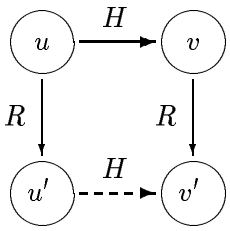


Figure 3.3: Simulation condition for u and v

We say that $\mathcal{M}_1$ *is simulated by* $\mathcal{M}_2$, or $\mathcal{M}_2$ *simulates* $\mathcal{M}_1$ (denoted by $\mathcal{M}_1 \lesssim \mathcal{M}_2$), if there exists a simulation relation H between $\mathcal{M}_1$ and $\mathcal{M}_2$. Simulation relates a model $\mathcal{M}_1$ to an *abstraction* $\mathcal{M}_2$ of the model $\mathcal{M}_1$. It guarantees that every behavior of the model is also a possible behavior of the abstraction. However, since a point in the abstract model usually represents a set of points in the original model, the abstraction might have behaviors that have no counterpart the original model. Thus, the term "simulation" is used as in "the PC simulates a gameboy" or "this program simulates the development of bacteria cultures".

Fact 3.6 $\lesssim$ *is a preorder on the class of all models.*

PROOF: The proof of reflexivity is immediate. For transitivity, note that the relational product of two simulation relations is again a simulation relation.□

If $\mathcal{M}_1 \sqsubseteq \mathcal{M}_2$, then $\mathcal{M}_1 \lesssim \mathcal{M}_2$. Moreover, if $\mathcal{M}_1 \lesssim \mathcal{M}_2$, then $\mathcal{M}_1 \models \mathcal{M}_2$: If $\mathcal{M}_2$ can simulate $\mathcal{M}_1$, then for every maximal run σ generated by $\mathcal{M}_1$ there exists a corresponding $\sigma' \in \mathcal{M}_2$. A model is *deterministic*, if for every $w \in U$ and $R \in \mathcal{R}$ there is at most one $w' \in U$ such that $(w, w') \in \mathcal{I}(R)$. For deterministic $\mathcal{M}_2$ also the converse holds: $\mathcal{M}_1 \models \mathcal{M}_2$ iff $\mathcal{M}_1 \lesssim \mathcal{M}_2$. This is true because for any word there is at most one path through a deterministic transition system. Deterministic models and properties are an important

special case. Whereas for many problems in nondeterministic transition systems an exponential search via backtracking is used, in the deterministic case the same problems can be solved with polynomial complexity.

Lemma 3.7 *Let H be a simulation relation between $\mathcal{M}_1 = (U_1, \mathcal{I}_1, w_1)$ and $\mathcal{M}_2 = (U_2, \mathcal{I}_2, w_2)$, and $(w_1', w_2') \in H$. Then $(U_1, \mathcal{I}_1, w_1') \lesssim (U_2, \mathcal{I}_2, w_2')$.*

PROOF: The proof is immediate from the definition of simulation relations.□

A modal box formula is a formula not involving any diamond operator. More precisely, literals (propositions and negated propositions) and $\bot, \top$ are modal box formulas, and if φ and ψ are modal box formulas, then $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ and $[R]\,\varphi$ are modal box formulas. Similar to Lemmas 3.2 and 3.4, the following lemma relates simulations between models and preservation of modal box formulas:

Lemma 3.8 *Let $\mathcal{M}_1 = (U_1, \mathcal{I}_1, w_1)$ and $\mathcal{M}_2 = (U_2, \mathcal{I}_2, w_2)$ be Kripke-models. Then $\mathcal{M}_1 \lesssim \mathcal{M}_2$ implies that for all modal box properties φ, if $\mathcal{M}_2 \models \varphi$ then $\mathcal{M}_1 \models \varphi$.*

PROOF: The proof is by induction on φ. The base cases $\bot, \top$ are trivial. For $\mathtt{p} \in \mathcal{P}$, the assumption $(w_1, w_2) \in H$ implies $w_1 \in \mathcal{I}_1(\mathtt{p})$ iff $w_2 \in \mathcal{I}_2(\mathtt{p})$. For boolean operators $\wedge$, $\vee$, the statement is an immediate consequence of the induction hypothesis. Finally, if $w_1 \not\models [R]\,\varphi$, then there is a $w_1' \in U_1$ such that $(w_1, w_1') \in \mathcal{I}_1(R)$ and $w_1' \not\models \varphi$. Since $\mathcal{M}_1 \lesssim \mathcal{M}_2$, there is a $w_2' \in U_2$ such that $(w_2, w_2') \in \mathcal{I}_2(R)$ and $(w_1', w_2') \in H$. Lemma 3.7 asserts that $(U_1, \mathcal{I}_1, w_1') \lesssim (U_2, \mathcal{I}_2, w_2')$. According to the induction hypothesis, $w_2' \not\models \varphi$. Therefore, $w_2 \not\models [R]\,\varphi$, which was to be proved. □

This lemma makes it possible to check safety in the abstracted (small) model $\mathcal{M}_2$ rather than in the original (large) model $\mathcal{M}_1$: If $\mathcal{M}_1$ violates a modal safety property, then this violation will also occur in $\mathcal{M}_2$.

The above statement can be extended to more expressive logics. The logic **ACTL** [Lon93, CGL93, CLM89, DGG94] is "**CTL** without **E** quantifier". That is, literals and $\top, \bot$ are **ACTL** formulas, and if φ and ψ are **ACTL** formulas, then $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $\mathbf{A}(\varphi\ \mathbf{U}^+\ \psi)$ and $\mathbf{A}(\varphi\ \mathbf{W}^+\ \psi)$ are **ACTL** formulas, where $\mathbf{A}(\varphi\ \mathbf{W}^+\ \psi) \triangleq \neg\mathbf{E}(\neg\psi\ \mathbf{U}^+\ \neg(\varphi \vee \psi))$.

Theorem 3.9 *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be Kripke-models and φ be an* **ACTL** *formula. If $\mathcal{M}_1 \lesssim \mathcal{M}_2$ and $\mathcal{M}_2 \models \varphi$, then $\mathcal{M}_1 \models \varphi$.*

PROOF: Intuitively, this theorem is true because formulas in **ACTL** describe properties that are true of all paths of a model. They cannot express

the existence of a specific path in the model. If $\mathcal{M}_1 \lesssim \mathcal{M}_2$, then every behavior of $\mathcal{M}_1$ is a behavior of $\mathcal{M}_2$. Thus every formula of **ACTL** that is true in $\mathcal{M}_2$ must also be true in $\mathcal{M}_1$. Formally, if $w_1 \not\models \mathbf{A}(\varphi\ \mathbf{U}^+\ \psi)$, then there is in $\mathcal{M}_1$ either a finite sequence of nodes w_1', w_1'', ..., $w_1^{(n)}$, such that $w_1^{(i)} \not\models \psi$ for $0 < i < n$, and $w_1^{(n)} \not\models (\varphi \vee \psi)$, or a maximal path w_1', w_1'', ..., such that $w_1^{(i)} \not\models \psi$ for all $i > 0$. Similar to the above, the induction hypothesis proves that a corresponding finite or infinite sequence w_2', w_2'', ..., $w_2^{(n)}$ or w_2', w_2'', ..., exists, such that $w_2^{(i)} \not\models \psi$ for $0 < i < n$, and $w_2^{(n)} \not\models (\varphi \vee \psi)$, or $w_2^{(i)} \not\models \psi$ for all $i > 0$. Thus $w_2 \not\models \mathbf{A}(\varphi\ \mathbf{U}^+\ \psi)$. □

In general the converse of the above lemma and theorem are not valid. Essentially, this is due to the same reason why Lemma 3.3 fails to hold for non-finitary transition system. Consider the counterexample of Figure 3.4.

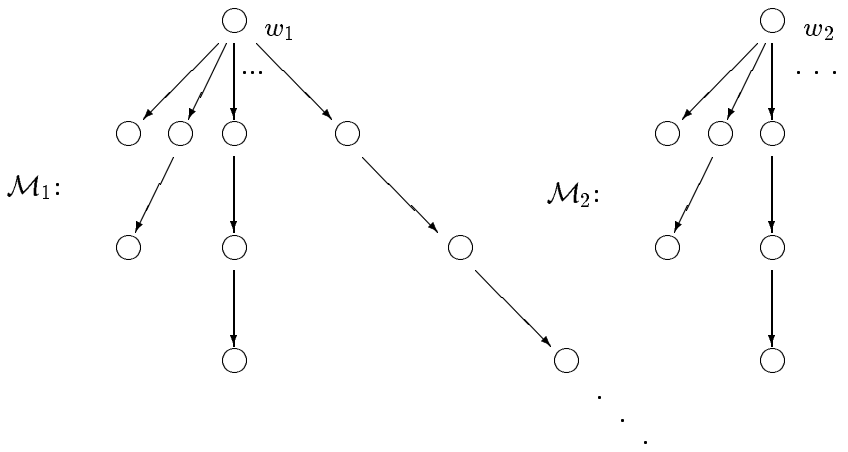


Figure 3.4: Two modally indistinguishable models

Both models have infinitely many branches from the root, one branch of length one, one branch of length two, one branch of length three, and so on. $\mathcal{M}_1$ has an additional branch of infinite length. These two models cannot be distinguished by any modal formula:

Lemma 3.10 *For any $\varphi \in$ **ML** it holds that $\mathcal{M}_1 \models \varphi$ iff $\mathcal{M}_2 \models \varphi$*

PROOF: The statement is proved by induction on φ. The only interesting case is $\varphi = \langle R \rangle\ \psi$, $\mathcal{M}_1 \models \varphi$, and the successor w_1' of w_1 for which $w_1' \models \psi$

is on the additional infinite branch of $\mathcal{M}_1$. Choose any branch of $\mathcal{M}_2$ of length at least n, where n is the number of modal operators in φ. Denote the i-th point on the infinite branch of $\mathcal{M}_1$ and on the chosen branch of $\mathcal{M}_2$ by $w_1^{(i)}$ and $w_2^{(i)}$, respectively (where $w_1^{(0)} = w_1$ and $w_2^{(0)} = w_2$). Then for all $i \leq n$ and all sub-formulas ξ_i of φ with at most $(n-i)$ modal operators it holds that $w_1^{(i)} \models \xi_i$ iff $w_2^{(i)} \models \xi_i$. This is proved by subinduction on $n-i$: If $n-i=0$, then it holds by definition of the models. If $n-i>0$ and $w_1^{(i+1)} \models \xi_{i+1}$ iff $w_2^{(i+1)} \models \xi_{i+1}$, then $w_1^{(i)} \models \langle R \rangle\ \xi_{i+1}$ iff $w_2^{(i)} \models \langle R \rangle\ \xi_{i+1}$. Especially, since φ has n modal operators, $w_1^{(0)} \models \varphi$ iff $w_2^{(0)} \models \varphi$. □

In particular, Lemma 3.10 implies that for every modal box formula φ, if $\mathcal{M}_2 \models \varphi$ then $\mathcal{M}_1 \models \varphi$. Yet, $\mathcal{M}_2$ does not simulate $\mathcal{M}_1$: Assume a simulation relation H mapping the first node w of the infinite path of $\mathcal{M}_1$ to any node w' of any finite path in $\mathcal{M}_2$. Then H must map the successor of w to the successor of w', the successor of the successor of w to the successor of the successor of w', and so on. There are finitely many successors from w', but infinitely many successors from w. Thus, after a finite number of steps, there will be nodes $u \in \mathcal{M}_1$ and $v \in \mathcal{M}_2$ such that $(u,v) \in H$, and u has a successor in $\mathcal{M}_1$, but v has no successor in $\mathcal{M}_2$.

This is a somewhat contrived counterexample. In "many" cases, the converse will hold. Recall that a model is called *image finite*, if every point has only finitely many successors.

Theorem 3.11 *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be image finite Kripke-models. Then $\mathcal{M}_1 \lesssim \mathcal{M}_2$ iff for all modal box formulas φ, if $\mathcal{M}_2 \models \varphi$ then $\mathcal{M}_1 \models \varphi$.*

PROOF: Assume that all modal box formulas holding in $\mathcal{M}_2$ are also valid for $\mathcal{M}_1$, and construct a simulation between $\mathcal{M}_1$ and $\mathcal{M}_2$. Define H by $(u,v) \in H$ iff for all modal box formulas φ, if $v \models \varphi$ then $u \models \varphi$. Then $(w_1, w_2) \in H$ by definition, and $(u,v) \in H$ implies $\mathcal{L}_1(u) = \mathcal{L}_2(v)$ since literals are modal box formulas. Assume $(u,v) \in H$ and $(u,u') \in \mathcal{I}_1(R)$. We have to show that there is a v' such that $(v,v') \in \mathcal{I}_2(R)$ and for all modal box formulas φ, if $u' \not\models \varphi$ then $v' \not\models \varphi$. Assume for contradiction that for each v' with $(v,v') \in \mathcal{I}_2(R)$ there is a $\varphi_{v'}$ such that $u' \not\models \varphi_{v'}$ and $v' \models \varphi_{v'}$. Since $\mathcal{M}_2$ is image finite, $\bigvee \varphi_{v'}$ exists and is a modal box formula. Moreover, for all such v', we have $v' \models \bigvee \varphi_{v'}$, which means $v \models [R] \bigvee \varphi_{v'}$. This implies $u \models [R] \bigvee \varphi_{v'}$ and therefore $u' \models \bigvee \varphi_{v'}$, a contradiction to the assumption that $u' \not\models \varphi_{v'}$ for some $\varphi_{v'}$. □

We already mentioned that the above theorems can be used to reduce the complexity of model checking. To prove that $\mathcal{M}_1 \models \varphi$, it can help to find an appropriate abstraction $\mathcal{M}_2$, and to prove $\mathcal{M}_1 \lesssim \mathcal{M}_2$ and $\mathcal{M}_2 \models \varphi$.

Extremely efficient algorithms are known to check language inclusion for deterministic finite automata [HU79]. These algorithms can be used to check the simulation preorder for deterministic models. For nondeterministic finite models $\mathcal{M}_1 = (U_1, \mathcal{I}_1, w_1)$ and $\mathcal{M}_2 = (U_2, \mathcal{I}_2, w_2)$, to check whether $\mathcal{M}_1 \lesssim \mathcal{M}_2$ we define a sequence of relations H^0, H^1, ...on $U_1 \times U_2$ as follows:

1. $(u, v) \in H^0$ iff for all $\mathtt{p} \in \mathcal{P}$ it holds that $u \in \mathcal{I}_1(\mathtt{p})$ iff $v \in \mathcal{I}_2(\mathtt{p})$
2. $(u, v) \in H^{n+1}$ iff $(u, v) \in H^n$ and for all R and $u' \in U_1$ such that $(u, u') \in \mathcal{I}_1(R)$ there is a v' with the property that $(v, v') \in \mathcal{I}_2(R)$ and $(u', v') \in H^n$.

The intersection H^* of all H^n is the largest simulation relation between $\mathcal{M}_1$ and $\mathcal{M}_2$. That is, $\mathcal{M}_1 \lesssim \mathcal{M}_2$ iff $(w_1, w_2) \in H^*$. Algorithmically, if $H^n = H^{n-1}$, then $H^* \triangleq H^n$ and the construction terminates. Since the structures are finite, there are only finitely many different H^n. Thus, termination is guaranteed. In Figure 3.3, $R(u)$ denotes the set $\{u' \mid (u, u') \in \mathcal{I}(R)\}$, and

start: $H^{new} := \{(u, v) \mid \mathcal{L}_1(u) = \mathcal{L}_2(v)\}$
repeat
  $H^{old} := H^{new}$; $H^{new} := \emptyset$
  for all $(u, v) \in H^{old}$ **do**
    $add := \top$; **for all** $R \in \mathcal{R}$ **do**
      if not $R(u) \subseteq ((R(u) \times R(v)) \cap H^{old})|_1$ **then** $add := \bot$
    if add **then** $H^{new} := H^{new} \cup \{(u, v)\}$
until $H^{new} = H^{old}$

Figure 3.5: Algorithm for simulation checking

$|_1$ is the first component of a tuple. In the next chapter, a more elaborate implementation of a similar algorithm for symmetric simulation relations is given, which is based on partition refinement.

In this chapter, we consider symmetric preorders, i.e., equivalences, and equivalence transformations between models. There are various possibilities for defining equivalences on models. For any preorder from the preceding chapter, an equivalence can be defined by $\mathcal{M}_1 \simeq \mathcal{M}_2$ iff $\mathcal{M}_1 \preceq \mathcal{M}_2$ and $\mathcal{M}_2 \preceq \mathcal{M}_1$. In this way, the equivalence induced by the submodel ordering $\sqsubseteq$ is isomorphism. For $\mathcal{M}_1 \models \mathcal{M}_2$, the symmetric version is equality

of the generated languages. Other model equivalences are introduced by equivalence with respect to logical formulas, and by symmetric simulations.

3.4 Bisimulations (p-morphisms)

A classical notion from modal logic is *p-morphism*[Seg68], [Seg71, p37] or *bisimulation*[Hen80, Par81]. A bisimulation is a relation $\sim$ between the universes of two Kripke-models $(U_1, \mathcal{I}_1, w_1)$ and $(U_2, \mathcal{I}_2, w_2)$ such that

1. $w_1 \sim w_2$,
2. If $u \sim v$, then $u \in \mathcal{I}_1(\mathtt{p})$ iff $v \in \mathcal{I}_2(\mathtt{p})$
3. If $u \sim v$ and $(u, u') \in \mathcal{I}_1(R)$, then there exists v' such that $(v, v') \in \mathcal{I}_2(R)$ and $u' \sim v'$.
4. If $u \sim v$ and $(v, v') \in \mathcal{I}_2(R)$, then there exists u' such that $(u, u') \in \mathcal{I}_1(R)$ and $u' \sim v'$.

Two Kripke-models $\mathcal{M}_1$ and $\mathcal{M}_2$ are *bisimilar* (denoted by $\mathcal{M}_1 \sim \mathcal{M}_2$), if there exists a bisimulation between them. Figure 3.6 shows some examples for bisimilar models.

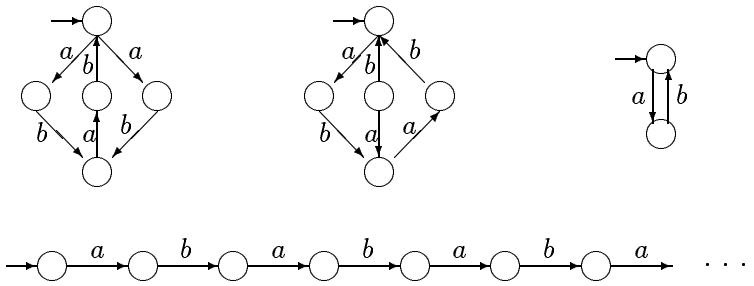


Figure 3.6: Bisimilar models

Observations:

- Each model is bisimilar to one where duplicate states (which have the same input and output) are removed,
- Each model is bisimilar to its unfolding, and
- Each model is bisimilar to its reachable part.

If $\mathcal{M}_1 \sim \mathcal{M}_2$, then $\mathcal{M}_1 \lesssim \mathcal{M}_2$ and $\mathcal{M}_2 \lesssim \mathcal{M}_1$; the other direction of this statement is not necessarily true. For example, each of the models in Figure 3.7 simulates the other one, but they are not bisimilar.

Figure 3.7: Not-bisimilar models

Bisimulation relations are precisely those equivalences which preserve modal formulas:

Lemma 3.12 *Bisimilar models are modally equivalent. In other words, for all Kripke-models $\mathcal{M}_1 \sim \mathcal{M}_2$ and all multimodal formulas φ, we have $\mathcal{M}_1 \models \varphi$ iff $\mathcal{M}_2 \models \varphi$.*

The proof is by induction on the structure of φ, analogous to the proof of Lemma 3.8.

Hence, it is "safe" to substitute a model by a bisimilar one in a structured software development process: All multimodal formulas which are valid for the original model will remain valid for the substituted model. The converse of this theorem again requires image finiteness:

Theorem 3.13 (Segerberg71) *Image finite models are modally equivalent iff they are bisimilar.*

Again, the proof is similar to the proof of Theorem 3.11 in the previous chapter. The only difference is that bisimulation is a symmetric relation.

In general, this theorem does not hold for more expressive logics. For *finite* Kripke-models, however, it can be lifted even to logics like monotonic μ**TL**. Given any formula φ which is syntactically monotonic in q, and a natural number n, we define $\nu^0 q\ \varphi \triangleq \top$, and $\nu^{n+1} q\ \varphi \triangleq \varphi\{q := \nu^n q\ \varphi\}$. That is, $\nu^n q\ \varphi \triangleq \varphi\{q := \varphi\}\{q := \varphi\} \cdots \{q := \top\}$.

Lemma 3.14 *Let $\mathcal{M} \triangleq (U, \mathcal{I}, w)$ be a finite model, where $|U| = n$. Then $\mathcal{M} \models \nu q\ \varphi$ iff $\mathcal{M} \models \nu^n q\ \varphi$*

PROOF: One direction of this lemma follows from the fact that $\nu q\ \varphi$ denotes a fixed point, i.e., $(\nu q\ \varphi \rightarrow \varphi\{q := \nu q\ \varphi\})$. Since φ is monotonic, this

implies $(\varphi\{q := \nu q\ \varphi\} \rightarrow \varphi\{q := \varphi\{q := \nu q\ \varphi\}\})$. By chain reasoning, $(\nu q\ \varphi \rightarrow \varphi\{q := \varphi\{q := \nu q\ \varphi\}\})$. By induction, $(\nu q\ \varphi \rightarrow \varphi\{q := \varphi\}\{q := \varphi\}...\{q := \nu q\ \varphi\})$. Again, since φ is monotonic in q, it holds that $(\varphi\{q := \nu q\ \varphi\} \rightarrow \varphi\{q := \top\})$, thus $(\nu q\ \varphi \rightarrow \nu^n q\ \varphi)$ is valid. For the other direction, consider the sequence $((\nu^n q\ \varphi)^{\mathcal{M}})_{n\geq 0}$ of sets of points. Clearly, $(\nu^0 q\ \varphi)^{\mathcal{M}} = \top^{\mathcal{M}} = U \supseteq (\nu^1 q\ \varphi)^{\mathcal{M}}$. Since $\varphi_q^{\mathcal{M}}$ is monotonic (cf. Fact 2.9), $(\nu^1 q\ \varphi)^{\mathcal{M}} = \varphi_q^{\mathcal{M}}((\nu^0 q\ \varphi)^{\mathcal{M}}) \supseteq \varphi_q^{\mathcal{M}}((\nu^1 q\ \varphi)^{\mathcal{M}}) = (\nu^2 q\ \varphi)^{\mathcal{M}}$. Continuing this argument, we conclude that $((\nu^n q\ \varphi)^{\mathcal{M}})_{n\geq 0}$ is a descending chain of sets. There are two possibilities: either there exists an $i < |U|$ such that $(\nu^i q\ \varphi)^{\mathcal{M}} = (\nu^{i+1} q\ \varphi)^{\mathcal{M}}$, hence $(\nu^i q\ \varphi)^{\mathcal{M}} = (\nu^n q\ \varphi)^{\mathcal{M}}$, or $(\nu^n q\ \varphi)^{\mathcal{M}} = \{\ \}$. In either case, the sequence stabilizes after at most $|U|$ steps: $(\nu^n q\ \varphi)^{\mathcal{M}} = (\nu^{n+1} q\ \varphi)^{\mathcal{M}}$. As a consequence, $(\nu^n q\ \varphi \rightarrow \nu^{n+1} q\ \varphi)$ is universally valid in $(U, \mathcal{I})$. Now assume that $\mathcal{M} \not\models \nu q\ \varphi$, and show that $\mathcal{M} \not\models \nu^n q\ \varphi$. According to the definition, $(U, \mathcal{I}, w) \not\models \nu q\ \varphi$ means that for all $Q \subseteq U$ such that $w \in Q$ there exists a $v \in Q$ such that $(U, \mathcal{I}', v) \not\models \varphi$, where $\mathcal{I}'(q) = Q$. Let $Q = (\nu^n q\ \varphi)^{\mathcal{M}}$. If $w \notin Q$, then $(U, \mathcal{I}, w) \not\models \nu^n q\ \varphi$ and we are done. If $w \in Q$, then for some v it holds that $(U, \mathcal{I}, v) \models \nu^n q\ \varphi$, and $(U, \mathcal{I}', v) \not\models \varphi$, where $\mathcal{I}'(q) = (\nu^n q\ \varphi)^{\mathcal{M}}$. In other words, $(U, \mathcal{I}', v) \not\models \varphi\{q := \nu^n q\ \varphi\}$, which means that $(U, \mathcal{I}, v) \not\models \nu^{n+1}\varphi$. Since $(U, \mathcal{I}, v) \models (\nu^n q\ \varphi \rightarrow \nu^{n+1} q\ \varphi)$, this is a contradiction. □

This lemma is important for model checking of μ**TL** on finite Kripke models. Moreover, it allows to prove the following result.

Theorem 3.15 *Finite models are monotonic μ**TL**-equivalent iff they are bisimilar.*

PROOF: Finite and monotonic μ**TL**-equivalent models are also image finite and modal equivalent, since μ**TL**-equivalence implies modal equivalence. Hence as an immediate consequence of Theorem 3.13 such models are bisimilar. For the other direction, assume that $\mathcal{M}_1 \models \varphi$ and $\mathcal{M}_2 \not\models \varphi$, where φ is a monotonic μ**TL**-formula. As a consequence of Lemma 3.14, $\mathcal{M}_1 \models \varphi^n$ and $\mathcal{M}_2 \not\models \varphi^n$, where $n \triangleq \max(|U_1|, |U_2|)$ and φ^n is φ where every sub-formula $\nu q\ \psi$ is replaced by $\nu^n q\ \psi$. Since φ^n is a multimodal formula, $\mathcal{M}_1$ and $\mathcal{M}_2$ are modally inequivalent and therefore not bisimilar. □

Corollary 3.16 *Any two finite Kripke-models which can be distinguished by a monotonic μ**TL**-formula can also be distinguished by a multimodal formula.*

[BCG88] proved that if two finite models can be distinguished by a formula of the logic **CTL***, then they can be distinguished by a **CTL** formula. Since every **CTL*** formula has a monotonic μ**TL** equivalent, this result can be obtained as a consequence of the above.

3.5 Bisimulation Minimization

In this section we show how to minimize a given Kripke-model with respect to bisimulation equivalence. Note that our definitions did not exclude bisimulations from a model to itself (*auto-bisimulations*); i.e., some points in a model can be bisimilar to other points in the *same* model.

Lemma 3.17 *The union of any number of auto-bisimulations on a model is again an auto-bisimulation.*

Thus, for any model, there exists a largest auto-bisimulation, namely, the union of all auto-bisimulations of this model. Additionally, the reflexive transitive symmetric closure of any auto-bisimulation is again an auto-bisimulation. Hence, for any auto-bisimulation $\sim$ there is a largest equivalence relation $\equiv$ containing it ($\sim \subseteq \equiv$) which is again an auto-bisimulation. And, the largest auto-bisimulation must be an equivalence relation on the set of points of a model.

Given any model $\mathcal{M} \triangleq (U, \mathcal{I}, w_0)$, and any equivalence relation $\equiv$ on U. Define the *quotient of $\mathcal{M}$ with respect to $\equiv$* to be the model $\mathcal{M}^{\equiv} \triangleq (U^{\equiv}, \mathcal{I}^{\equiv}, w_0^{\equiv})$, where $U^{\equiv}$ is the set of equivalence classes of U with respect to $\equiv$, $w_0^{\equiv}$ is the equivalence class of w_0, $w^{\equiv} \in \mathcal{I}^{\equiv}(\mathtt{p})$ if $w \in \mathcal{I}(\mathtt{p})$, and $(w_1^{\equiv}, w_2^{\equiv}) \in \mathcal{I}^{\equiv}(R)$ if $(w_1, w_2) \in \mathcal{I}(R)$.

Lemma 3.18 *If the equivalence relation $\equiv$ is an auto-bisimulation, then $\mathcal{M} \sim \mathcal{M}^{\equiv}$.*

PROOF: Define $u \sim v^{\equiv}$ iff $u \equiv v$. That is, each point in the original model is mapped to its equivalence class in the quotient model. We have to show that for this relation the four conditions defining a bisimulation (cf. page76) hold. For the initial point, $w_0 \sim w_0^{\equiv}$ holds because $w_0 \equiv w_0$. Since $\equiv$ is a bisimulation, $u \equiv v$ implies that $\mathcal{L}(u) = \mathcal{L}(v)$. Thus if $u \sim v^{\equiv}$ then $u \in \mathcal{I}(\mathtt{p})$ iff $v^{\equiv} \in \mathcal{I}^{\equiv}(\mathtt{p})$. Furthermore, if $(u_1, u_2) \in \mathcal{I}(R)$ and $u_1 \sim v_1^{\equiv}$), then by definition $(u_1^{\equiv}, u_2^{\equiv}) \in \mathcal{I}^{\equiv}(R)$ and $u_1 \equiv v_1$. Therefore, $u_1^{\equiv} = v_1^{\equiv}$, i.e., $(v_1^{\equiv}, u_2^{\equiv}) \in \mathcal{I}^{\equiv}(R)$. For the last condition, assume that $(v_1^{\equiv}, v_2^{\equiv}) \in \mathcal{I}^{\equiv}(R)$ and $v_1^{\equiv} \sim u_1$. Then there exist w_1 and w_2 such that $w_1 \equiv v_1$, $w_2 \equiv v_2$ and $(w_1, w_2) \in \mathcal{I}(R)$. From $v_1^{\equiv} \sim u_1$ we infer $u_1 \equiv v_1$ and thus $u_1 \equiv w_1$. Since $\equiv$ is a bisimulation, there exists a $u_2 \equiv w_2$ such that $(u_1, u_2) \in \mathcal{I}(R)$. From $u_2 \equiv w_2$ and $w_2 \equiv v_2$ we conclude that $u_2 \equiv v_2$, i.e., $u_2 \sim v_2^{\equiv}$. □

The quotient of a model with respect to its largest auto-bisimulation can be regarded as a minimal representation of this model. In finite models, this minimal representation can be constructed very efficiently.

For any set of points $P \subseteq U$, let $\langle R \rangle\ P \triangleq \{w \mid \exists w' \in P, (w, w') \in \mathcal{I}(R)\}$. Given any partition of U into equivalence classes, call a component $w^{\equiv}$ *uniform*, if for all $\mathtt{p} \in \mathcal{P}$ it holds that $w^{\equiv} \subseteq \mathcal{I}(\mathtt{p})$ or $w^{\equiv} \cap \mathcal{I}(\mathtt{p}) = \emptyset$. That is, $w^{\equiv}$ is uniform if $\mathcal{L}(w_1) = \mathcal{L}(w_2)$ for all $w_1, w_2 \in w^{\equiv}$. A component $w^{\equiv}$ is called *stable with respect to* P, if for all R either $w^{\equiv} \subseteq \langle R \rangle\ P$ or $w^{\equiv} \cap \langle R \rangle\ P = \emptyset$. The partition is called *stable*, if all components are uniform and stable with respect to all components.

Theorem 3.19 *The coarsest stable partition is the largest auto-bisimulation.*

PROOF: First, we show that any stable partition is an auto-bisimulation. Trivially, $w_0 \equiv w_0$. Since $u^{\equiv}$ is uniform, $u \equiv v$ implies $\mathcal{L}(u) = \mathcal{L}(v)$. If $(u, u') \in \mathcal{I}(R)$, then $u^{\equiv} \subseteq \langle R \rangle\ u'^{\equiv}$, because $u^{\equiv}$ is stable with respect to $u'^{\equiv}$. In other words, $u^{\equiv} \subseteq \{w \mid \exists w' \equiv u',\ (w, w') \in \mathcal{I}(R)\}$. Therefore, if $u \equiv v$, then there is a $v' \equiv u'$ such that $(v, v') \in \mathcal{I}(R)$. The symmetric condition is proved symmetrically. Vice versa, every auto-bisimulation defines a stable partition: To show that $u^{\equiv}$ is stable with respect to $v^{\equiv}$, assume that $u_1 \equiv u_2 \in u^{\equiv}$. Since $\equiv$ is a bisimulation, for every $(u_1, u_1') \in \mathcal{I}(R)$ and $u_1' \in v^{\equiv}$ there must be a $u_2' \equiv u_1' \in v^{\equiv}$ such that $(u_2, u_2') \in \mathcal{I}(R)$. Therefore, $u^{\equiv} \subseteq \langle R \rangle\ v^{\equiv}$ or $u^{\equiv} \cap \langle R \rangle\ v^{\equiv} = \emptyset$. If $\equiv$ is the coarsest stable partition, then for any auto-bisimulation $\sim$ it holds that $\sim \subseteq \equiv$. Assuming for contradiction that u, v and $\sim$ exist such that $u \sim v$ and not $u \equiv v$, according to Lemma 3.17 the union of $\sim$ and $\equiv$ would be a stable partition coarser than $\equiv$. □

The following algorithm can be used to construct the coarsest stable partition:

— Start with the trivial partition consisting of only one component

— Repeat

  – Choose a component $w_0^{\equiv}$ and a proposition $\mathtt{p} \in \mathcal{P}$;
  – Split $w_0^{\equiv}$ into $w_0^{\equiv} \cap \mathcal{I}(\mathtt{p})$ and $w_0^{\equiv} \setminus \mathcal{I}(\mathtt{p})$

  or

  – Choose components $w_0^{\equiv}$ and $w_1^{\equiv}$, and a relation $R \in \mathcal{R}$;
  – Split $w_0^{\equiv}$ into $w_0^{\equiv} \cap \langle R \rangle\ w_1^{\equiv}$ and $w_0^{\equiv} \setminus \langle R \rangle\ w_1^{\equiv}$

  until no new components can be obtained that way

The Paige-Tarjan algorithm [PT87] given in Figure 3.8 below is a sophisticated implementation of this idea; it maintains two partitions: a coarser

one, C, and a finer one, F. All components in F are stable with respect to any component in C. The nondeterministic choice in the above *repeat*-loop is replaced by a systematic split of the finer partition with respect to all components of the coarser partition. Initially, C is the trivial partition and F is the split of C w.r.t. all $\mathtt{p} \in \mathcal{P}$ and $R \in \mathcal{R}$. Then, a $w^{\equiv} \in C$ is split into $w_1^{\equiv} \in F$ and $w_2^{\equiv} \triangleq w^{\equiv} \setminus w_1^{\equiv}$. Any $w_0^{\equiv} \in F$ is split into four parts: First, it is split with respect to $\langle R \rangle\ w_1$, and then again with respect to $\langle R \rangle\ w_2$.

start: $C := \{\{U\}\}$, $F := \{\{U\}\}$
  for all $\mathtt{p} \in \mathcal{P}$ and $w^{\equiv} \in F$ **do**
    $F := (F \setminus \{w^{\equiv}\}) \cup \{w^{\equiv} \cap \mathcal{I}(\mathtt{p}),\ w^{\equiv} \setminus \mathcal{I}(\mathtt{p})\}$
  for all $R \in \mathcal{R}$ and $w^{\equiv} \in F$ **do**
    $F := (F \setminus \{w^{\equiv}\}) \cup \{w^{\equiv} \cap \langle R \rangle\ \{U\},\ w^{\equiv} \setminus \langle R \rangle\ \{U\}\}$
while $C \neq F$ **do**
  choose $w^{\equiv} \in C \setminus F$ and $w_1^{\equiv} \in F$ such that $w_1^{\equiv} \subseteq w^{\equiv}$
  $w_2^{\equiv} := w^{\equiv} \setminus w_1^{\equiv}$; $\quad C := (C \setminus \{w\}) \cup \{w_1^{\equiv},\ w_2^{\equiv}\}$
  for all $R \in \mathcal{R}$ and $w_0^{\equiv} \in F$ **do**
    $F := F \setminus \{w_0^{\equiv}\} \cup$
      $\{(w_0^{\equiv} \cap \langle R \rangle\ w_1^{\equiv}) \cap \langle R \rangle\ w_2^{\equiv},\ (w_0^{\equiv} \cap \langle R \rangle\ w_1^{\equiv}) \setminus \langle R \rangle\ w_2^{\equiv},$
       $(w_0^{\equiv} \setminus \langle R \rangle\ w_1^{\equiv}) \cap \langle R \rangle\ w_2^{\equiv},\ (w_0^{\equiv} \setminus \langle R \rangle\ w_1^{\equiv}) \setminus \langle R \rangle\ w_2^{\equiv}\ \}$
end

Figure 3.8: Page-Tarjan algorithm for bisimulation minimization

In the last split, since $w_0^{\equiv}$ is stable with respect to C, either $w_0^{\equiv} \subseteq \langle R \rangle\ w$ or $w_0^{\equiv} \cap \langle R \rangle\ w = \emptyset$ for all R. Moreover, $w = w_1 \cup w_2$, thus $\langle R \rangle\ w = \langle R \rangle\ w_1 \cup \langle R \rangle\ w_2$. Therefore, either the last or the first three parts in the above split of $w_0^{\equiv}$ are empty. This algorithm has complexity $O(m \cdot \log n)$, where n is the number of points in the original model, and m is the number of points (partitions) in the result.

3.6 Conformance and Mirroring

A different approach to the validation of safety-critical systems is rooted in the process algebraic tradition. Dill [Dil89b] used *conformance* of an implementation with respect to a specification as a correctness criterion for reactive systems. In this approach both specification and implementation

are given in a modelling language, e.g., as process algebraic expression, state transition graphs or elementary net. Since such formalisms are familiar in engineering and computer science, the method is easily accepted in practice.

Recall form Section 1.2.3 the notion of *I/O-module*. We say that the I/O-module $M_E = (\Sigma_E^{in}, \Sigma_E^{out}, T_E)$ is an *admissible environment* for M_C, if Σ_C and Σ_E are compatible, $\Sigma_E^{out} \supseteq \Sigma_C^{in}$, and $\Sigma_E^{in} \subseteq \Sigma_C^{out}$. That is, an admissible environment provides an output for each of the I/O-module's inputs, and it depends only on inputs which are output from the I/O-module. Both the environment and the I/O-module, however, may have additional outputs which are not observed by the other one.

If $M = (\Sigma^{in}, \Sigma^{out}, T)$, then M *without* w $(M \setminus w)$ is the I/O-module $(\overline{\Sigma}^{in}, \overline{\Sigma}^{out}, \overline{T})$, where $\overline{\Sigma}^{in} = \Sigma^{in} - \{w\}$, $\overline{\Sigma}^{out} = \Sigma^{out} - \{w\}$ and $\overline{T} = T \restriction_{\overline{\Sigma}^{in} \cup \overline{\Sigma}^{out}}$. I/O-module M *allows* trace x $(M \models x)$ if there exists some trace y such that x is a prefix of y and $y \restriction_{\Sigma} \in T$. Furthermore, for $\mathcal{M} \triangleq \{M_1, ..., M_n\}$, we say that $\mathcal{M} \models x$ if $M_k \models x$ for all $k \leq n$.

Assume a system consisting of a set $\mathcal{M}_C \triangleq \{M_1, \cdots, M_n\}$ of I/O-modules and a specification given as a single I/O-module $M_S \triangleq (\Sigma_S^{in}, \Sigma_S^{out}, T_S)$ such that $\Sigma_S^{in} \triangleq \bigcup \Sigma_k^{in} - \bigcup \Sigma_k^{out}$ and $\Sigma_S^{out} \subseteq \bigcup \Sigma_k^{out}$. I/O-module M_S can be thought of as an abstract specification of the concrete system M_C: all external inputs of the system M_C appear as inputs of the specification M_S, and some (but not necessarily all) outputs of the system M_C are visible in the specification M_S.

We say that $\mathcal{M}_C$ *conforms to* M_S, if for any admissible environment $M_E \triangleq (\Sigma_S^{out}, \Sigma_S^{in}, T_E)$, whenever $\{M_S, M_E\}$ is failure-free, also $\mathcal{M}_C \cup \{M_E\}$ is failure-free.

In other words, the system $\mathcal{M}_C$ may have a failure in the environment M_E only if the specification M_S allows a failure in the same context. This conformance relation is reflexive and transitive, but not symmetric: The system may be failure-free even in contexts in which the specification fails.

Various verification tools based on conformance checking have been developed [Ebe95, RCP95, McM95b]. One of the biggest advantages of the approach is that the verification can be done hierarchically. This is essential for the verification of large systems.

Although the definition of conformance does not directly lend itself for implementation in a verification procedure, there is an equivalent property which can be implemented effectively: conformance of an implementation to a specification is equivalent to the failure-freeness between the implementation and the mirror of the specification.

Note that we do not actually compose the I/O-modules constituting the

implementation. Therefore, in our approach it is not necessary to eliminate so-called autofailures, which arise from internal communication errors in a composed I/O-module. Also we do not have an explicit hiding operation: Failures resulting from the effect of hiding variables are transparent to the specification and will also be detected during the verification procedure.

Part II

Verification

Chapter 4

Completeness and Decidability

4.1 Completeness

Logicians are interested in logical truths, i.e., in the set of formulas which are valid in *all* models of the logic.

What about specific theories like the theory of groups, or the theory of a specific given model? How does it help to know about the set of *all* valid formulas when we want to find out whether a particular formula φ holds for a given model or theory?

Answer: encode the model or theory as a set of *assumptions* Φ and check whether the formula in question *follows* from Φ!

In fact, a *logic* can be defined to be any set of well-formed formulas which is closed under provable consequence; and a *theory* is a set of well-formed formulas which is closed under semantical consequence.

Thus there are three notions of consequence involved here:

- $\Phi \mid\!\!\vdash \varphi$ if from Φ follows φ,
  i.e. if any model in which all formulas from Φ are valid also validates φ,
- $\Phi \vdash \varphi$ if Φ proves φ,
  i.e. if there is a proof of φ which uses only assumptions from Φ, and
- $\Phi \rightarrow \varphi$ if Φ implies φ,
  i.e. this is a statement of the object language which is only defined if Φ is a single formula. To be liberal, we can identify a finite set of formulas $\{\varphi_1, ..., \varphi_n\}$ with the conjunction $(\varphi_1 \wedge ... \wedge \varphi_n)$.

Note that $\Phi \mid\!\!\vdash \varphi$ is different from $\mathcal{M} \models \varphi$! The notations $\mid\!\!\vdash \varphi$ and $\vdash \varphi$ are short for $\{\} \mid\!\!\vdash \varphi$ and $\{\} \vdash \varphi$, respectively.

Of course, the semantical notion of *validity* sometimes is restricted to certain classes of models, e.g., to those satisfying certain axioms, or to natural or tree models. For the sake of simplicity, we will restrict ourself to natural models in this section. However, all results apply to tree models as well. To be able to talk about propositions *and* actions, we assume that the transition from one point to the next carries a unique label $a \in \mathcal{R}$.

Also, the syntactical notion of *provability* sometimes is parametrised to a certain proof-system. In this section, we will use *Hilbert-style* proof-systems, consisting of a set of *axioms* and derivation rules. Usually, axioms and derivation rules contain *propositional variables* $q \in \mathcal{Q}$ and a substitution rule allowing consistent replacement of propositional variables with formulas. (Conceptually, propositional variables are not the same as propositions, though many authors do not distinguish between these syntactic categories.)

To complicate things even more, there are two notions of validity of a formula: *local validity* $(U, \mathcal{I}, w_0) \models \varphi$, where the evaluation point is given, and *universal validity* $(U, \mathcal{I}) \models \varphi$. Traditionally, focus has been on complete axioms for universal validity rather than for the local version; proofs are much simpler. Thus, we are interested in formulas which are valid *in all models at all points.*

One of the major concerns after defining a logical language and its models is to find an *adequate* proof-system for the logic, i.e. one which is both *correct* and *complete*. That is, for any Φ and φ,

- if $\Phi \vdash \varphi$, then $\Phi \mid\!\!\vdash \varphi$ (Correctness), and
- if $\Phi \mid\!\!\vdash \varphi$, then $\Phi \vdash \varphi$ (Completeness).

Why should these statements be valid?

Correctness should be clear: We don't want to be able to "prove" false statements. Usually correctness is very easy to show, we just have to show that the axioms are valid, and that the derivation rules only allow to deduce valid formulas from valid formulas.

Completeness is in most cases much harder to show, if not impossible. So, why is it important to show completeness? Firstly, we would like to make sure that any specification which is satisfied by a program can be proved from the program axioms, provided the specification is expressible in the logic. Secondly, and more important, in many cases decision algorithms for automated verification can be obtained from the completeness proofs.

4.1.1 Deductions in Multimodal Logic

To illustrate the basic idea, we start with a deductive system for multimodal logic for natural models. A number of similar proofs can be found in [Bur84]. We use the following axioms and rules:

(taut) propositional tautologies
(MP) $p,\ (p \to q) \vdash q$
(N) $q \vdash [a]\ q$
(K) $\vdash [a]\ p \wedge [a]\ (p \to q) \to [a]\ q$
(U) $\vdash \langle a \rangle\ q \to [a]\ q$
(L) $\vdash \langle a \rangle\ q \to [b]\ \neg q$ for $a \neq b$

To prove $\Phi \vdash \varphi$ we have to give a *derivation* of φ from the assumptions Φ, i.e., a sequence of formulas such that the last element of this sequence is φ, and every element of this sequence is either from Φ, or a substitution instance of an axiom, or the substitution instance of the consequence of a rule, where all premisses of the rule for this substitution appear already in the derivation.

As an example, let us assume $(\mathtt{p} \to \mathtt{q})$ and derive some consequences:

1.	$\mathtt{p} \to \mathtt{q}$	(ass)
2.	$[a]\ (\mathtt{p} \to \mathtt{q})$	(1, **N**)
3.	$[a]\ (\mathtt{p} \to \mathtt{q}) \to ([a]\ \mathtt{p} \to [a]\ \mathtt{q})$	(**K**, taut)
4.	$[a]\ \mathtt{p} \to [a]\ \mathtt{q}$	(2,3,MP)
5.	$\neg\mathtt{q} \to \neg\mathtt{p}$	(1)
6.	$[a]\ \neg\mathtt{q} \to [a]\ \neg\mathtt{p}$	(5)
7.	$\langle a \rangle\ \mathtt{p} \to \langle a \rangle\ \mathtt{q}$	(6)

Lines (4) and (7) form the basis for an inductive proof of the following replacement and monotonicity rules:

(repl) $p \leftrightarrow q \vdash \varphi(p) \leftrightarrow \varphi(q)$, and
(mon) $p \to q \vdash \varphi(p) \to \varphi(q)$.

(mon) requires that $\varphi(q)$ is *positive* in q, that is, that every occurrence of q is under an even number of negation signs (an exact definition will be given in section 6.5). For example, $[a]\ q$, $\langle a \rangle\ q$, $q \wedge [a]\ (q \vee \langle a \rangle\ q)$ are all positive in q.

Using these rules, we prove that **(L)** is equivalent to $[a]\ \bot \vee [b]\ \bot$:

1.	$\langle a \rangle\ q \to [b]\ \neg q$	(**L**)

2.	$\langle a\rangle \neg\bot \rightarrow [b] \neg\neg\bot$	(1)
3.	$[a]\bot \vee [b]\bot$	(2,repl)

And the other direction:

4.	$\bot \rightarrow \neg q$	(taut)
5.	$[a]\bot \rightarrow [a]\neg q$	(4, mon)
6.	$\langle a\rangle q \rightarrow \neg [a]\bot$	(5)
7.	$\neg [a]\bot \rightarrow [b]\bot$	(3)
8.	$[b]\bot \rightarrow [b]\neg q$	(4, mon)
9.	$\langle a\rangle q \rightarrow [b]\neg q$	(6,7,8)

As a more practical example, let us derive from the assumptions
$\mathtt{set} \rightarrow \langle V\rangle \neg\mathtt{set}$ and
$\neg\mathtt{set} \rightarrow \langle P\rangle \mathtt{set} \vee \langle V\rangle \neg\mathtt{set}$ the property
$[P][P]\bot$.
The assumptions can be seen as describing the actions of a semaphore with two states, set and $\neg$set, which allows to be set with a P-operation when it is not set, and to be freed with a V-operation when it is either set or not set. The given property describes that there are never two P operations in a row.

1.	$\bot \rightarrow [P]\bot$	(taut)
2.	$[P]\bot \rightarrow [P][P]\bot$	(1,mon)
3.	$\langle V\rangle \top \rightarrow [P]\bot$	(**L**)
4.	$\langle V\rangle \neg\mathtt{set} \rightarrow \langle V\rangle \top$	(mon)
5.	$\langle V\rangle \neg\mathtt{set} \rightarrow [P]\bot$	(3,4)
6.	$\langle V\rangle \neg\mathtt{set} \rightarrow [P][P]\bot$	(2,5)
7.	$\langle P\rangle\langle V\rangle \neg\mathtt{set} \rightarrow \langle P\rangle[P]\bot$	(5,mon)
8.	$\mathtt{set} \rightarrow \langle V\rangle \neg\mathtt{set}$	(ass)
9.	$\mathtt{set} \rightarrow [P][P]\bot$	(6,8)
10.	$\langle P\rangle \mathtt{set} \rightarrow \langle P\rangle\langle V\rangle \neg\mathtt{set}$	(8,mon)
11.	$\langle P\rangle \mathtt{set} \rightarrow \langle P\rangle[P]\bot$	(7,10)
12.	$\langle P\rangle[P]\bot \rightarrow [P][P]\bot$	(**U**)
13.	$\langle P\rangle \mathtt{set} \rightarrow [P][P]\bot$	(11,12)
14.	$(\langle P\rangle \mathtt{set} \vee \langle V\rangle \neg\mathtt{set}) \rightarrow [P][P]\bot$	(6,13)
15.	$\neg\mathtt{set} \rightarrow \langle P\rangle \mathtt{set} \vee \langle V\rangle \neg\mathtt{set}$	(ass)
16.	$\neg\mathtt{set} \rightarrow [P][P]\bot$	(14,15)
17.	$(\mathtt{set} \vee \neg\mathtt{set}) \rightarrow [P][P]\bot$	(9,16)
18.	$[P][P]\bot$	(17)

As we see, even in such relatively easy examples it can be quite cumbersome to find a Hilbert-style proof "by hand"; it should be possible to conduct these proofs automatically. This will be the topic of the next section!

Let us argue about the correctness of our deduction rules. **(taut)** and **(MP)** are immediately clear. **(N)** is the so called *necessitation rule*. Its validity depends on the universal interpretation of validity: If some formula is true in every point of a model, it is true in every point which is the a-successor of some other point in that model. **(K)** is the classical *Kripke*-axiom which holds for all normal modal logics. If in all accessible points p holds, and in all accessible points $(p \to q)$ holds, then in all accessible points q must hold. **(U)** is the axiom describing that the next-step relation is *univalent*: If there is any successor satisfying q, then all successors satisfy q. This holds because at any given moment, there is at most one successor which can be reached. While this is true for natural models, it does not hold for trees or other branching structures. **(L)** finally is an additional axiom for the labelling of the next-step relation by transition relations. If some a-successor satisfies q, then the next state is determined by an a-step, hence it is not a b-step, and all states reachable by a b-step are false. Again, this only holds because we are considering natural models (i.e. paths through a Kripke model, not the Kripke model as such).

4.1.2 Completeness of Multimodal Logic

The classical way to prove completeness is the so–called Henkin/Hasenjäger construction. A set Ψ of formulas is *inconsistent with* Φ, if there is a finite subset $\{\psi_1, ..., \psi_n\} \subseteq \Psi$ such that $\Phi \vdash (\neg\psi_1 \vee ... \vee \neg\psi_n)$. To prove completeness, we have to show

(*) Every formula consistent with Φ is satisfiable in a model validating Φ.

For, if $\Phi \mid\!\!\vdash \varphi$, then no model validating Φ satisfies $\{\neg\varphi\}$; therefore with **(*)** it follows that $\{\neg\varphi\}$ is inconsistent with Φ, hence $\Phi \vdash \varphi$. (Without loss of generality, we can assume here Φ to be consistent with itself, or else $\Phi \mid\!\!\vdash \varphi$ holds).

Thus, the task is to construct a model for a given consistent set of formulas. *Lindenbaum's extension lemma* states that for any formula φ which is consistent with Φ there exists a maximal consistent set w_0 such that $\varphi \in w_0$ and $\Phi \subseteq w_0$: Start with $\Phi \cup \{\varphi\}$; for every formula ψ according to a fixed enumeration add either ψ or $\neg\psi$ to w, whichever is consistent with the set constructed so far.

The *canonical model* for Φ is $(U, \mathcal{I}, w)$, where

- U is the set of maximal consistent sets which include Φ,
- $\mathcal{I}(a) \triangleq \{(w_0, w_1) \mid q \in w_1 \rightarrow \langle a \rangle\ q \in w_0\}$, and
- $\mathcal{I}(\mathtt{p}) \triangleq \{w_0 \mid \mathtt{p} \in w_0\}$, and
- w is any element from U such that $\varphi \in w$.

For every $w_0 \in U$ of our canonical model and every $a \in \mathcal{R}$ there is at most one w_1 with $(w_0, w_1) \in \mathcal{I}(a)$. For, assume $(w_0, w_1) \in \mathcal{I}(a)$ and $(w_0, w_1') \in \mathcal{I}(a)$. Then, there must be a formula ψ such that $\psi \in w_1$ and $\psi \notin w_1'$, or else $w_1 = w_1'$. Since w_1' is maximal, $\neg\psi \in w_1'$. Therefore $\langle a \rangle\ \psi \in w_0$ and $\langle a \rangle\ \neg\psi \in w_0$. But, this is a contradiction to the consistency of w_0: axiom **U** requires that if $\langle a \rangle\ \psi \in w_0$, then $\neg\ \langle a \rangle\ \neg\psi \in w_0$.

Similarly, we can show that for every $w_0 \in U$ there is at most one a such that $(w_0, w_1) \in \mathcal{I}(a)$. The opposite assumption would lead to a contradiction with axiom **L**.

The fundamental 'truth' or 'killing' lemma states that for any formula φ and maximal consistent set w it holds that $\varphi \in w$ iff $(U, \mathcal{I}, w) \models \varphi$.

In the inductive step for this lemma, we have to show that $\langle a \rangle\ \varphi \in w_0$ iff $(U, \mathcal{I}, w_0) \models \langle a \rangle\ \varphi$. The 'if' direction being a direct consequence of definition and induction hypothesis, assume that $\langle a \rangle\ \varphi \in w_0$. We have to find a maximal consistent set w_1 such that $(w_0, w_1) \in \mathcal{I}(a)$ and $\varphi \in w_1$. Since $\vdash\ (\langle a \rangle\ \varphi \wedge [a]\ \psi) \rightarrow \langle a \rangle\ (\varphi \wedge \psi)$, the set $\{\varphi\} \cup \{\psi_j \mid [a]\ \psi_j \in w\}$ is consistent. Let w_1 be any maximal consistent extension of this set. Then for all $\psi \in w_1$ the formula $\langle a \rangle\ \psi$ must be in w_0 (otherwise, a contradiction could be derived). Therefore $(w_0, w_1) \in \mathcal{I}(a)$. Since $\varphi \in w_1$, the induction hypothesis gives $(U, \mathcal{I}, w_1) \models \varphi$. Together with $(w_0, w_1) \in \mathcal{I}(a)$ we have $(U, \mathcal{I}, w_0) \models \langle a \rangle\ \varphi$.

Since for the canonical model $(U, \mathcal{I}, w)$ it holds that $\Phi \subseteq w$ and $\varphi \in w$, we proved that $(U, \mathcal{I}, w) \models \Phi$ and $(U, \mathcal{I}, w) \models \varphi$. Thus we have achieved our goal of constructing a natural model for $\Phi \cup \{\varphi\}$.

4.1.3 Completeness of Temporal Logics

How can this completeness proof be lifted to more expressive logics like **LTL**?

Let us for the moment focus on temporal logic on natural models with the operators **X** for the union of all accessibility relations and $\langle R \rangle$ for the transitive closure; the extensions for until- and since operators being almost straightforward extensions of the basic ideas.

The relation between **X** and $\langle a \rangle$ is fixed by the following axiom:

(nex) $\vdash\langle a\rangle\ q \rightarrow \mathbf{X}\ q$ for all $a \in \mathcal{R}$

A close inspection of the semantics of $\mathbf{F}^+$ reveals a fundamental problem: Consider the set $\Phi \triangleq \{\mathbf{X}\ \mathtt{p},\ \mathbf{XX}\ \mathtt{p},\ \mathbf{XXX}\ \mathtt{p},\ ...\}$. Then clearly $\Phi \mid\!\!\vdash \Box\varphi$. However, $\Phi \not\vdash \Box\varphi$, since every proof of $\Box\varphi$ from Φ can use only a limited number of premisses (proofs are *finite* sequences). But, for no finite subset $\Phi_0 \subset \Phi$ the statement $\Phi_0 \vdash \Box\varphi$ holds.

Where does the above completeness proof fail? It is not possible to find a maximal consistent extension, since we can not apply an axiom to show the consistency of an infinite set of premisses.

When dealing with second order concepts like transitive closure we have to limit ourselves to a weaker form of completeness. Call a logic *weakly complete*, if for all *finite* Φ it holds that $\Phi \mid\!\!\vdash \varphi$ implies $\Phi \vdash \varphi$.

In first order logic, the *deduction theorem* allows to discard any finite set of assumptions: $\psi \mid\!\!\vdash \varphi$ iff $\mid\!\!\vdash \forall\psi \rightarrow \varphi$, where $\forall\psi$ is the universal closure of ψ. In temporal logic, similar deduction theorem holds:

$$\psi \mid\!\!\vdash \varphi \text{ iff } \mid\!\!\vdash \psi \wedge \Box\psi \rightarrow \varphi$$

Therefore, to prove weak completeness it suffices to prove that $\mid\!\!\vdash \varphi$ implies $\vdash \varphi$.

We use the following axiom set (in addition to the modal axioms above):

(Rec) $\vdash \mathbf{X}\ (q \vee \mathbf{F}^+\ q) \rightarrow \mathbf{F}^+\ q$
(Ind) $\mathbf{X}\ (p \vee q) \rightarrow q \vdash \mathbf{F}^+\ p \rightarrow q$

Dually, this can be written as

(Rec) $\vdash \Box q \rightarrow \neg\ \mathbf{X}\ \neg(q \wedge \Box q)$
(Ind) $q \rightarrow \neg\ \mathbf{X}\ \neg(p \wedge q) \vdash q \rightarrow \Box p$

These are the so-called *Segerberg axioms* [Seg82, personal communication] reflecting the definition of the transitive closure as the minimal transitive relation which includes all $a \in \mathcal{R}$. **(Rec)** is the *recursion* axiom which can be used to unfold a Box-operator:

$$\Box\varphi \rightarrow \neg\ \mathbf{X}\ \neg(\varphi \wedge \neg\ \mathbf{X}\ \neg(\varphi \wedge \neg\ \mathbf{X}\ \neg(\varphi \wedge ...))).$$

(Ind) is the *induction* axiom which can be used to derive a property $\Box\varphi$ from an *invariant* ψ, i.e. from a formula ψ for which $\psi \rightarrow [a]\ \psi$ and $\psi \rightarrow [a]\ \varphi$ are derivable (for all $a \in \mathcal{R}$).

How do these axioms prove completeness of the transitive closure relation? Up to the truth lemma, the proof is almost the same as for modal logic. But, we only use *finite* maximal consistent sets: we start with a single (finite) consistent formula φ for which we have to construct a model. Define the notion of *extended sub-formula* of φ (sometimes also called *Fisher-Ladner closure*) as follows:

- φ is an extended sub-formula of φ,
- $\neg\varphi$ is an extended sub-formula of φ, if φ is not of form $\neg\varphi'$,
- φ_1 and φ_2 are extended sub-formulas of $(\varphi_1 \to \varphi_2)$, (thus φ is an extended sub-formula of $\neg\varphi$)
- φ is an extended sub-formula of $\mathbf{X}\,\varphi$,
- $\mathbf{X}\,\varphi$ is an extended sub-formula of $\mathbf{F}^+\,\varphi$, and
- $\mathbf{X}\mathbf{F}^+\,\varphi$ is an extended sub-formula of $\mathbf{F}^+\,\varphi$

For any given formula, there are finitely many different extended sub-formulas. Now, a consistent set of formulas is called *finitely maximal*, if it is maximal with respect to extended sub-formulas; that is, for every extended sub-formula ψ of φ, either ψ or $\neg\psi$ is in the finitely maximal consistent set.

In the proof of the truth lemma we additionally have to show

$$\mathbf{F}^+\,\varphi \in w_0 \text{ iff } (U, \mathcal{I}, w_0) \models \mathbf{F}^+\,\varphi.$$

One direction again is easy: If $\Box\varphi \in w_0$, then $\Box\varphi$ must be in any finitely maximal consistent set reachable from w_0 by any number of steps, because the recursion axiom forces $\neg\,\mathbf{X}\,\neg\Box\varphi$ to be in w_0, and hence $\Box\varphi$ is in every w_1 with $(w_0, w_1) \in \mathcal{I}(a)$.

The other direction follows from the induction axiom: Assume that $\mathbf{F}^+\,\varphi \in w_0$, but no finitely maximal consistent set reachable from w_0 has φ in it. Let $\Psi_1, ..., \Psi_n$ be all different finitely maximal consistent sets in the same strongly connected component as w_0, and $\Psi \triangleq \Psi_1 \vee ... \vee \Psi_n$, where $\Psi_i \triangleq \bigwedge\{\psi \mid \psi \in \Psi_i\}$ (remember that the Ψ_i are finite). Then

- $\vdash w_0 \to \Psi$, since w_0 is one of the Ψ_i of which Ψ is composed. Furthermore,
- $\vdash \Psi \to \neg\varphi$, since $\neg\varphi$ was assumed to hold in the whole component. Finally,

- $\vdash \Psi \rightarrow \neg\, \mathbf{X}\, \neg\Psi$, since Ψ consists of *all* finitely maximal consistent sets in this component.

Putting these parts together, we have a contradiction, since the induction axiom gives $\vdash w_0 \rightarrow \Box\neg\varphi$, but $\mathbf{F}^+\, \varphi \in w_0$.

Completeness on natural models

The above proof can be easily extended to more expressive branching time logics such as **CTL**; see, e.g., [CS01]. We now show how to prove completeness for **LTL** on natural models. Several elaborate proofs can be found in the literature [Pri57, GPSS80, Bur84, LPZ85, Krö87], where the setting in [LPZ85] is closest to ours. Let $\mathbf{X}^-\, \varphi \triangleq (\bot\, \mathbf{U}^-\, \varphi)$ and $\widetilde{\mathbf{X}}^-\, \varphi \triangleq \neg\, \mathbf{X}^-\, \neg\varphi$ be *strong* and *weak previous* operators. The following axioms are adequate:

(N)	$q \vdash (\widetilde{\mathbf{X}}\, q \wedge \widetilde{\mathbf{X}}^-\, q)$
(K)	$\vdash (\widetilde{\mathbf{X}}\, (p \rightarrow q) \rightarrow (\widetilde{\mathbf{X}}\, p \rightarrow \widetilde{\mathbf{X}}\, q))$
	$\vdash (\widetilde{\mathbf{X}}^-\, (p \rightarrow q) \rightarrow (\widetilde{\mathbf{X}}^-\, p \rightarrow \widetilde{\mathbf{X}}^-\, q))$
(B')	$\vdash (p \rightarrow (\widetilde{\mathbf{X}}\mathbf{X}^-\, p \wedge \widetilde{\mathbf{X}}^-\mathbf{X}\, p)$
(Init)	$\vdash (\widetilde{\mathbf{X}}^-\, \bot \vee \mathbf{F}^-\widetilde{\mathbf{X}}^-\, \bot)$
(U)	$\vdash (\mathbf{X}\, p \rightarrow \widetilde{\mathbf{X}}\, p)$
	$\vdash (\mathbf{X}^-\, p \rightarrow \widetilde{\mathbf{X}}^-\, p)$
(RecU$^+$)	$\vdash (\mathbf{X}\, (q_2 \vee q_1 \wedge (q_1\, \mathbf{U}^+\, q_2)) \rightarrow (q_1\, \mathbf{U}^+\, q_2))$
(RecU$^-$)	$\vdash (\mathbf{X}^-\, (q_2 \vee q_1 \wedge (q_1\, \mathbf{U}^-\, q_2)) \rightarrow (q_1\, \mathbf{U}^-\, q_2))$
(IndU$^+$)	$(\mathbf{X}\, (q_2 \vee q_1 \wedge p) \rightarrow p) \vdash ((q_1\, \mathbf{U}^+\, q_2) \rightarrow p)$
(IndU$^-$)	$(\mathbf{X}^-\, (q_2 \vee q_1 \wedge p) \rightarrow p) \vdash ((q_1\, \mathbf{U}^-\, q_2) \rightarrow p)$

Axiom **(B')** relates future and past operators. It corresponds to the fact that the relation described by $\mathbf{X}^-$ is the converse of that described by $\mathbf{X}$. Axiom **(Init)** can be translated as $\forall x \exists y \leq x \neg\exists z (z \prec y)$; it guarantees the existence of an initial point.

Axiom **(U)** reflects the fact that **LTL** is a linear time logic, which is interpreted on natural models. With infinite maximal consistent sets, it forces the transition relation of the canonical model to be deterministic in both directions. For any w, there is at most one u such that $w \prec u$ and at most one v such that $v \prec w$. However, this is not true for the definition of $\mathcal{I}(\prec)$ in the finite canonical model (where points are finitely maximal consistent sets). Assume that $\mathtt{p} \in ESF(\varphi)$, but $\mathbf{X}\ \mathtt{p} \notin ESF(\varphi)$. For any finitely maximal consistent set w such that neither $\vdash (\hat{w} \rightarrow \widetilde{\mathbf{X}}\, \mathtt{p})$ nor $\vdash (\hat{w} \rightarrow \widetilde{\mathbf{X}}\, \neg\mathtt{p})$, there will be w_1 and w_2 with $w \prec w_1$, $w \prec w_2$, and $\mathtt{p} \in w_1$, $\mathtt{p} \notin w_2$. To overcome this difficulty, we have to thread a path through the

finite canonical model. A *strongly connected component* (SCC) is a set W of points such that for all $w_1, w_2 \in W$, if $w_1 \neq w_2$, then there is a path from w_1 to w_2 and back. An SCC W is called *terminal*, if for all $w \in W$, $w' \notin W$, it is not the case that $w \prec w'$. It is called *self-fulfilling*, if for any $w \in W$ and $(\varphi_1\ \mathbf{U}^+\ \varphi_2) \in w$ there exists a $w' \in W$ such that $\varphi_2 \in w'$. In the finite canonical model, any terminal SCC must be self-fulfilling. This can be proved from axiom **(RecU$^+$)**: Similar to above, if $w \in W$ and $(\varphi_1\ \mathbf{U}^+\ \varphi_2) \in w$ there exists a $w' > w$ such that $\varphi_2 \in w'$. Since W is terminal, it follows that $w' \in W$. A path σ through the finite canonical model for φ is called *accepting*, if

- σ starts in an initial point (without predecessors),
- σ contains a point w_0 such that $\varphi \in w_0$,
- σ is finite and ends in a terminal point, or
- σ is infinite and there is a terminal SCC W such that $\mathit{inf}(\sigma) = W$.

Recall that $\mathit{inf}(\sigma)$ is the set of points occurring infinitely often in σ. Any accepting path σ constitutes a natural model for φ. If σ infinitely often traverses all points in a nontrivial terminal SCC, then for all $w_i \in \sigma$ such that $(\varphi_1\ \mathbf{U}^+\ \varphi_2) \in w_i$ there will be some $w_{i+n} \in \sigma$ such that $\varphi_2 \in w_{i+n}$. A similar statement holds, if σ is finite. Thus, we have to show that such a path exists. We use the finite canonical model for $\varphi' \triangleq \varphi \wedge (\mathbf{X}^- \bot \vee \mathbf{F}^- \mathbf{X}^- \bot)$. In this model, there exists a point $w_{\mathit{init}} \leq w_0$ such that $\mathbf{X}^- \bot \in w_{\mathit{init}}$. Since w_{init} is consistent, there is no $\mathbf{X}^-\ \psi$ in w_{init}. From axiom **(B')** we can prove that there is no w such that $w \prec w_{\mathit{init}}$. Therefore, w_{init} is an initial point. For any $(\varphi_1\ \mathbf{U}^-\ \varphi_2) \in w_0$ there is a point $w_1 < w_0$ such that $\varphi_2 \in w_1$ and $\varphi_1 \in w_2$ for all $w_1 < w_2 < w_0$ as above. Let w_{term} be a point in any terminal SCC reachable from w_0. For all $(\varphi_1\ \mathbf{U}^+\ \varphi_2) \in w_0$ we can prove: either there exists a point $w_0 < w \leq w_{\mathit{term}}$ such that $\varphi_2 \in w$, or $(\varphi_1\ \mathbf{U}^+\ \varphi_2) \in w_{\mathit{term}}$. Since W is self-fulfilling, any path from w_{init} via w_0 which repeatedly traverses all points in W is accepting. □

This completes the completeness proof for **LTL**. A detailed exposition of this proof can be found in [Krö87].

4.1.4 Completeness of the μ-calculus

We just briefly indicate how the above axioms can be extended for μ**TL**.

(Recν) $\vdash \nu q\ \varphi(q) \rightarrow \varphi(\nu q\ \varphi(q))$
(Indν) $q \rightarrow \varphi(q) \vdash q \rightarrow \nu p\ \varphi(p)$

The recursion and induction axiom can be obtained as special cases of these very general axioms by defining $\Box p \triangleq \nu q \bigwedge_i [a_i] (p \wedge q)$.

The above completeness proof can be adapted to show completeness for a certain subclass of monotonic $\mu\mathbf{TL}$ formulas, the *aconjunctive* ones.

The problem of completeness of these axioms for *all* $\mu\mathbf{TL}$ formulas was solved in [Wal95]. It can be shown that for any formula there exists an equivalent aconjunctive formula. Thereby it suffices to derive this aconjunctive formula from the axioms in order to prove any given formula.

This proof also applies to tree models. In general, however, for each class of models under consideration the completeness question has to be solved independently.

4.2 Decidability

In this section we derive *decision procedures* for some of the logics introduced above. We already indicated that the decision procedures will be extracted from the completeness proofs of the previous section. In the next section, this line of thought is continued to derive *model checking algorithms* from the decision procedures.

Given a set Φ of assumptions, and a formula φ. We want to decide whether $\Phi \mid\!\!\!\vdash \varphi$, which by completeness is the same as $\Phi \vdash \varphi$. Now $\Phi \mid\!\!\!\vdash \varphi$ iff $\Phi \cup \{\neg\varphi\}$ is (universally) unsatisfiable. Hence we need an algorithm which, given a set of formulas, decides whether this set has a model or not.

4.2.1 Modal Decision Algorithms

We considered completeness of modal logic (with $\langle a \rangle$-operators) and of temporal logic (with operators $\mathbf{X}$ and $\mathbf{F}^+$). The completeness proof of temporal logic depended on the fact that we could use *finite* maximal consistent sets to construct our model. For multimodal logic, we allowed infinite sets of assumptions because we wanted to show strong completeness. If we restrict ourselves to the weak notion of completeness ($\Phi \mid\!\!\!\vdash \varphi \Rightarrow \Phi \vdash \varphi$ for *finite* Φ), then also here it is not necessary that maximal consistent sets are maximal in the space of all formulas. It is sufficient to consider maximality with respect to all *extended subformulas* of the given consistent set.

Let us quickly recall the definition of *extended subformula* for multimodal formulas:

- $SF(\bot) \triangleq \{\bot\}$
- $SF((\varphi \to \psi)) \triangleq \{(\varphi \to \psi)\} \cup SF(\varphi) \cup SF(\psi)$

- $SF(\mathtt{p}) \triangleq \{\mathtt{p}\}$
- $SF(\langle a\rangle\ \varphi) \triangleq \{\langle a\rangle\ \varphi\} \cup SF(\varphi)$
- $NSF(\varphi) \triangleq \{\neg\psi \mid \psi \in SF(\varphi)\}$
- $ESF(\varphi) \triangleq SF(\varphi) \cup NSF(\varphi)$
- $ESF(\Phi) \triangleq \bigcup\{ESF(\varphi) \mid \varphi \in \Phi\}$

Expanding the definition, we see that for any formula $\varphi = (\psi_1 \wedge \psi_2)$ or $\varphi = (\psi_1 \vee \psi_2)$, all $\{\psi_1, \psi_2, \neg\psi_1, \neg\psi_2\} \subseteq ESF(\varphi)$.

For any finite Φ, there are only finitely many different extended subformulas, and hence only finitely many sets of extended subformulas. Any such set is called *maximal* with respect to Φ, if for any $\psi \in ESF(\Phi)$, either $\psi \in w$ or $\neg\psi \in w$.

Call such a set w of subformulas *propositionally consistent*, if

$\bot \notin w$, and

if $(\psi_1 \to \psi_2) \in ESF(\varphi)$ then
$(\psi_1 \to \psi_2) \in w$ iff $\psi_1 \in w$ implies $\psi_2 \in w$.

That is, if $(\psi_1 \to \psi_2) \in w$ then $\neg\psi_1 \in w$ or $\psi_2 \in w$, and if one of $\neg\psi_1$, $\psi_2 \in w$ then $(\psi_1 \to \psi_2) \in w$. Again, expanding the definitions we see that

- for any $(\psi_1 \wedge \psi_2) \in ESF(\varphi)$,
$(\psi_1 \wedge \psi_2) \in w$ iff both $\psi_1 \in w$ and $\psi_2 \in w$,
- for any $(\psi_1 \vee \psi_2) \in ESF(\varphi)$, $(\psi_1 \vee \psi_2) \in w$ iff $\psi_1 \in w$ or $\psi_2 \in w$.

Any propositionally maximal consistent sets is "consistent for propositional logic": if we consistently replace any modal formula in w by a new proposition, then the resulting set of formulas is satisfiable in propositional logic. A satisfying propositional interpretation is given by $\mathcal{I}(\mathtt{p}) \triangleq$ **true** iff $\mathtt{p} \in w$.

The modal formulas in w determine the structure of the accessibility relation(s) in any model for Φ, if such a model exists. There are two approaches to construct these accessibility relations.

The first, 'local' algorithm, is tableaux-based. Start with the set $w_0, ..., w_n$ of all propositionally maximal consistent sets which include Φ and try to systematically extend one of these to a model. Given a propositionally maximal consistent set w_i, if it does not contain any formula $\langle a\rangle\ \psi$, we are finished. If it contains both some $\langle a\rangle\ \psi_1$ and some $\langle b\rangle\ \psi_2$, then this set is unsatisfiable (because we are considering models in which the labelling

of any arc from some point is unique); thus we backtrack. Otherwise, there is exactly one a such that some formulas $\langle a\rangle\ \psi \in w_i$. Construct the sets $w_i' \triangleq \{\psi \mid \langle a\rangle\ \psi \in w_i\} \cup \{\neg\psi \mid \neg\ \langle a\rangle\ \psi \in w_i\}$ and $w_i'' \triangleq w_i' \cup \Phi$ (We are considering *linear* accessibility relations and *universal* consequence!). There are finitely many propositionally maximal consistent extensions $w_{i.1}, ..., w_{i.n}$ of w_i''.

If w_i'' is not propositionally consistent, then there is no such extension ($n = 0$): backtrack. Otherwise, recurse with all propositionally maximal consistent extensions $w_{i.j}$ of w_i and continue *ad infinitum*. No wait; that might take too long. Since there are only finitely many propositionally consistent sets, we will hit onto a cycle sooner or later. In that case, we are also finished: we have constructed a model consisting of an infinite loop.

4.2.2 Modal Tableau Rules

Before giving a example, we give a set of tableau rules which can be seen as another formulation of this idea. A large number of similar tableau rules for all sorts of modal logics can be found in [Fit83].

Let Φ be the set of formulas whose satisfiability we have to check. Γ is any set of formulas;

$$(\rightarrow)\ \frac{\Gamma, (\psi_1 \rightarrow \psi_2)}{\Gamma, \neg\psi_1 \qquad \Gamma, \psi_2} \qquad (\neg\rightarrow)\ \frac{\Gamma, \neg(\psi_1 \rightarrow \psi_2)}{\Gamma, \psi_1, \neg\psi_2}$$

$$(\bot_1)\ \frac{\Gamma, \psi, \neg\psi}{*} \qquad (\bot_2)\ \frac{\Gamma, \bot}{*} \qquad (\mathrm{L})\frac{\Gamma, \langle a\rangle\ \psi_1, \langle b\rangle\ \psi_2}{*}$$

$$(\neg\neg)\ \frac{\Gamma, \neg\neg\psi}{\Gamma, \psi}$$

$$(\langle\cdot\rangle)\ \frac{\Gamma, \langle a\rangle\ \varphi_1, ..., \langle a\rangle\ \varphi_n, \neg\ \langle a\rangle\ \psi_1, ..., \neg\ \langle a\rangle\ \psi_m}{\Phi, \varphi_1, ..., \varphi_n, \neg\psi_1, ..., \neg\psi_m} \qquad ([\cdot])\ \frac{\Gamma}{}$$

Derived rules:

$$(\vee)\ \frac{\Gamma, (\psi_1 \vee \psi_2)}{\Gamma, \psi_1 \qquad \Gamma, \psi_2} \qquad (\neg\vee)\ \frac{\Gamma, \neg(\psi_1 \vee \psi_2)}{\Gamma, \neg\psi_1, \neg\psi_2}$$

$$(\wedge)\ \frac{\Gamma, (\psi_1 \wedge \psi_2)}{\Gamma, \psi_1, \psi_2} \qquad (\neg\wedge)\ \frac{\Gamma, \neg(\psi_1 \wedge \psi_2)}{\Gamma, \neg\psi_1 \qquad \Gamma, \neg\psi_2}$$

$$(\neg\ \langle a\rangle)\ \frac{\Gamma, \neg\ \langle a\rangle\ \psi}{\Gamma, [a]\ \neg\psi} \qquad (\neg\ [a])\ \frac{\Gamma, \neg\ [a]\ \psi}{\Gamma, \langle a\rangle\ \neg\psi}$$

$$(\mathbf{L}')\ \frac{\Gamma, \langle a\rangle\ \psi_1, [b]\ \psi_2}{\Gamma, \langle a\rangle\ \psi_1} \qquad (\mathbf{U})\ \frac{\Gamma, \langle a\rangle\ \psi_1, [a]\ \psi_2}{\Gamma, \langle a\rangle\ \psi_1, \langle a\rangle\ \psi_2}$$

The tableau rules allow to derive a set of sets of formulas from any set of formulas. Additional regulations are:

- Rule $(\rightarrow)$ can only be applied if $\psi_2 \neq \bot$
- Rules $(\langle\cdot\rangle)$ and $([\cdot])$ can only applied if no other rule is applicable.
- Rule $(\langle\cdot\rangle)$ can only be applied if no other $\langle a\rangle\ \varphi$ or $\neg\ \langle a\rangle\ \psi$ is in Γ
- Rule $([\cdot])$ can only be applied if no $\langle a\rangle\ \varphi$ is in Γ

A *tableau* is a finite tree of sets of formulas such that

- The root of the tableau is Φ, and
- The successors of each node are constructed according to some tableau rule.

A leaf is called *closed*, if it consists of the symbol $*$. It is called *open*, if it consists of a subset of formulas of some other node on the path from the root to this leaf. (In particular, if rule $(\langle\cdot\rangle)$ regenerates the root Φ, the new leaf is open. Also, any empty node constructed by rule $([\cdot])$ is open). A tableau is *completed*, if any leaf is closed or open. A completed tableau is *successful*, if it contains an open leaf.

The tableau rules are formulated in a nondeterministic way, since we did not specify any order in which the rules have to be applied. Nevertheless all tableaus for a given formula are equivalent: If Φ has any successful tableau, then every completed tableau for Φ is successful.

4.2.3 Adequacy of the Modal Tableau Procedure

Φ is satisfiable iff Φ has a successful tableau.

The proof of this statement is more or less straightforward: Assume Φ is satisfiable in a natural model $\mathcal{M} \triangleq ((w_0, w_1, w_2, ...), \mathcal{I}, w_0)$, and show that there is a tableau for Φ with an open leaf. Equivalently, assume that any tableau for Φ is given, and show that it contains an open leaf. We construct a sequence of tableau nodes n_i, and associate a point $w(n_i)$ in the model with any n_i. As an invariant of this construction, we show that for all formulas $\psi \in n_i$ it holds that $w(n_i) \models \psi$. Initially n_0 is the root of the tableau, with $w(n_0) \triangleq w_0$. Since $w_0 \models \Phi$, the invariant is satisfied. Given any tableau node n_i with $w(n_i) = w_j$, no closing rules can be applicable, because this would contradict the invariant. Assume the successor of w_i is constructed by rule $(\neg \rightarrow)$ or $(\neg\neg)$. Then $w(n_{i+1}) \triangleq w_j$, and the invariant is

preserved. If two successors of w_i are constructed by rule $(\rightarrow)$, then any one of them is chosen which preserves the invariant, and again $w(n_{i+1}) \triangleq w_j$. If n_i has a successor obtained by rule $(\langle\rangle)$, then $w(n_{i+1}) \triangleq w_{j+1}$. The specific formulation of the rule guarantees that the invariant is preserved. Since the tableau is finite, and we can never apply one of the closing rules, we must hit onto an open leaf sooner or later.

For the other direction, we have to show that from any tableau with open leafs we can construct a model. The construction is similar to above. We consider the *unfolding* of the tableau, which is the tree arising from the repeated substitution of any open leaf with the subtableau rooted at the node subsuming this open leaf. If the tableu contains open leafs, then the unfolding contains infinite paths. In the unfolding, call any node whose successor is constructed by rule $(\langle\rangle)$ or $([])$ a *pre-state*. The set of pre-states of any infinite path from the root constitutes an infinite model.

As an example for the tableau construction, we again prove $[P][P]\ \bot$ from the assumptions

$$\Phi = \{(\mathtt{s} \rightarrow \langle V\rangle\ \neg\mathtt{s}), (\neg\mathtt{s} \rightarrow \langle P\rangle\ \mathtt{s} \vee\ \langle V\rangle\ \neg\mathtt{s})\}.$$

A formula is valid, if its negation is unsatisfiable; hence we start the tableau with $(\mathtt{s} \rightarrow \langle V\rangle\ \neg\mathtt{s})$, $(\neg\mathtt{s} \rightarrow \langle P\rangle\ \mathtt{s} \vee\ \langle V\rangle\ \neg\mathtt{s})$, and $\langle P\rangle\langle P\rangle\ \top$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\Phi,\ \ \langle P\rangle\langle P\rangle\ \top}{\neg\mathtt{s},\ \ \langle P\rangle\ \mathtt{s},\ \ \langle P\rangle\langle P\rangle\ \top} \qquad \dfrac{\langle V\rangle\ \neg\mathtt{s}, \langle P\rangle\langle P\rangle\ \top}{*} \qquad \dfrac{\ldots}{*}}{\Phi,\ \mathtt{s},\ \ \langle P\rangle\ \top}}{\mathtt{s}, \langle V\rangle\ \neg\mathtt{s},\ \ \langle P\rangle\ \top}}{*}$$

Here the dots indicate a number of other branches closed by rule (**L**). In the completed tableau, since every leaf is closed, the original formula $[P][P]\ \bot$ follows from the assumptions Φ.

This example exhibits the connection between the tableau method and the local satisfiability algorithm sketched above: The propositional tableau rules systematically generate all necessary propositionally maximal consistent extensions of a given set of formulas, and the modal rules fix the structure of the accessibility relations in the generated model graph.

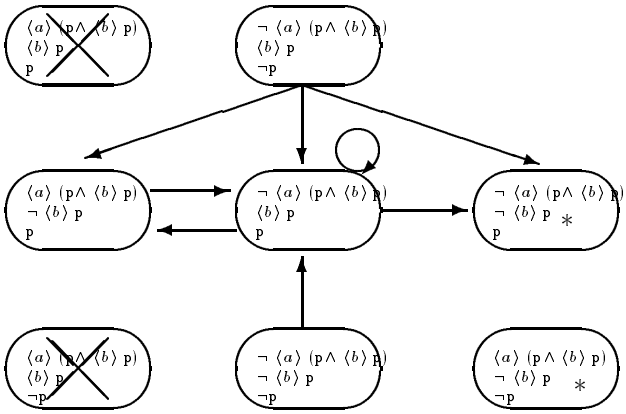
4.2.4 Global Modal Satisfiability

The second, 'global' algorithm for testing satisfiability of a set of formulas starts with the set W of all propositionally maximal consistent sets and the universal relation for any $\langle a\rangle$ operator. We first delete all nodes which

contain both some $\langle a \rangle\ \psi_1$ and $\langle b \rangle\ \psi_2$. Then we iteratively delete 'bad arcs' and 'bad nodes' until stabilisation is reached. *Bad arcs* are pairs $(w_0, w_1) \in \mathcal{I}(a)$ such that w_0 contains $\langle b \rangle\ \psi'$ for some $b \neq a$, or $\langle a \rangle\ \psi$ or $[a]\ \psi$, but it is not the case that $\psi \in w_1$. *Bad nodes* w_0 contain a formula $\langle a \rangle\ \psi$, but there does not (or no longer) exist a tuple $(w_0, w_1) \in \mathcal{I}(a)$ with $\psi \in w_1$. The given formula set Φ is satisfiable iff upon termination there is a node n left in which it is included ($\Phi \subseteq n$).

Since this algorithm iterates on all nodes and on all subformulas, we can implement it by a search on all nodes, with nested iteration on all subformulas of this node, or by a bottom up iteration on all diamond-subformulas, where we check all node whether they are 'bad' with respect to this subformula. In both cases, it is important to re-iterate after some deletions have taken place, until stabilisation is reached.

As a simple example for global satisfaction, we show the result of constructing all models for $\langle a \rangle\ (\mathtt{p} \wedge \langle b \rangle\ \mathtt{p})$.



(The starred ovals indicate points which are connected to all other points in the picture.)

4.2.5 Decidability for Branching Time

It is almost obvious how to extend both of the above approaches for branching time, i.e., for tree models in which every point can have arbitrarily many successors for each accessibility relation.

In the "local approach", we build a tree of trees: Again, we start with the set $w_0, ..., w_n$ of all propositionally maximal consistent sets which include

Φ. By backtracking, we try to extend one of these to a tree model. Given a propositionally maximal consistent set w_i, if it does not contain any formula $\langle a\rangle\ \psi$, this branch of the tree is finite. In contrast to linear models, the set $\{\langle a\rangle\ \psi_1, \langle b\rangle\ \psi_2\}$ can be satisfied by constructing *two* successor nodes. In general, if w_i contains diamond-formulas $\psi_1, ..., \psi_n$, there will be n successor nodes of w_i in the constructed tree model. Let $S(w_i)$ be the set of all n-tuples $(\Psi_1, ..., \Psi_n)$ of propositionally maximal consistent sets satisfying

If $\psi_j = \langle a\rangle\ \psi$, then Ψ_j is any maximal consistent extension of $\psi \cup \{\psi' \mid [a]\ \psi' \in w_i\}$.

Every n-tuple in $S(w_i)$ determines a set of possible tree-successors of w_i in the constructed tree; therefore we have to recurse and backtrack on *all* elements of *all* of these n-tuples. Formally, a leaf w_i is open if it is included in some node above, or if for *some* n-tuple $(\Psi_1, ..., \Psi_n) \in S(w_i)$, *all* Ψ_i are open.

To formulate this procedure with tableaus, we use *non-determined* tableau rules: For any given formula, there can be both successful and unsuccessful tableaus. The formula is satisfiable, if there is at least one successful tableau. (Thus, in the worst case, all possible tableaus have to be checked.) We only give the rules which are different from above:

$$(\rightarrow_1)\ \frac{\Gamma, (\psi_1 \rightarrow \psi_2)}{\Gamma, \neg\psi_1} \qquad (\rightarrow_2)\ \frac{\Gamma, (\psi_1 \rightarrow \psi_2)}{\Gamma, \psi_2}$$

$$(\langle\rangle)\ \frac{\Gamma, \Psi}{\Phi, \Psi_1 \quad ... \quad \Phi, \Psi_n}$$

where $\Psi = \{\langle a_1\rangle\ \psi_1, ..., \langle a_n\rangle\ \psi_n\}$, no $\langle a\rangle\ \psi$ is in Γ, and $\Psi_i \triangleq \{\psi_i\} \cup \{\psi \mid [a_i]\ \psi \in \Gamma\}$.

It seems to be much easier to abridge the 'global' algorithm for satisfiability testing to the branching paradigm. Again, we start with the set W of all propositionally maximal consistent sets and the universal relation for any $\langle a\rangle$ operator, and delete bad arcs and bad nodes until stabilisation is reached. Here, bad arcs are pairs $(w_0, w_1) \in \mathcal{I}(a)$ such that w_0 contains $[a]\ \psi$, but it is not the case that $\psi \in w_1$. Bad nodes w_0 contain a formula $\langle a\rangle\ \psi$, but there does not (or no longer) exist a tuple $(w_0, w_1) \in \mathcal{I}(a)$ with $\psi \in w_1$.

Hence, on the multimodal level, it appears that for natural models the local algorithm is simpler, whereas for tree models the global algorithm is easier to implement. In fact, many automatic provers for linear time are

tableau based, and many provers for branching time are global. However, the average complexity in both cases depends largely on the structure of the formulas which are to be proven.

4.2.6 Tableaus for LTL

Can we extend these methods for transitive closure operators?

Firstly, in the definition of extended subformula, all $\mathbf{X}\ \varphi$ should be regarded as extended subformula of the formula $\mathbf{F}^+\ \varphi$. Furthermore, we define also $\mathbf{XF}^+\ \varphi$ to be an extended subformula of $\mathbf{F}^+\ \varphi$. We have to be a bit careful to avoid a nonterminating recursion in the algorithmic reformulation of the recursive definition:

- $SF(\bot) \triangleq \bot$
- $SF((\varphi \to \psi)) \triangleq \{(\varphi \to \psi)\} \cup SF(\varphi) \cup SF(\psi)$
- $SF(\mathtt{p}) \triangleq \{\mathtt{p}\}$
- $SF(\mathbf{X}\ \varphi) \triangleq \{\mathbf{X}\ \varphi\} \cup SF(\varphi)$
- $SF(\mathbf{F}^+\ \varphi) \triangleq \{\mathbf{F}^+\ \varphi, \mathbf{X}\ \varphi\} \cup SF(\varphi)$
- $TSF(\varphi) \triangleq SF(\varphi) \cup \{\mathbf{XF}^+\ \psi, \mathbf{X} \neg\ \mathbf{F}^+\ \psi\ |\mathbf{F}^+\ \psi \in SF(\varphi)\}$
- $NSF(\varphi) \triangleq \{\neg\psi \mid \psi \in TSF(\varphi)\}$
- $ESF(\varphi) \triangleq SF(\varphi) \cup NSF(\varphi)$
- $ESF(\Phi) \triangleq \bigcup\{ESF(\varphi) \mid \varphi \in \Phi\}$

Now, this definition of extended subformula guarantees that for any $\mathbf{F}^+\ \psi$ in w_i, all possible successors will contain $\mathbf{F}^+\ \psi$ or $\neg\ \mathbf{F}^+\ \psi$ as well.

For **LTL** on natural models, we try to construct a linear path through the space of all propositionally maximal consistent sets by depth-first-search. If a node w_i contains some formula $\mathbf{F}^+\ \psi$, but no formula $\mathbf{X}\ \psi'$, we can discard it, because the eventuality $\mathbf{F}^+\ \psi$ is not fulfilled. Also, if some candidate successor w_i' of node w_i with $\mathbf{F}^+\ \psi \in w_i$ contains $\neg\ \mathbf{F}^+\ \psi$ and $\neg\psi$, we can discard w_i'. But, if w_i' contains $\mathbf{F}^+\ \psi$ again, this *unsatisfied eventuality* could be propagated, resulting in a cycle where the fulfilment of $\mathbf{F}^+\ \varphi$ is infinitely delayed. The solution is to require that a backward loop only can be regarded as open, if for any $\mathbf{F}^+\ \psi$ which occurs in any w_i in the loop, there must be a w_j in the loop such that $\psi \in w_j$.

For the tableau, we add the rules:

$$(\mathbf{X})\ \frac{\Gamma, \mathbf{X}\ \psi}{\Gamma, \langle a_1\rangle\ \psi \quad ... \quad \Gamma, \langle a_n\rangle\ \psi}$$

$$(\neg\ \mathbf{X})\ \frac{\Gamma, \neg\ \mathbf{X}\ \psi}{\Gamma, \neg\ \langle a_1\rangle\ \psi, ..., \neg\ \langle a_n\rangle\ \psi}$$

$$(\mathbf{F}^+)\ \frac{\Gamma, \mathbf{F}^+\ \psi}{\Gamma, \mathbf{X}\ (\psi \vee \mathbf{F}^+\ \psi)} \qquad (\neg\ \mathbf{F}^+)\ \frac{\Gamma, \neg\ \mathbf{F}^+\ \psi}{\Gamma, \neg\ \mathbf{X}\ (\psi \vee \mathbf{F}^+\ \psi)}$$

These rules are based on the unfolding of the $\mathbf{X}$- and $\mathbf{F}^+$-operators:

- $\mathbf{X}\ \psi \leftrightarrow \bigvee_a \langle a\rangle\ \psi$
- $\mathbf{F}^+\ \psi \leftrightarrow \mathbf{X}\ (\psi \vee \mathbf{F}^+\ \psi)$

Additionally we have to require that a leaf is only called open, if for all formulas $\mathbf{F}^+\ \psi$ occurring in it, the formula ψ is contained in some node between the leaf and its subsuming ancestor (*loop condition*).

As an example for the loop condition, we show that the formula $\mathbf{F}^+\ \bot$ is unsatisfiable:

$$\dfrac{\dfrac{\mathbf{F}^+\ \bot}{\mathbf{X}\ (\bot \vee \mathbf{F}^+\ \bot)}}{\dfrac{\dfrac{\langle a_1\rangle\ (\bot \vee \mathbf{F}^+\ \bot)}{\bot \vee \mathbf{F}^+\ \bot}}{\dfrac{\bot}{*} \quad \dfrac{\mathbf{F}^+\ \bot}{*}}} \quad \cdots \quad \dfrac{\dfrac{\langle a_n\rangle\ (\bot \vee \mathbf{F}^+\ \bot)}{\bot \vee \mathbf{F}^+\ \bot}}{\dfrac{\bot}{*} \quad \dfrac{\mathbf{F}^+\ \bot}{*}}$$

Each left branch closes because of rule $(\bot)$, each right branch is closed because it forms a loop with unsatisfied eventuality $\mathbf{F}^+\ \bot$.

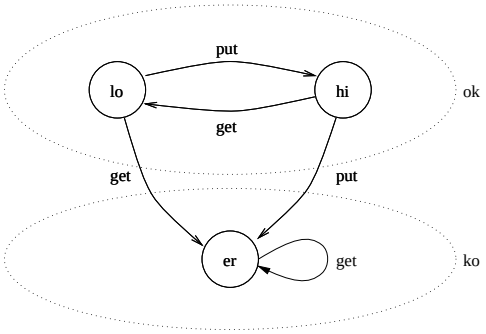
Equally, for $(\psi_2\ \mathbf{U}^+\ \psi_1)$ and $\neg(\psi_2\ \mathbf{U}^+\ \psi_1)$, there are two tableau rules based on the fixed point unfolding of the $\mathbf{U}^+$-operator:

- $(\psi_2\ \mathbf{U}^+\ \psi_1) \leftrightarrow \mathbf{X}\ (\psi_1 \vee \psi_2 \wedge (\psi_2\ \mathbf{U}^+\ \psi_1))$

There is a close connection between the tableau decision procedure and ω-automata: The pre-states in the tableau can be seen as states of a generalised Büchi-automaton. The set of open leafs forms the acceptance condition, and the recurrence condition is given as follows: For every subformula $(\psi_2\ \mathbf{U}^+\ \psi_1)$, either it is infinitely often not contained in the accepting run, or ψ_1 is contained infinitely often. This can be formulated as generalised Büchi-acceptance condition.

4.3 Incompleteness Results

We have seen that propositional multimodal and temporal logics are complete and decidable. The same holds for **qTL** and μ**TL** on natural models. The relational μ-calculus, however, can categorically define addition and multiplication of natural numbers; therfore, it is highly undecidable. What about quantified temporal logics, interpreted on Kripke structures? In this section we will sketch an undecidability proof for this case.



Consider the above state transition diagram of a counter machine, which increments and decrements its counter with every *put* and *get* operation, respectively. We show how this machine can be coded by a finite set of formulas, such that every model of these formulas describes the sequence of memory states of a complete run.

Let the set of operators be $\{hi, lo, er, put, get, eoq, \langle X\rangle, \langle F\rangle, \langle M\rangle\}$. The operator $\langle X\rangle$ will be used to describe the execution steps of the program in time, the operator $\langle F\rangle$ to denote the transitive closure of $\langle X\rangle$, and the operator $\langle M\rangle$ to access the content of the memory. The following formulas describe that X and M are functional, form a half-grid, and that F is the transitive closure of X:

- $\forall q(\langle X\rangle\ q \rightarrow [X]\ q), \qquad \forall q(\langle M\rangle\ q \rightarrow [M]\ q)$
- $\forall q(\langle M\rangle\langle X\rangle\ q \rightarrow \langle X\rangle\langle M\rangle\ q)$
- $\forall q(\langle X\rangle\ q \vee\ \langle X\rangle\langle F\rangle\ q \rightarrow \langle F\rangle\ q)$,
  $\forall q([F]\ (p \rightarrow [X]\ p) \rightarrow ([X]\ q \rightarrow [F]\ q))$

Using these relations, we fix the propositions such that

— the number of M^*–successors in any world labelled *eoq* is the value of the counter,

— every world is labelled hi,lo, or er, according to the machine state it denotes, and

— every world is labelled put or get, according to which action is executed next.

The relevant formulas are:

- put increases the length of the counter by one:
  $(put \wedge [M]\ \mathbf{false} \rightarrow \langle X\rangle\langle M\rangle[M]\ \mathbf{false})$
- get decreases the length of the counter by one:
  $(\langle M\rangle\ (get \wedge [M]\ \mathbf{false}) \rightarrow \langle X\rangle[M]\ \mathbf{false})$
- Every world is exactly one of $\{put, get\}$ and $\{hi, lo, er\}$:
  $(put\ xor\ get) \wedge (hi\ xor\ lo\ xor\ er)$ (xor denoting exclusive disjunction)
- All worlds reachable by M^* have the same marking:
  $(P \rightarrow [M]\ P)$ for $P \in \{put, get, hi, lo, er\}$
- eoq propagates only in one dimension:
  $(eoq \rightarrow [X]\ eoq), \qquad [M]\ \neg eoq$
- Transitions:
  $(lo \wedge put \rightarrow [X]\ hi), \qquad (hi \wedge get \rightarrow [X]\ lo),$
  $(lo \wedge get \rightarrow [X]\ er), \qquad (hi \wedge put \rightarrow [X]\ er),$
  $(er \wedge get \rightarrow [X]\ er), \qquad (er \wedge put \rightarrow [X]\ \mathbf{false})$

For a conditional transition like "from er go to lo if counter is zero" we could use the sentence $(er \wedge eoq \wedge [M]\ \mathbf{false} \rightarrow [X]\ lo)$. For a multiple counter machine, we can use a similar encoding with several memory access functions $\langle M_i\rangle$. Now there is a computation in which such a machine reaches a certain state (say, hi) infinitely often from its initial state iff the sentence $(lo \wedge eoq \wedge [M]\ \mathbf{false} \wedge [F]\langle F\rangle\ hi)$ is satisfiable in a model validating all of the above axioms and sentences. Of course, for our example machine, we easily see that the formula is satisfiable; for all single counter machines this *recurrence problem* is decidable. But, for multiple counter machines the problem is Σ^1_1-complete, therefore also the problem whether any sentence follows from a set of universally quantified multimodal formulas is Σ^1_1-hard. Recall that such formulas are monadic Π^1_1-properties, so in this case the problem is in Σ^1_1 as well.

Chapter 5

Model Checking Examples

Model checking is an automatic technique for verifying correctness properties of safety-critical reactive systems. This method has been successfully applied to find subtle errors in industrial-size problems such as the design of sequential circuits, communication protocols and digital controllers. The technique was first suggested in [CE81, QS81]. However, then it was only possible to handle concurrent systems with a few thousand states. In the last few years the size of the concurrent systems that can be handled has increased dramatically. By using sophisticated data structures and heuristic search procedures, it is now possible to check systems many orders of magnitude larger [BCM+92, YS97]. It is expected that besides classical quality assurance measures such as static analysis and testing, model checking will become a standard procedure in the design of reactive systems.

Much of the success of model checking is due to the fact that it is an fully automatic verification method. Interactive methods are more general but harder to use; automatic methods have a limited range but are more likely to be accepted. In interactive verification, the user provides the overall proof strategy; the machine augments the user by

- checking the correctness of each step,
- maintaining a list of assumptions and subgoals,
- applying the rules and substitutions which the user indicates, and by
- searching for applicable transformation rules and assumptions.

Sophisticated tools also are able to prove certain lemmas automatically, usually by applying a heuristic search. Although there has been considerable

research on the use of theorem provers, term rewriting systems and proof checkers for verification, these techniques are time consuming and often require a great deal of manual intervention. Moreover, since most interactive provers are designed for undecidable languages (e.g., first or higher order logic), the proof process can never be completely automatic. User interaction is required, e.g., to find loop invariants or inductive hypotheses, and it requires an experienced user to perform a nontrivial proof.

On the other hand, with model checking all the user has to provide is a model of the system and a formulation of the property to be proven. The verification tool will either terminate with an answer indicating that the model satisfies the formula or give an execution showing that the formula fails to hold in the model. These counterexamples are particularly helpful in locating errors in the model or system.

With the completely automatic approach it may be necessary for the model checking algorithm to traverse all reachable states of the system. This is only possible if the state space is finite. However, many interesting systems like sequential circuits or network protocols are finite state. Moreover, in the design of safety critical systems it is often possible to separate the (finite state) control structure from the (infinite state) data structure of a given module. Finally, in many cases it is possible to *abstract* an infinite domain into an appropriate finite one, such that "interesting" properties are preserved.

The main disadvantage of the fully automatic approach is the state explosion: If any state of the system is uniquely described by n state bits, then there are 2^n possible states the system can be in. At the present time, the number of states that can be represented *explicitly* (e.g., by lists or hash tables) is approximately 10^6. In [BCM+92, McM93], *binary decision diagrams* (BDDs) were used to represent state spaces *symbolically*. With this technique, models with several hundred state bits and more than 10^{100} reachable states can be checked. Therefore it is possible to verify reactive systems of realistic industrial complexity, and a number of major companies including Intel, Motorola, ATT, Fujitsu and Siemens have started using symbolic model checkers to verify actual designs.

Given a formal model of a system to be verified, and a formulation of the properties the system should satisfy, there are three possible results which an automated model checker can produce:

1. either it finds a *proof* for the formula in the model and outputs "verified", or

2. it constructs a *refutation*, i.e., an execution of the (model of the) sys-

tem which dissatisfies the (formulation of the) property, or

3. the complexity of the verification procedure exceeds the given memory limit or time bound.

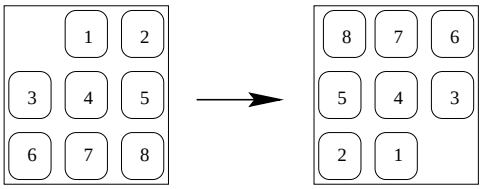
If there is not sufficient space or time, in some cases it is possible to use bigger and faster machines for verification. Alternatively, one can use a coarser abstraction of the system and its properties. The third possibility is to employ heuristics which improve the performance of the verifier. Some of these heuristics are discussed in subsequent chapters.

In some sense it is more interesting to get a refutation than to get a proof. With a refutation, one can decide whether it is due to the modelling and formulation, or whether this undesired sequence of events could indeed happen in reality. In the former case, the unrealistic behavior can be eliminated by additional assumptions on the model or formula. In the latter case, one has found a bug, and the system and model can be changed appropriately. One of the major advantages of the fully automatic approach is that there is almost no additional overhead for the new verification of the changed system.

If the model checker is able to prove all specified formulas for the given model, then the verification is successfully completed. However, there can never be any guarantee that a system which has been verified by a computer aided tool will function correctly in reality. Even if we could assume that the verifier's hard- and software is correct (which we can not), there is a fundamental source of inaccuracy involved. Verification proves theorems about models of systems and formulations of properties, not about physical systems and desired behavior; we can never know to what extent our models and formulations reflect physical reality and intuitions. It is not possible to guarantee that a physical system will behave correcly in unexpected (i.e., unmodeled) situations. It would be unreasonable, however, to reject formal methods because they cannot offer such guarantees. Civil engineering can never *prove* that a certain building will not collapse. Nevertheless it uses mathematical models to calculate loads and wall thicknesses and so on. Similarly, we can never *prove* that our model adequately represents the reality. Therefore we can never *prove* that a system will function as planned. Nevertheless, computer aided verification can help *locating errors* during the design phase of a complex system, and it can help to *increase reliability* of these systems. In the future, formal verification by model checking will augment classical software design tools like structured analysis, code review and testing.

5.1 A Combinatorial Game

As a first example, we describe the use of model checking in a combinatorial search. Although this example is not very typical for real applications, it can demonstrate the capabilities and limits of present technology. A well-known puzzle from 1870 by the American Sam Loyd consists of a $h \times v$ grid in which there are $(h \cdot v) - 1$ numbered tiles and one blank space. A move consists in moving any tile into the position of the blank. The goal is to achieve a certain predetermined order on the tiles.



This puzzle can be described by a shared variables program as follows. For each tile there is a program variable which notes its horizontal and vertical position. Furthermore, there is a program variable `move` indicating whether the next move will be a shift up, down, left or right of the blank space. If the move would bring it out of the borders, nothing is changed; otherwise, its position is swapped with the respective adjacent tile.

The SMV code corresponding to this description[1] is shown in Figure 5.1. For $h = 3$ and $v = 3$, the internal representation of the transition relation takes about 3KB. There are $4 \cdot (h \cdot v)! = 1.4 \cdot 10^6$ states, of which 50% are reachable from any initial state. The specification claims that a certain final state is *not* reachable; the model checker contradicts this claim by showing a sequence of moves (rrddlluurrddlluurrddlluurrdd) which gives a solution to the puzzle. The solution is found within a couple of minutes on a 32 MB Pentium 133.

For $h = 4$, $v = 3$, there are approximately 10^9 reachable states. Although the model checker detects rather quickly that some solution must exist, for the construction of a concrete solution sequence the state space has to be partitioned into strongly connected components. This requires several days of CPU time and approximately 1GB RAM on a Sparc Ultra. For model checking applications, virtual memory is not very useful; if the representation of the reachable state space exceeds the available main memory, then constant swapping occurs. To find a solution for $h = 4$, $v = 4$ by exhaustive

[1] In the actual SMV code, variable array bounds or indices, e.g., vpos[i], are not allowed and have to be replaced by the respective constant values vpos[1],vpos[2],...

```
MODULE main
DEFINE h := 3; v := 3;
VAR move:  u,d,l,r;
    hpos:  array 0..(h*v-1) of 1..h;
    vpos:  array 0..(h*v-1) of 1..v;
ASSIGN
next(vpos[0]) := case
  (move=l) & !(vpos[0]=1) :  vpos[0] - 1;
  (move=r) & !(vpos[0]=v) :  vpos[0] + 1;
  1:  vpos[0]; esac;
next(hpos[0]) := case
  (move=u) & !(hpos[0]=1) :  hpos[0] - 1;
  (move=d) & !(hpos[0]=h) :  hpos[0] + 1;
  1:  hpos[0]; esac;
for all i:
next(vpos[i]) := case
  (move=l) & !(vpos[0]=1) & hpos[i]=hpos[0] & vpos[i]=vpos[0]+1 |
  (move=r) & !(vpos[0]=v) & hpos[i]=hpos[0] & vpos[i]=vpos[0]-1 :  vpos[0];
  1:  vpos[i]; esac;
next(hpos[i]) := case
  (move=u) & !(hpos[0]=1) & vpos[i]=vpos[0] & hpos[i]=hpos[0]-1 |
  (move=d) & !(hpos[0]=h) & vpos[i]=vpos[0] & hpos[i]=hpos[0]+1 :  hpos[0];
  1:  hpos[i]; esac;
init(hpos[i]) := i div v; init(vpos[i]) := i mod v;
DEFINE goal := (hpos[i] = 3 - (i div v) & vpos[i] = 3 - (i mod v))
SPEC !EF goal
```

Figure 5.1: SMV Code for Loyds Puzzle

state space exploration seems to be beyond the limits of present technology. In [ER98], a combination of model checking and heuristic search is used to automatically construct solutions to this and other combinatorial games.

5.2 A Sequential Circuit

Our second example is from hardware verification. We consider a shift register for interfacing a parallel data bus. The register is from the 74x95 TTL family and is described in [NA90]. It is used to exchange data between the bus and a serial device. It thus acts as parallel-serial converter and vice versa. A functional diagram of the register is given in Figure 5.2.

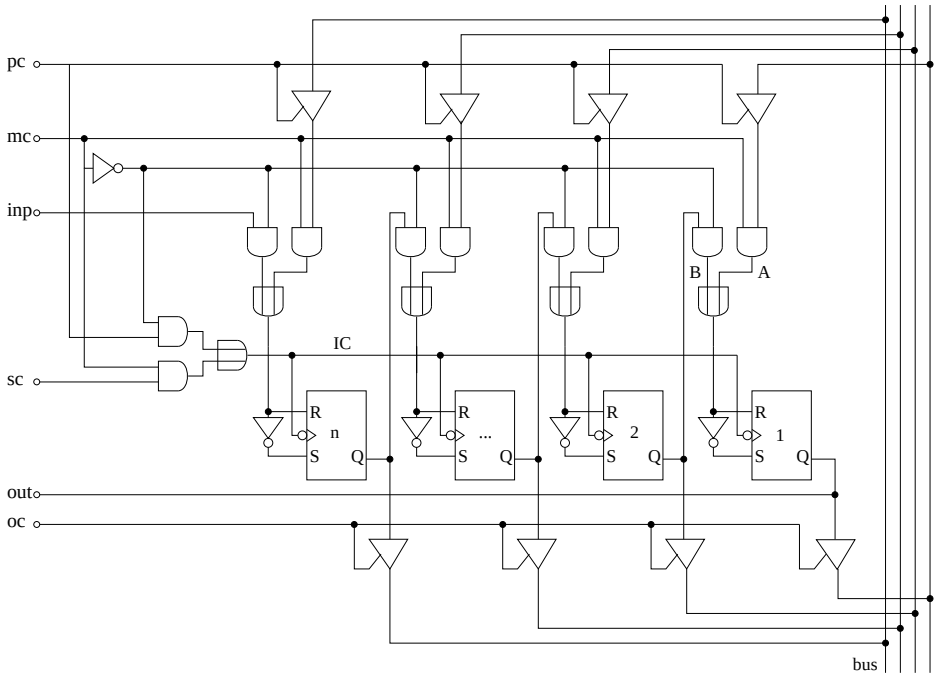


Figure 5.2: A shift register for data bus interfacing

The register has a *mode control* input `mc` to choose between parallel or serial access mode. For each mode, there is a corresponding input clock (`pc` and `sc`). Parallel loading is performed if `mc` is high and a `pc` clock pulse arrives. In this case, data is read from the bus into the associated flip-flops. The data appears at the `Q` outputs at the pulse of the `pc` clock.

For serial loading, mode control should be low. Data is input serially with every tick of the `sc` clock. At each pulse the state of all flip-flops is transferred one stage to the right. After n cycles, the data is positioned at the parallel output and can be sent to the bus by an `oc` command. A right shift occurs if the serial input `inp` is held low. By a sequence of n right shifts, data which has been obtained in parallel from the bus can be written serially to the `out` port.

The register is implemented with SR-bistables which have the following characterictic function. If both inputs are low, the bistable keeps its state.

S	R	Q'
0	0	Q
1	0	1
0	1	0
1	1	-

The output `Q` is set if input `S` is high, and reset if input `R` is high. If both `S` and `R` are high, then `Q` is undefined. This can be modelled by a nondeterministic internal choice between high and low output. The latch is triggered by a negative edge of the clock pulse. That is, a change of output occurs only at the time instant when the clock line goes from high to low. If the value of the clock line is part of the state space, then the clock value would be low in every new state. For an accurate state-based model (e.g., of an asynchronous circuit), we would have to include timing information of all gates. However, if the clock is only used as trigger, an event based modelling is more adequate: The high-to-low change of the clock line is considered as an event occurrence. In each state, this event may or may not occur. To prevent executions in which the input or output clocks are indefinitely blocked, we require infinitely many input and output clock ticks in every infinite run.

The model is just a representation of the circuit's truth table, where the outputs are a boolean function of inputs and latch states. It can be derived automatically from any standard hardware description language; in fact, several model checkers support such front-end translations. Correctness of parallel and sequential input is expressed by the following formulas:

$$\mathbf{A}\ \mathbf{G}^{*}\ (\mathtt{mc} \wedge \mathtt{pc} \rightarrow (\mathtt{bus[i]} \leftrightarrow \mathbf{A}((\mathtt{oc} \rightarrow \mathbf{A}\ \mathbf{X}\ \mathtt{bus[i]})\ \mathbf{U}^{+}\ \mathtt{ic})))$$

$$\mathbf{A}\ \mathbf{G}^{*}\ (\neg\mathtt{mc} \wedge \mathtt{sc} \rightarrow (\mathtt{Q[i]} \leftrightarrow \mathbf{A}(\mathtt{Q[i\text{-}1]}\ \mathbf{U}^{+}\ \mathtt{ic})))$$

```
MODULE main
VAR Q, bus:  array 1..n of boolean;  -- n SR-latches, n databits
    inp, mc, pc, sc, oc:  boolean;   -- input lines
DEFINE out := Q[1]; ic := ((mc & pc)  |  (!mc & sc));
       A[i] := mc & pc & bus[i]; B[i] := !mc & Q[i + 1];
       R[i] := !(A[i] | B[i]); S[i] := !R[i];
ASSIGN next(Q[i]) := case ic:  case
                                !S[i] & !R[i]:  Q[i];        --hold
                                 S[i] & !R[i]:  1;           --set
                                !S[i] &  R[i]:  0;           --reset
                                 S[i] &  R[i]:  {0,1}; esac; --undef
                          !ic:  Q[i]; esac; -- unchanged if no input
       next(bus[i]) := case oc:  Q[i];  !oc:  {0, 1}; esac;
FAIRNESS ic FAIRNESS oc
```

Figure 5.3: Model of shift register

Intuitively, these formulas assure that data which is input into the register remains there until a new input occurs. With appropriate ordering on the BDD variables, the model checker verifies these formulas for a bus width of 32 bit in less than a second. Similar formulas can be used to verify that after a sequence of n sequential load operations, the correct data word will be put onto the bus on a subsequent output pulse.

If the connection structure of wires within the circuit is "well-behaved", then automatic verification is successful even on much bigger circuits. A circuit is "well-behaved" if there exists an ordering of all wires such that the value of a wire only depends on the value of wires which are close in the ordering. For a formal definition of this condition see [McM93]. A large number of circuits with hundreds of storage places have been verified automatically in this way.

5.3 A Communication Protocol

The third example is a set of communicating processes within the operating system of a Siemens cellular phone. In this system, there are a number of basic processes communicating with one another by priority messages. Each of the processes implements a finite state machine, which is described by a set of SDL diagrams. Basically, a process waits in a certain state until it receives a message from some other process. It then performs some specified

operations, sends a number of messages to other messages, and transitions to another state. Figure 5.4 shows part of the transition graph of a process and the corresponding SDL diagram. The displayed part is used to implement the following quote from the GSM international standard.

> *"Initially the MS looks for a cell which satisfies the suitability constraints by checking cells in descending order of received signal strength. If a suitable cell is found, the MS camps on it and performs any registration necessary."*

A property to be verified is that the system never deadlocks:

$$\mathbf{A}\ \mathbf{G}^*\ \mathbf{E}\ \mathbf{F}^*\ \texttt{init}$$

That is, we proved that no sequence of user actions can bring the phone into a state from where it cannot be reset. Since the number of merchandized units is expected to be very high, correctness is an important design issue.

In the model to be checked, there are five basic processes, plus the operating system kernel. There are approximately 50 different types of messages which can be sent by the processes, and each process has between 10 and 20 states. The operating system is responsible for the scheduling of processes according to a priority scheme, and for the storage and delivery of messages. Therefore, it has to maintain a buffer, in which for each process all messages are kept. The size of these buffers turns out to be the most important parameter in the verification. Basically, each buffer slot could be filled with every message; thus a combinatorical explosion similar to the one in our first example can occur. However, a buffer overflow almost certainly indicates an error in the implementation; for example, if some high-priority process keeps resending the same message, it will eventually fill up any bounded buffer. In the modelled system, a total number of 15-20 buffer slots was sufficient; a fairness assumption is used to select only those computations in which no buffer overflow occurs. Moreover, the buffer contents usually follows a regular pattern, therefore the above mentioned state explosion is avoided. In practical applications, an exponential growth in the number of reachable states almost certainly indicates an error. For buffers in which all messages have the same priority, the transition relation of a bounded buffer can be defined by the transition table in Figure 5.5.

In the right half of this table, an empty entry means that the respective program variable is set by the environment. An input value of *nil* in i indicates that there is no message to be sent; in this case the next value of i is determined by the sender. If this process has put a non-*nil* value x into

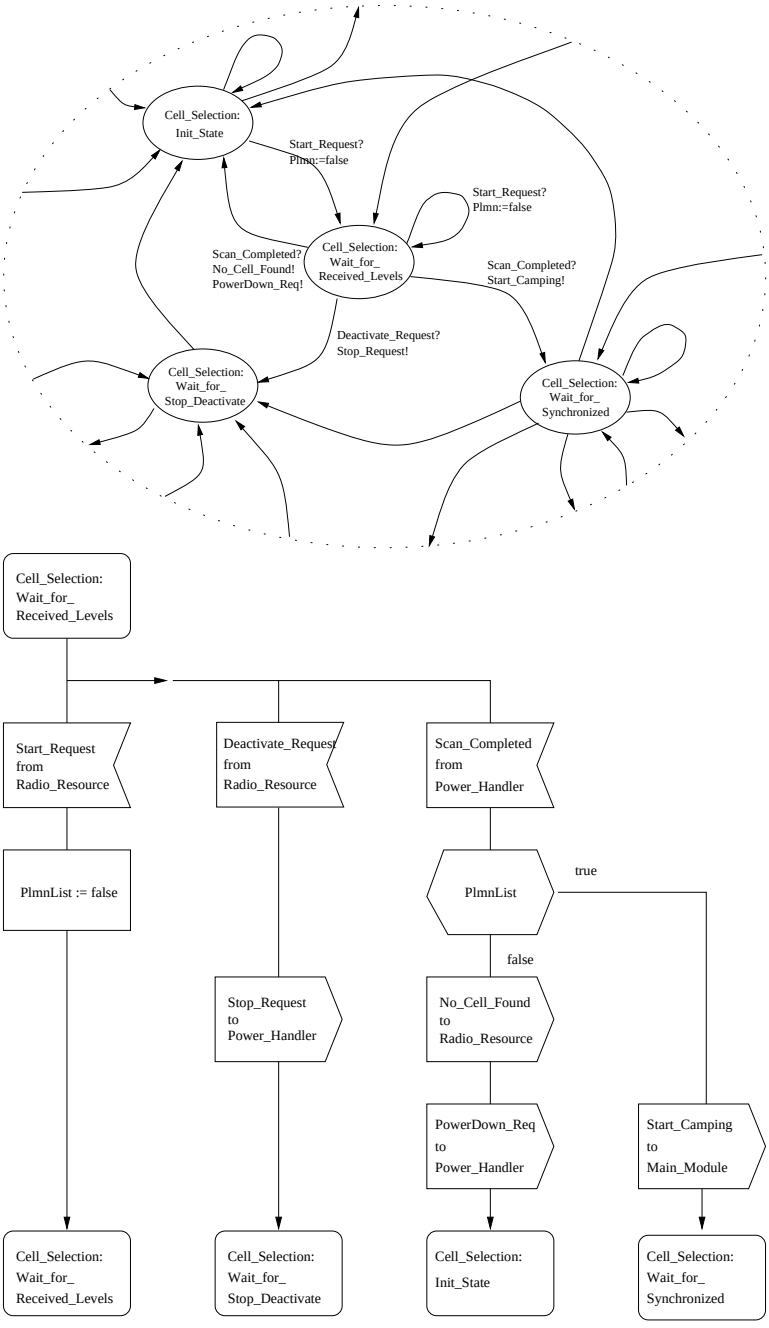


Figure 5.4: Transition graph and SDL diagram

i	b	o		i'	b'	o'
nil	$\langle\rangle$	nil			$\langle\rangle$	nil
x	$\langle\rangle$	nil		nil	$\langle\rangle$	x
nil	$\langle x_1, ..., x_\nu \rangle$	nil			$\langle x_1, ..., x_{\nu-1} \rangle$	x_ν
x	$\langle x_1, ..., x_\nu \rangle$	nil		nil	$\langle x, x_1, ..., x_{\nu-1} \rangle$	x_ν
nil	$\langle\rangle$	y			$\langle\rangle$	
x	$\langle\rangle$	y		nil	$\langle x \rangle$	
nil	$\langle x_1, ..., x_\nu \rangle$	y			$\langle x_1, ..., x_\nu \rangle$	
x	$\langle x_1, ..., x_\nu \rangle$	y	$(\nu < n)$	nil	$\langle x, x_1, ..., x_\nu \rangle$	
x	$\langle x_1, ..., x_n \rangle$	y		x	$\langle x_1, ..., x_n \rangle$	

Figure 5.5: Transition relation of a bounded buffer

i, then this value is appended to the buffer, and i is reset to nil. The last line indicates a buffer overflow: If a message is to be sent with the message buffer already filled, i remains stable. Thus, the formula $\mathbf{A}\ \mathbf{G}^*\ (i \neq nil \rightarrow \mathbf{X}\ (i = nil))$ can be used to determine whether a buffer overflow can occur. If the output variable o is nil and there is a message to deliver, it is copied into o. When the operating system delivers a message y from o, it resets o to nil.

The content of the buffer b is given as a sequence $\langle x_1, ..., x_\nu \rangle$ of messages, where $\langle\rangle$ denotes the empty buffer. There are various possibilities to model such sequences. In Figure 5.6 we show a modelling which uses n program variables $b_1, ..., b_n$, such that b_1 contains the front element of the message queue, and incoming messages are appended into the smallest b_ν which is empty (contains nil as value).

```
next(b[j]) := case
   (i=nil) & !(o=nil) :  b[j];
   (i=nil) &  (o=nil) :  b[j+1];
  !(i=nil) & !(o=nil) :  if !(b[j-1]=nil) & b[j]=nil then i
                         else b[j] fi;
  !(i=nil) &  (o=nil) :  if b[j]=nil then nil
                         else if b[j+1]=nil then i
                         else b[j+1] fi fi;            esac;
```

Figure 5.6: Model of bounded buffer

In this modelling, we rely on the fact that whenever $b_j = nil$, then for all $k \geq j$, also $b_k = nil$. This assumption only holds for the reachable states of a buffer which is initially empty; there are many transitions from illegal, i.e., nonreachable states to other illegal states in this model. In an explicit representation of the transition relation, one should try to avoid these redundant entrys. Below, we discuss symbolic representations with BDDs. With such a representation, even though the size of the transition relation is much bigger than the transition relation restricted to the reachable states, its representation is much smaller. Since the value of each buffer slot depends only on its immediate neighbours, in fact the size of the representation is linear in the number and width of the buffer slots.

Chapter 6

Model Checking Algorithms

Given a model $\mathcal{M}$ and a formula φ, the model checking problem is to decide whether $\mathcal{M} \models \varphi$. In principle, this can be done by encoding $\mathcal{M}$ as a set of assumptions (or premisses, or program axioms) Φ, and deciding whether $\Phi \vdash \varphi$. However, some experiments will quickly convince the reader that a naïve approach of doing so is doomed to failure. Usually, the program axioms all have a very special form, such as

$$(\texttt{state\_i} \rightarrow (\mathbf{X}\ \texttt{succ\_i1} \vee \cdots \vee \mathbf{X}\ \texttt{succ\_in}))$$

in a linear time modelling, or

$$(\texttt{state\_i} \rightarrow (\langle a_1 \rangle\ \texttt{succ\_i1} \wedge \cdots \wedge \langle a_n \rangle\ \texttt{succ\_in}))$$

in a branching time approach. The decision procedure in general can not take advantage of this special form of the assumptions and will in every step break down all assumptions to its basic propositional components. This results in a very inefficient behavior; usually only very small systems can be verified and debugged that way.

Therefore, model checking algorithms avoid the encoding of the models as a set of program axioms; they use the models directly instead. Model checking determines whether a given specification formula is satisfied in a given Kripke-model, i.e., whether a tree or natural model satisfying the formula can be generated from it.

There are two variants of this task, depending on whether the initial or universal definition of satisfaction of a formula in a model is used. In the usual definition, a Kripke-model, consisting of universe U, accessibility relation(s) defined by $\mathcal{I}$, and current point $w_0 \in U$, is given, and we have to check whether the formula φ is satisfied: $(U, \mathcal{I}, w_0) \models \varphi$. In the universal definition, we are given universe and interpretation, and want to know

whether the formula is satisfied in *all* points w of the universe: $(U,\mathcal{I}) \models \varphi$ iff for all $w_0 \in U$ it holds that $(U,\mathcal{I},w_0) \models \varphi$. Equivalently, we want to know whether $\varphi^{\mathcal{M}} = U$, where $\varphi^{\mathcal{M}} \triangleq \{w \in U \mid w \models \varphi\}$ is the set of points satisfying φ.

Of course, any algorithm which calculates $\varphi^{\mathcal{M}}$ can also be used to decide whether $(U,\mathcal{I},w_0) \models \varphi$ holds: $w_0 \models \varphi$ iff $w_0 \in \varphi^{\mathcal{M}}$. Vice versa, if we have an efficient algorithm to decide whether $w_0 \models \varphi$, we can calculate $\varphi^{\mathcal{M}}$ by an iteration on all states.

Model checking has two parameters: model $\mathcal{M}$ and formula φ. Algorithms, which iterate on the structure of φ and in each step traverse the whole of $\mathcal{M}$ are sometimes called *global*. Algorithms, which iteratively extend the checked part of $\mathcal{M}$ and in each step determine the truth of each sub-formula of φ are sometimes called *local*. Although the theoretical worst-time complexity is not influenced by this choice, the average case behavior may differ significantly.

In principle, the three axes (branching/linear, universal/initial, global/local) are independent. In practice, however, for branching time logics mostly global algorithms and universal validity is used, whereas with linear time logics local algorithms for initial validity have been suggested.

6.1 Global Branching Time Model Checking

Given a Kripke-model $\mathcal{M} = (U,\mathcal{I})$ and a multimodal formula φ, the set $\varphi^{\mathcal{M}} \triangleq \{w \in U \mid w \models \varphi\}$ of points validating φ can be calculated by a recursive descent on the structure of φ. If $\mathtt{p}$ is an atomic proposition, then $\mathtt{p}^{\mathcal{M}} \triangleq \mathcal{I}(\mathtt{p})$. Furthermore, $\bot^{\mathcal{M}} \triangleq \{\}$ and $(\varphi \to \psi)^{\mathcal{M}} \triangleq U \setminus \varphi^{\mathcal{M}} \cup \psi^{\mathcal{M}}$. Finally, $(\langle R\rangle\ \psi)^{\mathcal{M}} \triangleq \{w \in U \mid \exists w' \in \psi^{\mathcal{M}}, (w,w') \in \mathcal{I}(R)\}$.

This algorithms seems to be just a trivial reformulation of the semantical definition for the logical operators. However, there are some important observations. Firstly, $(\langle R\rangle\ \psi)^{\mathcal{M}}$ can be calculated from $\psi^{\mathcal{M}}$ in two ways: We can either check for each $w \in U$, whether the intersection of $\psi^{\mathcal{M}}$ and $R(w)$ is nonempty. Alternatively, we can calculate $\bigcup\{R^{-1}(w') \mid w' \in \psi^{\mathcal{M}}\}$, where $R^{-1}(w') \triangleq \{w \mid (w,w') \in \mathcal{I}(R)\}$ is the *inverse image* of point w under the relation R. This inverse image calculation can be accomplished by a traversal of all arcs $(w,w') \in \mathcal{I}(R)$: If $w' \in \psi^{\mathcal{M}}$, then $w \in (\langle R\rangle\ \psi)^{\mathcal{M}}$. Secondly, to avoid recalculation of common subformulas, we use a table, where for each sub-formula ψ the set $\psi^{\mathcal{M}}$ is stored. Thus, we need an efficient data structure for large sets of points. Thirdly, the overall *complexity* of this algorithm is linear in the number of different sub-formulas and in the

size of the model. However, even for infinite models which are given by some symbolic description (e.g., Petri nets or Turing machines), the model checking problem can be decidable [BE97].

Similar to the above modal logic procedure, the **CTL** model checking algorithm proceeds by marking each point with the set of sub-formulas which are valid for this point. Suppose we have already marked the set of points satisfying ψ_1 and the points satisfying ψ_2. To label the set of points satisfying $\varphi \triangleq \mathbf{E}(\psi_2\ \mathbf{U}^+\ \psi_1)$ or $\varphi \triangleq \mathbf{A}(\psi_2\ \mathbf{U}^+\ \psi_1)$, we use the fixpoint unfoldings

$$\mathbf{E}(\psi_2\ \mathbf{U}^+\ \psi_1) \leftrightarrow \mathbf{E}\ \mathbf{X}\ (\psi_1 \vee \psi_2 \wedge \mathbf{E}(\psi_2\ \mathbf{U}^+\ \psi_1))$$

$$\mathbf{A}(\psi_2\ \mathbf{U}^+\ \psi_1) \leftrightarrow \mathbf{A}\ \mathbf{X}\ (\psi_1 \vee \psi_2 \wedge \mathbf{A}(\psi_2\ \mathbf{U}^+\ \psi_1))$$

For $\varphi \triangleq \mathbf{E}(\psi_2\ \mathbf{U}^+\ \psi_1)$, we label all points with φ which have a successor that is labelled with ψ_1, or with ψ_2 and also φ. This process is repeated until stabilisation is reached. For $\varphi \triangleq \mathbf{A}(\psi_2\ \mathbf{U}^+\ \psi_1)$, we label all points with φ for which all successors are labelled with ψ_1, or with ψ_2 and also φ. Again, this process must be repeated until no new points can be marked. A recursive formulation of this algorithm is given below.

Since the Kripke-model has a finite number of points, each **repeat** stabilises after at most $|U|$ passes. In the worst case, each pass searches the whole model, hence the complexity is linear in the number of different subformulas, and cubic in $|U|$.

This bound can be improved if the search is organised better. In [CES86], an algorithm is given which is linear in the size of the model as well. For the $\mathbf{E}\ \mathbf{F}^+$-operator, the problem of marking all points for which $\mathbf{E}\ \mathbf{F}^+\ \varphi$ holds, given the set of point satisfying φ, is equivalent to the *inverse reachability problem*: Given a set of points, mark all points from which any finite path leads into the given set. Assuming that for any two points we can decide in constant time whether they are connected by an arc, this can be done with time complexity quadratic in the number of points.

Every point enters the set *Search* in the **while** loop at most once. Moreover, all set operations can be performed in time linear in the size of these sets, i.e., in the number of points; thus the overall complexity is quadratic in $|U|$ or linear in the size of the Kripke-model.

For the $\mathbf{E}\ \mathbf{U}^+$-operator, this idea can be refined to give an evaluation procedure of linear complexity. The $\mathbf{A}\ \mathbf{U}^+$-operator can be expressed by

$$\mathbf{A}(\psi_2\ \mathbf{U}^+\ \psi_1) \leftrightarrow \neg(\mathbf{E}(\neg\psi_1\ \mathbf{U}^+\ (\neg(\psi_1 \wedge \psi_2)) \vee \mathbf{E}\ \mathbf{G}^+\ \neg\psi_1)$$

Thus, we only need a procedure marking all points for which $\mathbf{E}\ \mathbf{G}^+\ \varphi$ holds. This can be done as follows:

```
program CTL_check (Model (U, I, w_0), Formula φ) =
    if w_0 ∈ eval(φ)
    then print("φ is satisfied at w_0 in (U, I)")
    else print("φ not satisfied at w_0 in (U, I)");
procedure eval (Formula φ): Pointset =
    case φ of
        p : return I(p);
        ⊥ : return {};
        (ψ_1 → ψ_2) : return U \ eval(ψ_1) ∪ eval(ψ_2);
        E(ψ_2 U^+ ψ_1) : E1 := eval(ψ_1); E2 := eval(ψ_2); E := {};
            repeat until stabilization
                E := E ∪ {w | (succ(w) ∩ (E1 ∪ E2 ∩ E)) ≠ {}};
            return E;
        A(ψ_2 U^+ ψ_1) : E1 := eval(ψ_1); E2 := eval(ψ_2); E := {};
            repeat until stabilization
                E := E ∪ {w | succ(w) ⊆ E1 ∪ E2 ∩ E};
            return E;
procedure succ (Point w): Pointset = return {w' | (w, w') ∈ I(≺)};
```

Figure 6.1: naïve **CTL** model checking algorithm

- restrict the model to those states satisfying φ
- find the maximal strongly connected components in the restriction
- mark all points in the original model from which a nontrivial SCC or a point without successors can be reached by a path in the restricted model.

These operations can be accomplished with time complexity which is quadratic in U. Thus, the overall complexity of CTL model checking is linear in the size of the formula and in the size of the model.

Fairness Constraints

Some automated model checkers for **CTL** allow to specify a set of *constraints* Φ together with the Kripke-model. These constraints are assumed to hold in the whole model; i.e., they restrict the model to those parts where they are valid. This use of constraints is somewhat different from the assumptions

```
procedure reach (Pointset Target): Pointset =
    Source := {}; Search := Target;
    while Search ≠ {} do
        Search := pred (Search) \ Source;
        Source := Source ∪ Search
    enddo;
    return Source;
procedure pred (Point w): Pointset = return {w' | (w', w) ∈ I(≺)};
```

Figure 6.2: Inverse reachability calculation

in the previous sections, which were used to constrain the *set* of possible models. For example, an ω-automaton can be regarded as a Kripke-model, together with global eventuality and fairness constraints (accepting and recurring states). Constraints can be formulated in the same language in which the formula to be checked is specified; however, "mixed" approaches have been suggested [Jos93], where e.g. the constraints are described in **LTL** and the property is described in **CTL**.

As an example for the use of such constraints, often the path-quantifiers **A** and **E** are restricted to *fair* paths. *Simple* fairness constraints are of form $\mathbf{F}^+ \psi$, where ψ is a boolean combination of propositions. For example, the condition $\mathbf{F}^+ \top$ specifies that each run must be infinite. As another example for a simple fairness constraint, we might want to restrict our attention to execution sequences in which every component is always eventually scheduled. Streett fairness constraints are of form $(\mathbf{G}^+\mathbf{F}^+ \psi_1 \rightarrow \mathbf{G}^+\mathbf{F}^+ \psi_2)$ and are useful to restrict attention to *strongly fair* schedulers: if a component infinitely often requests a resource, it will be granted infinitely often. The above algorithm can be modified to deal with such fairness constraints by building the tableau of the **LTL**-assumption and checking the **CTL**-formula on the product of Kripke-model and tableau. The complexity increases by a factor which depends on the type of **LTL**-formulas in the assumption.

6.2 Local Linear Time Model Checking

For a given Kripke-model $\mathcal{M} = (U, \mathcal{I}, w_0)$ and **CTL**-formula φ, the relation $\mathcal{M} \models \varphi$ holds iff the maximal tree generated from $\mathcal{M}$ at w_0 satisfies φ. For linear time logics, $\mathcal{M} \models \varphi$ is interpreted by *sequence-validity*. That is,

we want to check whether *every maximal sequence* generated from $\mathcal{M}$ at w_0 satisfies φ. Equivalently, we have to decide whether $\neg\varphi$ is satisfiable in *some* natural model generated from $\mathcal{M}$. In some sense, this is a more complex question than the one for branching time, because a whole *set* of natural models has to be checked. Hence, we cannot simply mark a point in the Kripke-model with the set of linear-time formulas which are valid for this point: for example, $\mathbf{F}^+ \psi$ can both be true for one of the generated sequences, and false for another one.

Similar to above, we first consider sequence-validity of modal logic with a single accessibility relation R. Given a Kripke-model $\mathcal{M} = (U, \mathcal{I}, w_0)$ and a modal formula φ, we want to determine whether there is a maximal sequence generated from $\mathcal{M}$ at w_0 which satisfies φ in w_0. This is done by a depth-first-search in the product of the set of propositionally maximal consistent sets of sub-formulas and the set of points in the model.

Formally, an *atom* α is any pair (w, m), where $w \in U$ is a point, and $m \subseteq SF(\varphi)$ is a propositionally consistent set of sub-formulas. An atom is *admissible*, if w and m agree on the interpretation of propositions. That is, if $\mathtt{p} \in SF(\varphi)$, then $\mathtt{p} \in m$ iff $w \in \mathcal{I}(\mathtt{p})$.

An *initial atom* is any admissible atom $\alpha = (w_0, m_0)$, where w_0 is the current point of $\mathcal{M}$, and $\varphi \in m_0$. For each $R \in \mathcal{R}$, we define a relation X_R between admissible atoms: $X_R((w, m), (w', m'))$ iff

1. $(w, w') \in \mathcal{I}(R)$,
2. if $\langle R \rangle\, \psi \in SF(\varphi)$ and $\psi \in m'$, then $\langle R \rangle\, \psi \in m$,
3. if $\langle R \rangle\, \psi \in m$, then $\psi \in m'$,
4. some $\langle R \rangle\, \psi \in m$.

The first condition reflects the fact that the steps in the generated sequence are predetermined by the Kripke-model (system to be verified). The second condition is imposed by the semantics of the $\langle R \rangle$-operator; the third condition is a reformulations of the axiom **(U)** and the corresponding tableau rule ($\langle R \rangle$). The fourth condition corresponds to the tableau rule ($[R]$); it allows the generated sequence to be finite when no $\langle R \rangle\, \psi$ is contained in a node.

Now we can construct a forest of atoms as follows:

- initial nodes are all initial atoms
- any node α has as children all α' such that $X_R(\alpha, \alpha')$

Since for any finite Kripke-model there are only finitely many atoms, each branch in this forest can be made finite by appropriate backward arcs. As in the tableau definition, a leaf is called *open*, if it has no $\langle R \rangle$ formulas in its m-component; otherwise, it is *closed*.

An *accepting path* through the resulting structure starts with any initial node and is either infinite or ends with an open leaf. Any accepting path is a sequence generated from the Kripke-model which satisfies the given formula $\neg\varphi$, thereby forming a counterexample to the specification φ.

To implement the search for an accepting path, we perform a depth-first search with backtracking from the set of initial atoms to all of its X-successors. In order to be able to terminate loops in this search, we have to store all atoms which were encountered previously. Though there are several possibilities to represent such a set of atoms, the method of choice seems to be to employ a hash table. It is not necessary to use all components of m as hash indices, since the value of propositions is determined by w, and boolean combinations of formulas can be recovered from their constituent parts. Therefore, it is sufficient to store only the value of $\langle R \rangle$-subformulas.

In general, since we are only looking for *some* counter-model, we can terminate the search if a counter-model is found. Although in the worst case (if no counter-model exists) the whole forest must be searched, it is possible to find errors very quickly by an appropriate ordering of the depth-first search successors.

In the depth-first search, we have to remove closed atoms from the list of possible loop points. A better way is to *mark* these nodes as closed while backtracking; then the search will not recurse again if such an atom reappears. Also all other improvements mentioned above can be used for this algorithm.

Extensions for LTL

We have seen that the local model checking algorithm for multimodal logic is almost the same algorithm as the local tableau decision procedure. Similarly, the local model checking for **LTL** is very close to its respective satisfiability algorithm.

In the definition of $X_R((w,m),(w',m'))$, we replace $\langle R \rangle$ by $\mathbf{X}$ and require additionally

5. if $\mathbf{F}^* \psi \in SF(\varphi)$ then $\mathbf{F}^* \psi \in m$ iff $\psi \in m$ or $\mathbf{XF}^* \psi \in m$

This requirement corresponds to the recursion axiom $\vdash \mathbf{F}^* \psi \leftrightarrow \psi \vee \mathbf{XF}^* \psi$. As in the case of modal logic, we try to thread an accepting path through

the graph of atoms which arises from this definition. However, we can only accept those paths in which all eventualities $\mathbf{F}^* \psi$ are fulfilled. Since we can not guarantee that several eventualities are simultaneously fulfilled in some single loop, we have to calculate the strongly connected components of the reflexive transitive closure of X_R. An SCC W of atoms is called *self-fulfilling*, if for any $\mathbf{F}^* \psi$ in some $\alpha \in W$ there exists some $\alpha' \in W$ with $\psi \in \alpha'$. Any atom which does not contain positive future obligations $\mathbf{X} \psi$ is a trivial SCC, because it is a terminal node in the atom graph. Such a node forms a self-fulfilling SCC, because the above condition (5.) guarantees that for any $\mathbf{F}^* \psi \in \alpha$, also $\psi \in \alpha$. The given formula φ is satisfiable in $\mathcal{M}$ iff there exists a self-fulfilling SCC which is reachable from some initial atom. In this case, a natural model for φ generated by $\mathcal{M}$ is given by any sequence of atoms from an initial atom which ends in terminal atom or infinitely often passes through all atoms of a self-fulfilling SCC.

For $\mathbf{U}^+$-operators, each positive occurrence $(\psi_1 \mathbf{U}^+ \psi_2)$ is an eventuality which has to be fulfilled at some point; thus the SCC W is defined to be self-fulfilling, if it is nontrivial and for any $(\psi_1 \mathbf{U}^+ \psi_2)$ in some $\alpha \in W$ there exists some $\alpha' \in W$ with $\psi_2 \in \alpha'$, or it is trivial and does not contain any $(\psi_1 \mathbf{U}^+ \psi_2)$.

How can we construct maximal SCCs, and decide whether they are self-fulfilling? There are two different algorithms known in the literature. For model checking, Tarjan's algorithm [Tar72] is particularly well-suited, since it enumerates the strong components of a graph during the backtrack from the depth-first search. Thus model checking can be performed "on-the-fly" during the enumeration of the reachable atoms of the model.

In an implementation of this algorithm, atoms can be represented by bitstrings which contain one bit for each proposition $\mathtt{p} \in \mathcal{P}$ and one bit for each sub-formula $(\psi_2 \mathbf{U}^+ \psi_1) \in SF(\varphi)$. The function `children` constructs for a given atom α the set of all possible successor atoms according to the transition relation of the Kripke-model and to the fixed point definition of the until-operator.

The procedure `depth_first_search` recursively builds all atoms reachable from a given atom α. When the procedure backtracks, α is the root of a maximal SCC iff there are no atoms β in the subtree below α such that α is also in the subtree of β. In this case, the maximal SCC containing α consists of all nodes in the subtree below α, and this maximal SCC can be checked for acceptance. *table* is implemented as a hash table from atoms to natural numbers. $table[\alpha]$ contains

- UNDEFINED, as long as atom α has not occurred,

```
program LTL_check (Model M, Formula φ) =
    Nat depth_first_count := 0;    /* number of recursive call */
    Atomset stack := {};    /* Stack of searched atoms */
    Natarray table;    /* Hashtable from atoms to natural numbers */
    Atomset init := {α | α is an initial atom of M and φ};
    for all α ∈ init do depth_first_search(α);
    print("φ is not satisfiable in M");

procedure depth_first_search (Atom α) =
    if (table [α] = UNDEFINED) then    /* α is a new atom */
        Nat dfnumber := depth_first_count;    /* save current count */
        depth_first_count := depth_first_count+1;
        table[α] := dfnumber;    /* initialize with current depth */
        push(stack, α);
        Atomset succ := children(α);
        for all (β ∈ succ) do
            depth_first_search(β);
            table[α] := min(table[α], table[β]);    /* β above α? */
        if (table[α] = dfnumber) then    /* α is the root of an SCC */
            Formulaset required := {},  fulfilled := {};
            repeat
                β := pop(stack);
                table[β] := MAXNAT;
                required := required ∪ {ψ1 | (ψ2 U+ ψ1) ∈ β};
                fulfilled := fulfilled ∪ {ψ | ψ ∈ β}
            until (α = β);    /* all elements of SCC are popped */
            if required ⊆ fulfilled    /* SCC is self-fulfilling */
            then print("φ satisfiable in M"); exit;

procedure children (Atom (w, m)) =
    if {(ψ2 U+ ψ1) ∈ m} = {} then return {}    /*no future obligations*/
    else return {(w', m') | w ≺ w',
          (ψ2 U+ ψ1) ∈ m iff ψ1 ∈ m' or ψ2 ∈ m' and (ψ2 U+ ψ2) ∈ m'}
```

Figure 6.3: Depth–first–search **LTL** model checking algorithm

- the depth-first-number of α, when α is first encountered,
- the depth–first–number of the first encountered atom belonging to the same strongly connected component as α, after return from the recursive call, and
- MAXNAT (any value for which `min`$(n,$ MAXNAT$)$ is always n), after the maximal strong component containing α has been analysed.

The main program calls `depth_first_search` for all initial atoms, where for an initial atom (w_0, m_0)

1. w_0 is the current point of $\mathcal{M}$, and
2. $m_0 \subseteq SF(\varphi)$ is any propositionally consistent set such that $\varphi \in m_0$.

If during the construction of the atom graph a maximal final SCC is found, the algorithm reports success; if the whole graph is searched without success we know that the formula is not satisfiable, and the program terminates with this result.

This algorithm is exponential in the number of $\mathbf{U}^+$-formulas, because every set of such sub-formulas determines a propositionally consistent set. It is linear in the size of the Kripke-model. In general, it can be shown that the problem of **LTL**-modelchecking is PSPACE-complete in the size of the formula and NLOGSPACE in the size of the model (see [SC86, LP85]). The exponential complexity in the length of the formula usually is not very problematic, because specification formulas tend to be rather short. The linear complexity in the size of the model is a more serious limiting factor, since in the worst case (i.e., if the formula is unsatisfiable) all atoms have to be traversed. Current technology limits the applicability of such algorithms to models with approximately $10^5 - 10^6$ reachable atoms. In later chapters we will discuss approaches which try to overcome this limit.

6.3 Model Checking for μ-calculus

Both the local and the global model checking algorithms can be easily adapted to monotonic μ**TL**. Global model checking for **CTL** unfolds the fixpoint definition of the **A** $\mathbf{U}^+$ and **E** $\mathbf{U}^+$ operators. If we restrict our attention to *continuous* μ**TL**-formulas, then this idea can be used to obtain a global model checking algorithm for these formulas. Moreover, as we will discuss in Chapter 6.5, this algorithm can be efficiently implemented using BDDs (see [BCM$^+$92]).

According to the Knaster-Tarski theorem proved in Chapter 2.4,

$$(U, \mathcal{I}, w) \models \nu q\ \varphi \quad \text{iff} \quad w \in \bigcup\{Q \mid Q \subseteq \varphi^{\mathcal{I}}\{q := Q\}\}$$

$$(U, \mathcal{I}, w) \models \mu q\ \varphi \quad \text{iff} \quad w \in \bigcap\{Q \mid \varphi^{\mathcal{I}}\{q := Q\} \subseteq Q\}$$

A function $f : 2^U \to 2^U$ is called *union-continuous*, if $f(\bigcup_{i\in I}\{x_i\}) = \bigcup_{i\in I} f(x_i)$ for any index set I. If the functional defined by φ is union-continuous, then the fixpoints can be obtained as

$$\nu q\ \varphi = lim_{i\to\omega}\varphi^i(\top)$$

$$\mu q\ \varphi = lim_{i\to\omega}\varphi^i(\bot)$$

If U is finite, then every monotonic function is union-continuous. Moreover, according to Lemma 3.14, on finite models it is sufficient to consider the limit up to the cardinality of the universe:

$$\nu q\ \varphi = lim_{i\leq|U|}\varphi^i(\top)$$

$$\mu q\ \varphi = lim_{i\leq|U|}\varphi^i(\bot)$$

Consequently, for finite domains model checking can be performed by extending the naïve global algorithm. The result is depicted in Figure 6.4.

Since every **repeat** in this algorithm can iterate up to $|U|$ times, the complexity is of order $|\varphi| \cdot |U|^{qd(\varphi)}$, where $qd(\varphi)$ is the depth of nesting of fixpoint quantors in φ. This high complexity is due to the fact that the computation of any inner fixed point formula has to be restarted from scratch for every new iteration of an enclosing fixed point quantor. For example, consider the **CTL**-formula $\mathbf{F}^*\ (\mathtt{p}_1 \wedge \mathbf{F}^*\ \mathtt{p}_2)$.

$$\mu\mathbf{TL}(\mathbf{F}^*\ (\mathtt{p}_1 \wedge \mathbf{F}^*\ \mathtt{p}_2)) = \mu q_1(\mathbf{X}\ q_1 \vee (\mathtt{p}_1 \wedge \mu q_2(\mathbf{X}\ q_2 \vee \mathtt{p}_2))).$$

In the inner fixed point formula $\mu q_2(\mathbf{X}\ q_2 \vee \mathtt{p}_2)$ there is no occurrence of q_1. Therefore, in the evaluation of μq_1, this formula has a constant value. In contrast, consider the $\mu\mathbf{TL}$ formula

$$\mu q_1(\mathtt{p}_1 \wedge \mu q_2(\mathbf{X}\ q_1 \vee \mathbf{X}\ q_2 \vee \mathtt{p}_2)).$$

Here the inner formula $\mu q_2(\mathbf{X}\ q_1 \vee \mathbf{X}\ q_2 \vee \mathtt{p}_2)$ is re-evaluated for every new iteration of q_1. That is, if $\psi(q_1, q_2) \triangleq (\mathbf{X}\ q_1 \vee \mathbf{X}\ q_2 \vee \mathtt{p}_2)^{\mathcal{M}}$ and $\varphi(q_1) \triangleq (\mathtt{p}_1 \wedge \mu q_2\psi(q_1, q_2))^{\mathcal{M}}$, we can calculate $\mu q_1\varphi(q_1)$ by iterating

$\varphi^0 \triangleq \{\}$,

```
procedure eval (Formula φ): Pointset =
    case φ of
        p : return I(p); /* interpretation of proposition p */
        q : return v(q); /* valuation of proposition variable q */
        ⊥ : return {};
        (ψ1 → ψ2) : return U \ eval(ψ1) ∪ eval(ψ2);
        ⟨R⟩ ψ : return R^-1( eval(ψ));
        νq(ψ) : H := U;
            repeat until stabilization
                H := eval(ψ{q := H});
            return H;
        μq(ψ) : H := {};
            repeat until stabilization
                H := eval(ψ{q := H});
            return H;
```

Figure 6.4: naïve global branching time μ**TL** model checking algorithm

$$
\begin{aligned}
&\psi^{0,0} \triangleq \{\} \\
&\psi^{0,1} \triangleq \psi(\varphi^0, \psi^{0,0}) = (\mathbf{X} \perp \vee\, \mathbf{X} \perp \vee\, \mathtt{p}_2), \\
&\psi^{0,2} \triangleq \psi(\varphi^0, \psi^{0,1}) = (\mathbf{X} \perp \vee\, \mathbf{X}\ (\mathbf{X} \perp \vee\, \mathtt{p}_2) \vee\, \mathtt{p}_2), \\
&\ldots \\
&\psi^{0,n+1} \triangleq \psi(\varphi^0, \psi^{0,n}) = \mu q_2(\mathbf{X} \perp \vee\, \mathbf{X}\ q_2 \vee\, \mathtt{p}_2), \text{ if } \psi^{0,n+1} = \psi^{0,n}, \\
\varphi^1 \triangleq\ & \varphi(\varphi^0) = (\mathtt{p}_1 \wedge \mu q_2(\mathbf{X} \perp \vee\, \mathbf{X}\ q_2 \vee\, \mathtt{p}_2)) = (\mathtt{p}_1 \wedge \psi^{0,n}), \\
&\psi^{1,0} \triangleq \{\} \\
&\psi^{1,1} \triangleq \psi(\varphi^1, \psi^{1,0}) = (\mathbf{X}\ (\mathtt{p}_1 \wedge \psi^{0,n}) \vee\, \mathbf{X} \perp \vee\, \mathtt{p}_2), \\
&\ldots
\end{aligned}
$$

and so on. A more sophisticated algorithm was given in [EL86]. A sequence $\nu q_1 ... \nu q_n$ or $\mu q_1 ... \mu q_n$ of nested fixpoints of the same type can be calculated by a single loop. Since ψ is monotonic, and $\varphi^0 \subseteq \varphi^1$, we have $\psi^{0,n} \subseteq \psi^{1,n}$. To compute a least fixed point, it is sufficient to start with any value below the result. Therefore, $\psi^{1,0}$ can be initialized with $\psi^{0,n}$ instead of $\perp$. Generally, when restarting the computation of an inner fixed point of the same type, we can use the last approximation result as a starting value. Thus, the value of this inner fixed point can increase at most $|U|$ times. The overall complexity of this improved algorithm is $(|\varphi| \cdot |U|)^{ad(\varphi)}$, where $ad(\varphi)$ is the alternation depth of different fixpoint quantors in φ.

In [LBC+94] the authors observe that by storing even more intermediate values, the time complexity for evaluating fixpoint formulas can be reduced to $O(|U|^{\lfloor ad/2 \rfloor + 1})$. For more information, see [BCJM96]

For the local version, there have been a number of algorithms proposed in the literature [Win91, Cle90, Sti91]. We give a sketch of the tableau method from [SW91]. The idea is to explore only a (small) part of the model by depth-first search. Each node in the tableau is marked by a sequence $\Delta, w \models \psi$, where $w \in U$ is a point in the model, ψ is a sub-formula of the given formula and Δ is a *definition list*. This is a sequence of declarations $(q_1 = \psi_1, ..., q_n = \psi_n)$, where the proposition variables q_i are pairwise disjoint and ψ_i uses at most variables from $q_1, ..., q_{i-1}$. For simplicity, we use $\vee$, $\wedge$, $\langle R \rangle$, $[R]$, μ and ν as basic operators and assume that negations only occur in literals. Furthermore, we assume that in the formula to be checked each μ and ν quantification binds a different proposition variable.

Since in [SW91] the μ-calculus is interpreted on branching structures, the tableau rules given in Figure 6.5 are nondeterministic. Any node marked $\Delta, w \models (\psi_1 \wedge \psi_2)$ has two children, where one is marked $\Delta, w \models \psi_1$ and the other $\Delta, w \models \psi_2$. For a node marked $\Delta, w \models (\psi_1 \vee \psi_2)$ there is only one child node which is either marked $\Delta, w \models \psi_1$ or $\Delta, w \models \psi_2$. Thus, for a given point w and formula φ, there are several nonequivalent completed tableaus; $w \models \varphi$ iff *some* of these tableaus is successful. A tableau is successful, if *each* leaf is successful. To turn the tableau method into a concrete model checking algorithm, we have to perform a depth-first search through all possible tableaus.

$$(\vee)\ \frac{\Delta, w \models (\psi_1 \vee \psi_2)}{\Delta, w \models \psi_i} \qquad (\wedge)\ \frac{\Delta, w \models (\psi_1 \wedge \psi_2)}{\Delta, w \models \psi_1 \quad \Delta, w \models \psi_2}$$

$$(\langle R \rangle)\ \frac{\Delta, w \models \langle R \rangle\ \psi}{\Delta, w' \models \psi} \qquad ([R])\ \frac{\Delta, w \models [R]\ \psi}{\Delta, w_1 \models \psi \quad \cdots \quad \Delta, w_n \models \psi}$$

$$(\mu)\ \frac{\Delta, w \models \mu q \psi}{\Delta', w \models \psi} \qquad (\nu)\ \frac{\Delta, w \models \nu q \psi}{\Delta', w \models \psi}$$

$$(\mathrm{PVar})\ \frac{\Delta, w \models q}{\Delta, w \models \psi}$$

Figure 6.5: Tableau rules for branching time μ**TL**

The additional regulations for the tableau rules in Figure 6.5 are:

- $(\vee)$ abbreviates the two rules where $i = 1$ and $i = 2$, respectively

- Rule ($\langle R \rangle$) can only applied if $w' \in R(w)$.
- Similarly, in rule ($[R]$), it must hold that $R(w) = \{w_1, ..., w_n\}$.
- In rule (μ) and (ν), $\Delta' \triangleq \Delta \cup \{q = \psi\}$.
- Rule (PVar) can only be applied if $(q = \psi) \in \Delta$, and there is no ancestor node which is labelled $\Delta', w \models \psi$ (with the same w and ψ).

That is, to check whether $\mu q \psi$ holds in point w, we record that q must be interpreted as a fixpoint of $\psi(q)$, and check whether ψ holds in w. Whenever we hit upon the proposition variable q in the further decomposition of $\psi(q)$, we can unfold this occurrence to ψ. However, to guarantee that the unfolding terminates, each proposition variable may be unfolded at most once in every branch of the tableau and every point of the model. Thus, for finite models each tableau is finite.

A tableau is maximal, if there is no leaf for which any rule is applicable. In a maximal tableau, a leaf $\Delta, w \models \psi$ is called *successful*, if

- $\psi = \mathtt{p} \in \mathcal{P}$ and $w \in \mathcal{I}(\mathtt{p})$, or $\psi = \neg\mathtt{p}$ and $w \notin \mathcal{I}(\mathtt{p})$,
- $\psi = q \in \mathcal{Q}$, $q \notin \Delta$, $w \in \mathbf{v}(q)$, or $\psi = \neg q$, $q \notin \Delta$, $w \notin \mathbf{v}(q)$, or
- $\psi = [R]\ \psi'$ and $R(w) = \{\}$ (Rule ($[R]$) produces no children),
- $\psi = q \in \mathcal{Q}$ and q was included in Δ by rule (ν).

In other words, a maximal tableau is not successful if it contains some unsuccessful leaf $\Delta, w \models \psi$ which satisfies

- $\psi = \mathtt{p} \in \mathcal{P}$ and $w \notin \mathcal{I}(\mathtt{p})$, or $\psi = \neg\mathtt{p}$ and $w \in \mathcal{I}(\mathtt{p})$,
- $\psi = q \in \mathcal{Q}$, $q \notin \Delta$, $w \notin \mathbf{v}(q)$, or $\psi = \neg q$, $q \notin \Delta$, $w \in \mathbf{v}(q)$, or
- $\psi = \langle R \rangle\ \psi'$ and $R(w) = \{\}$ (Rule ($\langle R \rangle$) not applicable),
- $\psi = q \in \mathcal{Q}$ and q was included in Δ by rule (μ).

With these definitions, correctness and completeness of the tableau decision method is stated in the following fact, a proof of which can be found in [SW91].

Fact 6.1 *$w \in \varphi^{\mathcal{M}}$ iff there exists a successful tableau with root $\{\}, w \models \varphi$.*

6.4 Binary Decision Diagrams

Model checking methods derive a great deal of their success from the efficiency of the data structures that are used. Propositional formulas are boolean functions. Since very powerful techniques exist for manipulation of boolean functions, it makes sense to represent temporal and predicate logic formulas as well as frames in terms of boolean functions. The general idea is to *encode* each domain element by a boolean sequence. Predicates and relations are then represented by their characteristic functions. Temporal operators are interpreted algorithmically according to their fixpoint definitions.

For any shared variables program, we can obtain an equivalent shared variables program which uses only binary domains: $D = \{0,1\}^n$. To do so, we use an arbitrary binary encoding of domain D_i and introduce for any program variable v_i over domain D_i new binary program variables $v_{i1}, ..., v_{ik}$, where $k = \lceil \log_2(|D_i|) \rceil$. This encoding can be compared to the implementation of arbitrary data types on digital computers, where each bit can take only two values.

If all program variables $V = \{v_1, ..., v_n\}$ of a shared variables program are over a binary domain, then any propositional formula φ over $\mathcal{P} = \{v_1, ..., v_n\}$ describes a set of states of the program, namely the set of all propositional models (interpretations) which validate the formula. Here we assume the substitution 0 for **false** and 1 for **true**. Vice versa, for any set of states there is a propositional formula describing this set. However, this formula is not uniquely determined; the problem of finding a shortest formula describing a given set of states is NP-hard.

The transition relation of a shared variables program with binary program variables $V = \{v_1, ..., v_n\}$ can be represented as an ordinary propositional formula over $\mathcal{P} = \{v_1, ..., v_n, v'_1, ..., v'_n\}$. If the transition relation is given as a propositional formula with equalities, we replace 0 by $\bot$, and 1 by $\top$, and $(v = v')$ by $(v \leftrightarrow v')$. For example, the formula

$$v_1 = 0 \rightarrow ((v'_1 = 1) \wedge (v'_2 = v_2) \wedge (v'_3 \neq v_3))$$

in this notation becomes

$$\neg v_1 \rightarrow (v'_1 \wedge (v'_2 \leftrightarrow v_2) \wedge \neg(v'_3 \leftrightarrow v_3))$$

For a shared variables program with n program variables over binary domains the size of the state space is 2^n. Therefore e.g. the state space of a buffer of length 10 with values between 1 and 1000 is $2^{100} \simeq 10^{30}$. The

reachable state space is a subset of this state space, which can be of the same order of magnitude. The transition relation for this buffer consists of pairs of states and therefore has a size of approximately 10^{60}.

To perform global model checking on systems of this or bigger size, we need an efficient representation of large sets.

Clearly, a set could be represented by a table of boolean values. Containment of an element in such a set could then be calculated by selecting the appropriate element from the table. Another possible representation of a set is the explicit enumeration of its elements, e.g., as a list or array. However, these representations can be rather wasteful, since they pay no respect to the internal structure of the set. For example, given the domain $D = \{0, 1, ..., 15\}$, the explicit enumeration of the set "all numbers which are even or bigger than 11" is

$$S = \{0, 2, 4, 6, 8, 10, 12, 13, 14, 15\}$$

The bitstring representation is

$$S = (1010101010101111).$$

These representations take $O(|D| \cdot \lceil \log_2(|D|) \rceil)$ memory bits. Bitstrings provide extremely efficient (constant-time) access. In model checking applications, however, the space used by the data is usually more important than the execution time. So, it is desirable to have a concise data structure for representing large sets which still permits efficient access to the elements.

Given a binary encoding $n = n_4 n_3 n_2 n_1$ of the domain D, the above explicit enumeration is

$$S = \{0000, 0010, 0100, 0110, 1000, 1010, 1100, 1101, 1110, 1111\}$$

This description corresponds to a propositional formula in distributive normal form. A much more succinct representation of the same set can be given by the formula

$$S = \{n \mid n_1 = 0 \vee n_4 = 1 \wedge n_3 = 1\}$$

Usually it is hard to find a "minimal" propositional formula describing a given set of elements. Therefore attention is restricted to formulas in some normal form. *Binary decision diagrams* (BDDs, [Bry86]) are a canonical form for propositional formulas. They are often substantially more compact than traditional normal forms such as conjunctive or disjunctive normal form, and they can be manipulated and evaluated very efficiently. Hence,

they have become widely used for a variety of applications in computer-aided design applications. Many present tools in symbolic simulation and verification of combinational logic and sequential circuits use a BDD library for manipulating large sets. In model checking, binary decision diagrams are the datatype of choice for the representation of propositional formulas. They can be understood as an efficient implementation of binary decision trees. Usually, the BDD is much more concise than the original decision tree. Efficiency is gained by sharing of subtrees and by elimination of unnecessary nodes.

Consider a three-place boolean connective ite ("if-then-else"), such that

$$\mathsf{ite}(\varphi, \psi_1, \psi_2) \triangleq ((\varphi \to \psi_1) \wedge (\neg\varphi \to \psi_2)).$$

Equivalently, $\mathsf{ite}(\varphi, \psi_1, \psi_2) \leftrightarrow ((\varphi \wedge \psi_1) \vee (\neg\varphi \wedge \psi_2))$. Then $(\varphi \to \psi) \leftrightarrow \mathsf{ite}(\varphi, \psi, \top)$, hence all boolean operators can be expressed with ite, $\bot$ and $\top$. A formula ψ is said to be in *tree form*, if $\psi = \bot$, or $\psi = \top$, or $\psi = \mathsf{ite}(p, \psi_1, \psi_2)$, where $p \in \mathcal{P}$ and ψ_1 and ψ_2 are in tree form. In other words, a formula ψ is in tree form, if it uses only ite, $\bot$, $\top$, and propositions, and, additionally, for every subformula $\mathsf{ite}(\psi, \psi_1, \psi_2)$ of ψ, the formula φ is a proposition, and ψ_1 and ψ_2 are not propositions. A tree form formula can be drawn as *binary decision tree*, where for each subformula $\mathsf{ite}(p, \psi_1, \psi_2)$ there is a node labelled p which has ψ_2 and ψ_1 as left and right child nodes, respectively.

Assume a linear ordering $<$ on the set $\mathcal{P}$ of propositions. A tree form formula is said to be in *ordered tree form*, if for every subformula $\mathsf{ite}(p, \varphi_1, \varphi_2)$ of φ, and every subformula $\mathsf{ite}(q, \psi_1, \psi_2)$ of φ_1 or φ_2, it holds that $p < q$. An ordered tree form formula is called *reduced*, if it does not contain any redundant subformula $\mathsf{ite}(p, \psi, \psi)$ (with equal second and third argument). The sequence of leaves of the corresponding tree of a reduced ordered tree form formula has traditionally been called the *logical spectrum* of the formula. For any given ordering, the reduced ordered tree form is a normal form. That is, for every propositional formula there is exactly one equivalent formula in reduced ordered tree form. This formula can be obtained by repeated application of the so-called *Shannon expansion*:

$$\varphi \leftrightarrow \mathsf{ite}(p, \varphi\{p := \top\}, \varphi\{p := \bot\}),$$

and boolean transformations and simplifications like $\mathsf{ite}(p, \psi, \psi) \leftrightarrow \psi$ and $(\bot \to q) \leftrightarrow \top$.

For example, truth table and tree form formula for the above set are given in Figure 6.6.

n_4	n_3	n_2	n_1	S
1	1	1	1	1
1	1	1	0	1
1	1	0	1	1
1	1	0	0	1
1	0	1	1	0
1	0	1	0	1
1	0	0	1	0
1	0	0	0	1
0	1	1	1	0
0	1	1	0	1
0	1	0	1	0
0	1	0	0	1
0	0	1	1	0
0	0	1	0	1
0	0	0	1	0
0	0	0	0	1

$$
\begin{aligned}
S = \mathrm{ite}(n_4,\ & \\
& \mathrm{ite}(n_3,\ \\
& \quad \mathrm{ite}(n_2,\ \\
& \quad\quad \mathrm{ite}(n_1, \top, \top),\ \\
& \quad\quad \mathrm{ite}(n_1, \top, \top)),\ \\
& \quad \mathrm{ite}(n_2,\ \\
& \quad\quad \mathrm{ite}(n_1, \bot, \top),\ \\
& \quad\quad \mathrm{ite}(n_1, \bot, \top))),\ \\
& \mathrm{ite}(n_3,\ \\
& \quad \mathrm{ite}(n_2,\ \\
& \quad\quad \mathrm{ite}(n_1, \bot, \top),\ \\
& \quad\quad \mathrm{ite}(n_1, \bot, \top)),\ \\
& \quad \mathrm{ite}(n_2,\ \\
& \quad\quad \mathrm{ite}(n_1, \bot, \top),\ \\
& \quad\quad \mathrm{ite}(n_1, \bot, \top))))
\end{aligned}
$$

Figure 6.6: Truth table and tree form formula

The reduced ordered tree form formula for the ordering (n_4, n_3, n_2, n_1) is obtained by repeatedly replacing every redundant subformula $\mathrm{ite}(p, \psi, \psi)$ in the above tree form formula by ψ:

$$S = \mathrm{ite}(n_4, \mathrm{ite}(n_3, \top, \mathrm{ite}(n_1, \bot, \top)), \mathrm{ite}(n_1, \bot, \top))$$

In a reduced ordered tree form formula, there might be several identical subformulas. In order to further reduce the length of the formula, we introduce names for subformulas. An *abbreviated* formula is a formula over the extended alphabet $\mathcal{P}_0 \triangleq \mathcal{P} \cup \{\delta_1, ..., \delta_n\}$, together with a (nonrecursive) list of *abbreviations* ($\delta_1 \triangleq \psi_1$, ..., $\delta_n \triangleq \psi_n$). In each abbreviation, ψ_i is an abbreviated formula $\mathrm{ite}(p, \varphi, \varphi')$ over the alphabet $\mathcal{P}_i \triangleq \mathcal{P} \cup \{\delta_{i+1}, ..., \delta_n\}$. A formula is *maximally abbreviated*, if

1. no compound subformula $\mathrm{ite}(p, \varphi_1, \varphi_2)$ appears twice, and

2. no two abbreviations have the same right hand side.

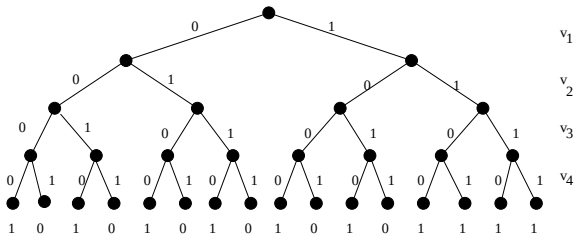
For the above example, a maximally abbreviated formula is

$$S = \mathrm{ite}(n_4, \mathrm{ite}(n_3, \top, \delta), \delta), \text{ where } \delta \triangleq \mathrm{ite}(n_1, \bot, \top)$$

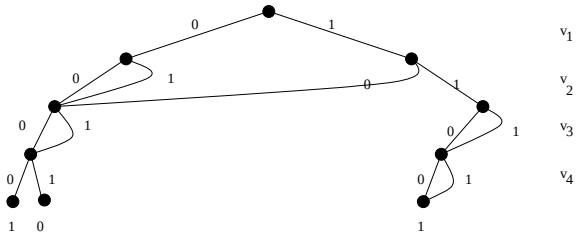
A maximally abbreviated formula is in *BDD form*, if for all subformulas $\mathrm{ite}(p, \varphi_1, \varphi_2)$, both φ_1 and φ_2 are from $\{\bot, \top, \delta_1, ..., \delta_n\}$. This normal form can be obtained by introducing further definitions:

$$S = \mathrm{ite}(n_4, \delta_1, \delta_2), \text{ where } \delta_1 \triangleq \mathrm{ite}(n_3, \top, \delta_2) \text{ and } \delta_2 \triangleq \mathrm{ite}(n_1, \bot, \top)$$

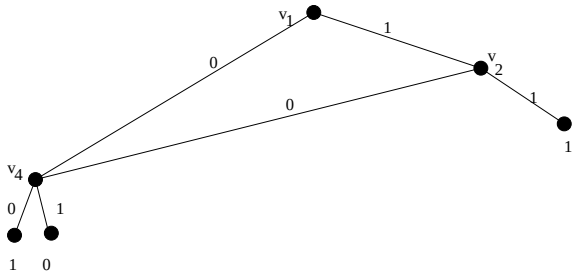
BDD form formulas can be drawn as *binary decision diagrams*: For any $\delta \triangleq \mathrm{ite}(p, \delta_1, \delta_2)$, draw a node labelled p with reference δ, which has the nodes referenced by δ_2 and δ_1 as left and right children, respectively. To illustrate these ideas with pictures, we give the binary decision tree for the above example S:



(As a convention, nodes labelled by $\top$ and $\bot$ are denoted by "+" and "-", respectively.) This tree is just a transcription of the truth table of S's characteristic function. It has many isomorphic subtrees. For any two isomorphic subtrees it is sufficient to maintain only one copy. We can replace the other one by a link to the corresponding subtree.



(For clarity, we did not identify the two "+" leafs). In the resulting structure, there are nodes for which both alternatives lead to the same subtree. These nodes represent redundant decisions and can be eliminated.



The resulting graph is the (ordered) binary decision diagram for this set with ordering (n_4, n_3, n_2, n_1). Given a variable ordering, there is a canonical BDD for every formula. The size of the BDD depends on the *structure* of the represented set rather than on its cardinality. For example, the representation of the empty set and the full set are both of constant size one. Because of this dependence on the structure of the represented object, the description by BDDs is sometimes called *symbolic*, and techniques using BDDs to represent objects are called *symbolic techniques.*

It can be constructed using the Shannon expansion in a simple recursive descent:

$$\varphi(v_i...v_n) \leftrightarrow \mathrm{ite}(v_i, \varphi\{v_i := \top\}(v_{i+1}...v_n), \varphi\{v_i := \bot\}(v_{i+1}...v_n))$$

This gives the unique binary decision tree for the chosen ordering. To obtain the BDD for $\varphi(v_i...v_n)$ we recursively calculate the BDD δ_1 for $\varphi\{v_i := \top\}(v_{i+1}...v_n)$ and δ_2 for $\varphi\{v_i := \bot\}(v_{i+1}...v_n)$. Upon backtrack, a new node $\delta \triangleq \mathrm{ite}(v_i, \delta_1, \delta_2)$ is added to the BDD. However, we do *not* create a new node if both branches in the recursion are equal (return a common result), or if an equivalent node already exists in the BDD. To check this latter condition, we implement the set of BDD nodes $\delta \triangleq \mathrm{ite}(v, \delta_1, \delta_2)$ as a hash table from (v, δ_1, δ_2) to δ.

Each entry in the hash-table is a quadruple $(\delta, v, \delta_1, \delta_2)$: pointers to BDD nodes are represented as integer numbers. A BDD is identified by its topmost node, and 0 is a pointer to $\bot$ and 1 is a pointer to $\top$. That is, the type "Bdd" is defined as "Int". Likewise, variable names are represented as integer numbers; for clarity we introduce the type "Bddvar" which is also defined as "Int". Thus, for each BDD node $(\delta, i, \delta_1, \delta_2)$ in the hash table, δ (of type "Bdd") is the number of the BDD node, i (of type "Bddvar") is the number of a BDD variable, and δ_1 and δ_2 (of type "Bdd") are links to other BDD nodes. For each (i, δ_1, δ_2) the hash table returns the pointer δ, if this node exists in the BDD.

The resulting algorithm is given in Figure 6.7. It takes as input a **PL** formula with $\mathcal{P} = \{v_1, ..., v_n\}$ and calculates the table of BDD nodes and a pointer to the topmost node for the variable ordering $(v_1, ..., v_n)$.

```
function PL2BDD (Formula φ) : (Nodeset, Bdd) =
    /* Calculates the BDD of φ
        as a set of nodes and a pointer to the topmost node */
    Nodeset table := ∅; /* Table of BDD nodes (δ, i, δ1, δ2) */
    Bdd max := 1; /* Index of maximal table entry */
    Bdd result := BDD(φ,1); /* Index of topmost BDD node */
    return (table, result);

function BDD (Formula φ, Bddvar i) : Bdd =
    /* φ is the current subformula, i is the current BDD variable */
    /* Return value is a pointer to the maximal BDD node */
    if i > n then return eval(φ) /* φ is a boolean constant */
    else δ1 := BDD(φ{v_i := ⊥}, i + 1); δ2 := BDD(φ{v_i := ⊤}, i + 1);
        if δ1 = δ2 then return δ1
        elsif ∃δ : (δ, i, δ1, δ2) ∈ table then return δ
        else max := max + 1; table := table ∪ {(max, i, δ1, δ2)};
            return max;
```

Figure 6.7: Transformation of propositional formulas into BDDs

In the BDD representation of sets, several operations can be performed very efficiently. Checking whether a given element w is contained in a set $W \subseteq U$ is done in time $O(\log |U|)$ by traversing the BDD of W according to the bitstring encoding $\vec{w}$ of w. Addition and deletion of elements as well as union and intersection of sets can be done by recursive descent. We now describe this procedure for the implication. Note that the ite-operator commutes with other boolean connectives:

$$\begin{aligned} (\mathrm{ite}(p, \varphi_1, \varphi_2) \to \psi) &\quad \leftrightarrow \quad \mathrm{ite}(p, (\varphi_1 \to \psi), (\varphi_2 \to \psi)) \\ (\psi \to \mathrm{ite}(q, \varphi_1, \varphi_2)) &\quad \leftrightarrow \quad \mathrm{ite}(q, (\psi \to \varphi_1), (\psi \to \varphi_2)) \end{aligned}$$

Similar equivalences hold for $\wedge$, $\vee$, etc. We prove only the first one of these equivalences. Recall that $\mathrm{ite}(p, \varphi_1, \varphi_2)$ is defined by $\mathrm{ite}(p, \varphi_1, \varphi_2) \leftrightarrow ((p \to \psi_1) \wedge (\neg p \to \psi_2))$.

$$(\mathrm{ite}(p, \varphi_1, \varphi_2) \to \psi) \quad \leftrightarrow \quad (((p \wedge \varphi_1) \vee (\neg p \wedge \varphi_2)) \to \psi)$$

$$\begin{aligned}
&\leftrightarrow \quad (((\neg p \vee \neg\varphi_1) \wedge (p \vee \neg\varphi_2)) \vee \psi) \\
&\leftrightarrow \quad ((\neg p \vee \neg\varphi_1 \vee \psi) \wedge (p \vee \neg\varphi_2 \vee \psi)) \\
&\leftrightarrow \quad ((p \wedge (\varphi_1 \rightarrow \psi)) \vee (\neg p \wedge (\varphi_2 \rightarrow \psi))) \\
&\leftrightarrow \quad \mathsf{Ite}(p, (\varphi_1 \rightarrow \psi), (\varphi_2 \rightarrow \psi))
\end{aligned}$$

□

Given BDDs for φ and ψ, the BDD for $(\varphi \rightarrow \psi)$ can be constructed as follows. Since $BDD(\varphi)$ and $BDD(\psi)$ can be either 0, 1, or $\mathsf{Ite}(v, \delta_1, \delta_2)$, there are nine cases which have to be considered. If $BDD(\varphi)$ is 0 or $BDD(\psi)$ is 1, the resulting BDD is 1. If $BDD(\varphi)$ is 1, the resulting BDD is $BDD(\psi)$. If $BDD(\varphi)$ is an internal node $\mathsf{Ite}(v, \delta_1, \delta_2)$, and $BDD(\psi)$ is the leaf 0, we use the equivalence:

$$(\mathsf{Ite}(v, \delta_1, \delta_2) \rightarrow \bot) \qquad \leftrightarrow \qquad \mathsf{Ite}(v, (\delta_1 \rightarrow \bot), (\delta_2 \rightarrow \bot))$$

Since $\neg\varphi \triangleq (\varphi \rightarrow \bot)$, this means that the BDD for $\neg\varphi$ is constructed from the BDD for φ by exchanging all leafs 0 and 1. The only remaining case is that both $BDD(\varphi) = \mathsf{Ite}(v, \varphi_1, \varphi_2)$ and $BDD(\psi) = \mathsf{Ite}(v', \psi_1, \psi_2)$ are internal nodes. There are three subcases:

1. $v = v'$: $(\mathsf{Ite}(v, \varphi_1, \varphi_2) \rightarrow \mathsf{Ite}(v, \psi_1, \psi_2)) \leftrightarrow \mathsf{Ite}(v, (\varphi_1 \rightarrow \psi_1), (\varphi_2 \rightarrow \psi_2))$

2. $v < v'$ in the order of variables:
   $(\mathsf{Ite}(v, \varphi_1, \varphi_2) \rightarrow \mathsf{Ite}(v', \psi_1, \psi_2))$
   $\leftrightarrow \mathsf{Ite}(v, \varphi_1 \rightarrow \mathsf{Ite}(v', \psi_1, \psi_2), \varphi_2 \rightarrow \mathsf{Ite}(v', \psi_1, \psi_2))$

3. $v > v'$ in the order of variables:
   $(\mathsf{Ite}(v, \varphi_1, \varphi_2) \rightarrow \mathsf{Ite}(v', \psi_1, \psi_2))$
   $\leftrightarrow \mathsf{Ite}(v', \mathsf{Ite}(v, \varphi_1, \varphi_2) \rightarrow \psi_1, \mathsf{Ite}(v, \varphi_1, \varphi_2) \rightarrow \psi_2)$

In all of these subcases, the BDD for $(\varphi \rightarrow \psi)$ is constructed by a recursive call according to the indicated equivalence. Again, upon backtrack a new node is created only if both links are different and no equivalent node exists so far. The algorithm is given in Fig. 6.8. Some BDD implementations use negated edges to avoid the recursive descent for $\neg\varphi$. Other implementations hash subformulas, such that certain recursive descents can be avoided all together. For more information, see [BRB90].

The complexity of the function `BDD_imp` is linear in the size of the argument BDDs. In principle, all 16 two-argument boolean operations on BDDs can be implemented with linear complexity via this procedure. For example, the BDD for the intersection of two sets φ and ψ can be calculated from the BDDs of φ and ψ using the definition $(\varphi \wedge \psi) \leftrightarrow \neg(\varphi \rightarrow \neg\psi)$. In practice,

```
function BDD_imp (Bdd φ, ψ) : Bdd =
    /* Calculates the BDD of (φ → ψ) from the BDDs of φ and ψ */
    if φ = 0 or ψ = 1 then return 1
    elsif φ = 1 then return ψ
    elsif ψ = 0 and (φ, i, φ1, φ2) ∈ table_φ
        then return new_node(i, BDD_imp(φ1, 0), BDD_imp(φ2, 0))
    else (φ, i, φ1, φ2) ∈ table_φ and (ψ, j, ψ1, ψ2) ∈ table_ψ
        if i = j then
            return new_node(i, BDD_imp(φ1, ψ1), BDD_imp(φ2, ψ2))
        elsif i < j then
            return new_node(i, BDD_imp(φ1, ψ), BDD_imp(φ2, ψ))
        elsif i > j then
            return new_node(j, BDD_imp(φ, ψ1), BDD_imp(φ, ψ2));

function new_node (Bddvar i, Bdd δ1, δ2) : Bdd =
    /* Returns a pointer to a new or existing BDD node */
    /* i is number of a BDD variable, δ1, δ2 pointers to BDD nodes */
    if δ1 = δ2 then return δ1
    elsif ∃δ : (δ, i, δ1, δ2) ∈ table then return δ
    else max := max + 1; table := table ∪ {(max, i, δ1, δ2)};
        return max;
```

Figure 6.8: Combination of BDDs

however, most BDD libraries achieve a better performance by providing for each connective a special recursive procedure which takes symmetries and idempotences in the arguments into respect. [Bry86] gives a uniform scheme to handle all 16 boolean connectives. In Fig. 6.9 this generic `BDD_apply` function is given; the idea of using a co-factoring function is from the BDD library by D. Long.

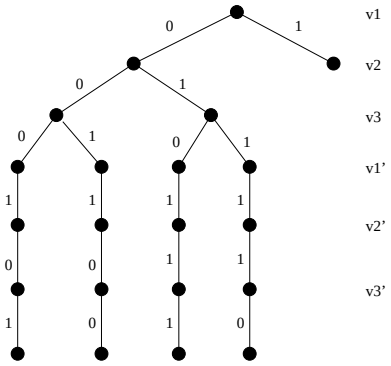
```
function BDD_apply (Fun ∘, Bdd φ, ψ) : Bdd =
    /* Calculates the BDD of (φ ∘ ψ) from BDDs of φ and ψ */
    if φ ∈ {0,1} and ψ ∈ {0,1} then return φ ∘ ψ
    else m := min_var(φ, ψ);
        (f0, f1) := co_factor(φ, m);   (g0, g1) := co_factor(ψ, m);
        δ1 := BDD_apply(∘, f0, g0);   δ2 := BDD_apply(∘, f1, g1);
        return new_node(m, δ1, δ2);

function min_var (Bdd φ, ψ) : Bddvar =
    /* Returns the minimal BDD variable in φ and ψ */
    if φ ∈ {0,1} and (ψ, j, ψ1, ψ2) ∈ table then return j
    elsif (φ, i, φ1, φ2) ∈ table and ψ ∈ {0,1} then return i
    elsif (φ, i, φ1, φ2) ∈ table and (ψ, j, ψ1, ψ2) ∈ table
        then return min(i, j);

function co_factor (Bdd δ, Bddvar m) : (Bdd, Bdd) =
    /* Returns two BDD pointers to combine */
    if δ ∈ {0,1} then return (δ, δ)
    else /* (δ, i, δ1, δ2) ∈ table */
        if i > m then return (δ, δ) else return (δ1, δ2);
```

Figure 6.9: Applying arbitrary functions to BDDs

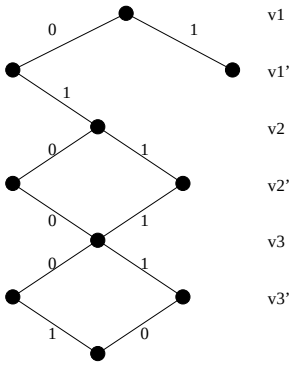
For a given boolean function, the size of the BDD depends critically on the ordering of the variables. For the example formula above

$$v_1 = 0 \rightarrow ((v_1' = 1) \wedge (v_2' = v_2) \wedge (v_3' \neq v_3))$$

and the variable ordering $(v_1, v_2, v_3, v_1', v_2', v_3')$, the above algorithm yields the following BDD. (We omit all branches leading to negative leafs.)



For the variable ordering $(v_1, v_1', v_2, v_2', v_3, v_3')$, however, we obtain the following much smaller BDD:



This is a common phenomenon when working with BDDs. In general, a good heuristics is to keep "dependent" variables as close together in the ordering as possible. For a more formal treatment in the context of sequential circuits, see [McM93]. Unfortunately, the problem of finding an *optimal* variable ordering is NP-hard. Basically, for every possible ordering one has to construct the BDD and compare their sizes, which is not feasable. Automatic reordering strategies usually proceed by steepest ascend heuristics [FYBV93, Rud93].

6.5 Symbolic Model Checking

In the global algorithm for model checking the propositional μ-calculus, all set operations can be directly performed with BDDs. Calculation of the BDD for $\langle R \rangle\ \psi$ from the BDD for ψ amounts to calculation the inverse image of ψ under the relation R. This is done using propositional quantification: Recall that the BDD for ψ is using variables $(v_1, ..., v_n)$, and the BDD for R is defined over the variables $(v_1, ..., v_n, v'_1, ..., v'_n)$. To get the BDD for $\langle R \rangle\ \psi$, we first rename all variables v_i in the BDD for ψ by v'_i, then build the intersection of this BDD with the BDD for R to obtain a BDD over $(v_1, ..., v_n, v'_1, ..., v'_n)$, and then "throw away" all primed variables by an existential quantification. In fact, all these operations can be performed during a single BDD traversal, if v_i and v'_i are always kept together in the variable order.

We now describe how symbolic model checking can be used for the relational μ-calculus. Given a finite relational frame $\mathcal{F} \triangleq (S, \mathcal{I})$ and a relational term ρ or formula φ, model checking can be used to determine the denotation $\rho^{\mathcal{F}}$ or $\varphi^{\mathcal{F}}$, respectively. In [BCM+92], a similar algorithm for a closely related logic is given (see Figure 6.10). Assume for simplicity that each domain is binary; for non–binary domains the algorithm can be extended by an appropriate encoding. In the frame, the interpretation $\mathcal{I}$ of a relation of type $(D_1, ..., D_n)$ is represented by a BDD with variables $v_1, v_2, ..., v_n$.

Likewise, a function of type $(D_0, ..., D_m)$ can be coded as a BDD over $v_0, ..., v_m$. Constants are represented as BDDs with a single node, namely $\delta \triangleq \mathsf{Ite}(v, 0, 1)$ or $\delta \triangleq \mathsf{Ite}(v, 1, 0)$. Function application $f t_0 ... t_{m-1}$ is done by renaming the variable v in t_i by v_i, renaming v_m in f by v and conjoining the resulting BDDs. For constant terms, this is the same as replacing each node $\mathsf{Ite}(v_i, \delta_1, \delta_2)$ by δ_1 or δ_2, respectively. In the BDD representation of functions, for each node $\delta \triangleq \mathsf{Ite}(v^i_m, \delta_1, \delta_2)$ it holds that either δ_1 or δ_2 is 0. This is due to the fact that for each argument there exists exactly one function value. Each individual variable x_i of type D corresponds to $\lceil \log |D| \rceil$ boolean variables $x^1_i, ... x^n_i$.

A variable valuation is given as a mapping from individual variables to the BDD constants $\{0, 1\}$, and from relation variables to BDD nodes. A term or formula with free individual variables $x_1, ..., x_m$ is represented as a BDD with additional BDD variables $x_1, ..., x_m$. A relation variable is represented by its name; each BDD node can contain (the name of) a relation variable as one of its successors. In other words, each BDD node is a tuple $(\delta, i, \delta_1, \delta_2)$, where δ is the name of this node, i is a variable from the set $\{v_1, ..., v_n, x_1, ..., x_m\}$, and each δ_j is one of the BDD constants 0 or 1, a

name of another BDD node, or the name of a relation variable. Substitution of a relation variable with a relation in a BDD can be done by a simple BDD traversal.

function `BDD_form` (Formula φ, Interpretation $\mathcal{I}$) : Bdd =
  /* Calculates the BDD of formula φ in the interpretation $\mathcal{I}$ */
  case φ **of**
    $x \in \mathcal{V}$: **return** ite$(x, 1, 0)$;
    $(x_1 = x_2)$: **return** ite$(x_1,$ ite$(x_2, 1, 0),$ ite$(x_2, 0, 1))$;
    $\bot$: **return** 0;
    $(\varphi_1 \to \varphi_2)$:
      return `BDD_imp`(`BDD_form`$(\varphi_1, \mathcal{I})$, `BDD_form`$(\varphi_2, \mathcal{I})$);
    $\exists x\ \varphi$: **return** `BDD_exists`$(x,$ `BDD_form`$(\varphi, \mathcal{I}))$;
    $\rho x_1 ... x_n$: **return** `BDD_term`$(\rho, \mathcal{I})\{v_1 := x_1\}...\{v_n := x_n\}$;

function `BDD_term` (RelationalTerm ρ, Interpretation $\mathcal{I}$) : Bdd =
  /* Calculates the BDD of term ρ in the interpretation $\mathcal{I}$ */
  case ρ **of**
    $R \in \mathcal{R}$: **return** $\mathcal{I}(R)$ /* pointer to BDD for R */;
    $X \in \mathcal{V}$: **return** X /* name of X */;
    $\lambda x_1 ... x_n\, \varphi$: **return** `BDD_form`$(\varphi, \mathcal{I})\{x_1 := v_1\}...\{x_n := v_n\}$;
    $\mu X\, \rho$: $r :=$ `BDD_term`$(\rho, \mathcal{I})$; **return** `BDD_lfp`$(r, 0)$;

function `BDD_lfp` (BDD r, BDD X^i) : BDD =
  /* Fixpoint iteration of BDD r for ρ w. substitution $\{X := X^i\}$ */
  $X^{i+1} := r\{X := X^i\}$;
  if $X^{i+1} = X^i$ **then return** X^i
  else return `BDD_lfp`$(r,\ X^{i+1})$;

Figure 6.10: Symbolic evaluation of formulas and terms

The model checking algorithm is divided into two functions, `BDD_form` and `BDD_term`, which recurse over the structure of the formula and term. `BDD_form` inputs a formula φ and (the BDD representation of) the interpretation $\mathcal{I}$ in frame $\mathcal{F}$, and returns a BDD which is satisfied by a given valuation $\mathbf{v}$ iff $(S, \mathcal{I}, \mathbf{v}) \models \varphi$. The first five cases in the function derive directly from the respective semantic definitions and should require no explanation. The last case, application of a relation term ρ, uses the function `BDD_term`$(\rho, \mathcal{I})$ to find a representation of the relational term ρ (under the

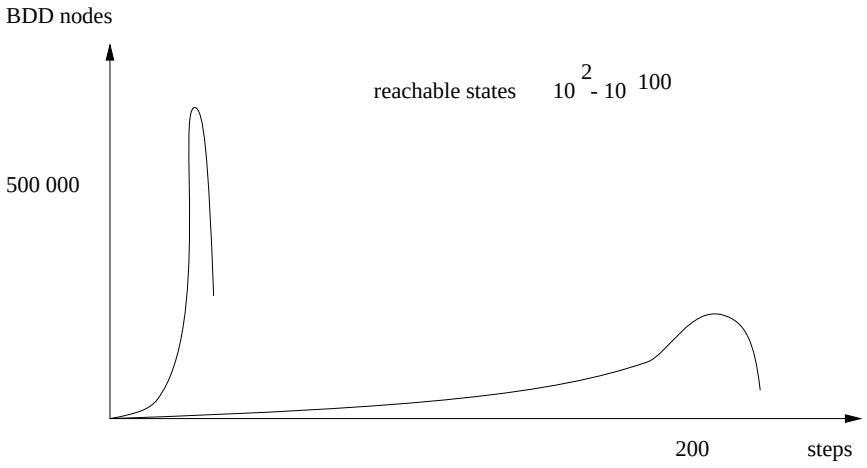
interpretation $\mathcal{I}$), then substitutes the argument variables x_1, ..., x_n for the place-holder variables v_1, ..., v_n, producing a BDD which is satisfied iff ρ holds for $x_1, ..., x_n$.

The function `BDD_term` takes as arguments a relational term ρ and the BDD representation of the interpretation $\mathcal{I}$. It returns a BDD which represents the relation term in the manner described above. The first and second case in the definition of `BDD_term`, a relation symbol or relation variable, simply return the BDD representation of the relation in the interpretation or the name of the relation variable, respectively. The third case, λ-abstraction, produces a BDD with variables v_1, ..., v_n substituted for the variables v_1, ..., v_n. This is the representation for an n-ary relation which holds iff its arguments satisfy the formula φ when assigned to x_1, ..., x_n. The most interesting case is the last: the fixed point operator μ. To find the fixed point of a relational term with respect to a free relation variable X we use the standard technique for finding the least fixed point of a monotonic functional in a finite domain. First we evaluate `BDD_term`$(\rho, \mathcal{I})$ to get a BDD r for ρ. Then we compute the fixed point by a series of approximations X^0, X^1, ..., beginning with the empty relation (which is represented by the BDD constant 0). To compute the BDD X^{i+1} from X^i we substitute all occurrences of the variable X in the BDD r with X^i. Since the domain is finite and ρ is positive in X, the series must converge to the least fixed point (cf. Lemma 3.14). Convergence is detected when $X^{i+1} = X^i$. In this case, X^i is the BDD for $\mu X\, \rho$. Note that testing for convergence is easy, since with a hash-table implementation of BDD nodes equality can be determined in constant time (cf. the algorithm in Fig. 6.7).

The μcke model checker [Bie97] is one of the first tools for the relational μ-calculus. It allows to use arbitrary finite domains defined by enumerations, subranges, arrays and compounds. For each non-binary domain, an appropriate binary encoding is generated automatically. The model is given in a C-like input language. It is compiled into an internal BDD representation. For each domain, an appropriate binary encoding is generated automatically. At present, μcke allows only constant (nullary function) symbols in the signature; n-ary functions and relations (e.g., arithmetical operations and comparisons on domain elements) can be defined by λ-abstraction as shown above. Since μcke uses several sophisticated heuristics for the allocation of BDD variables, its performance is comparable to more specialized systems like SMV.

The complexity of symbolic model checking for μ-calculus is potentially exponential in the number of variables and exponential in the formula. Nevertheless, in practice the number of iteration steps required to reach a fixed

point is often small ($\leq 10^3$). For hardware systems, that is, in the verification of sequential circuits, most states are reachable in very few steps, but the BDDs tend to grow exponentially in the first few steps. For software systems, especially if there is not much parallelism contained, the BDD often grows only linear with the number of steps, until the whole state space is traversed. The following picture shows the relation between the BDD size and number of steps in typical examples.



Part III

Real Time

Chapter 7

Formalisms for Real-Time

7.1 Real-Time and Hybrid Systems

Within the last decade, formal analysis methods have been applied also to *real time* and *hybrid systems*. In contrast to untimed systems, where only *causal* aspects of time are important, in some applications it is desirable to consider *quantitative* aspects of timing behavior. We say that a system has to satisfy *hard real time constraints*, if its correctness depends on the value or progress of "the real" clock. Without entering a philosophical debate, in real-time verification we assume that there exists a universal global notion of time. With sufficient preciseness, a second denotes the same amount of time for all objects on earth. Most countries maintain a "reference clock", which is used to synchronize all other clocks. In hard real-time systems, not only the relative order of events is important, but also their absolute duration with respect to this (conceptual) global clock. For example, in a traffic light controller, it might not be sufficient to show that if a pedestrian pushes a button, then *eventually* the green lights will be on. To allow approaching cars to pass, the light should stay red after the button has been pushed *for at least 10 seconds*. To avoid that pedestrians start crossing at red, it should also change *not later than 30 seconds* after the request. In this example, we assume that both the pedestrian and the traffic light controller have the same measure of the duration of a second. Of course, it is possible to model the global clock as separate concurrent part of the system. Then this global clock synchronizes the local clocks of both pedestrian and traffic light controller. Thus, it is possible to consider real-time verification as special case of the untimed methods described above. However, in hard real-time systems, global time is ubiquitous, therefore this approach may not be the

most efficient.

It is important to note that "hard real time" does not mean "as fast as possible". As the above example shows, predictability of timing behavior can also mean that some events do not occur before a certain amount of time has elapsed. As another example, consider a real-time protocol, where all necessary computation steps *must* be performed in *exactly* a fixed time slot. Currently, hard real time systems are designed with trial and error: if a component is too fast, an idle waiting loop is incorporated; if it is too slow, more expensive hardware is used. This method has several disadvantages. Firstly, it can add intricate hardware-software dependencies to a system. Therefore the migration to new hardware generations is complicated. Secondly, the execution time of single statements can vary depending on input data, nondeterministic scheduling, cache behavior, etc. Timing measurement can not guarantee that the actual timing will be within required boundaries. Finally, in applications like the design of asynchronous circuits, an arbitrary delay of signals can be expensive.

As mentioned above, a *hybrid system* combines discrete and continuous components. For example, consider a system consisting of a water tank with inlet and outlet valves, a sensor measuring the current filling rate, and a digital controller reading the sensor data and opening or closing the valves. In this example, the value of the continuous component (the filling rate) depends linearly on the value of the discrete components (the setting of the valves) and the flow of time. In more complex examples, e.g., a train braking control system, this dependency could be expressed by a nonlinear differential equation system.

A real-time clock is the special case of a continuous component: its value changes constantly over time. Thus every real-time system is a special hybrid system. The example above shows that not every hybrid system can be reduced to a real-time system. There are, however, large classes of hybrid systems for which such a reduction is possible.

In real time verification, clock values usually are assumed to be nonnegative real, rational or natural numbers. As opposed to untimed systems, there is no generally accepted representation of sets or regions of timing values. Common tools use *difference bound matrices* [Dil89a] and *clock regions* [ACD90, Alu91] to represent timing constraints. Real time systems often are modelled with timed automata [Alu98] or timed transition systems [HMP92]. Reachability and model checking algorithms for these models are given in [ACD90]. Generally, the verification of real-time systems is much higher than that of untimed systems. Moreover, timing constructs are often represented using an explicit state representation. Consequently, the

number of states that can be handled is relatively small ($10^5 - 10^7$). Thus, at present, only small examples can be verified automatically by model checking tools like KRONOS [DOTY96, Yov97, Yov98, BDM+98] or UPPAAL [BLL+96, LPY97]. For a recent comparison of HyTech, Kronos and UPPAAL on a railroad crossing example, see [BS00].

For untimed systems, the state explosion problem in model checking can be avoided by partial orders and thus to avoid the construction of equivalent states reachable by different interleaving of atomic events. Several methods [Val90, God90] based on this approach have been proposed for reachability analysis and various other properties of Petri nets. In this part, we describe how these methods can be extended to real-time. Recently, some model checkers have introduced partial-order packages also for real-time. We give some experimental results showing the viability of these methods. All results of this part of the book are joint research with T. Yoneda.

7.2 Timed Automata and Time Petri Nets

Real-time systems are often represented by finite automata, whose transitions are labeled by time intervals [AH92, and others], or which have a finite number of clocks [Alu98, ACD90].

Let $\mathbf{Q}$ be the set of rational numbers, and $\mathbf{T}$ the set of nonnegative rational numbers. $\mathbf{T}$ is the so-called *time-domain*. Mostly, we will use τ as a metavariable to range over time points (this is completely unrelated to the CSP τ-event). In our context, a *time transition system* T is a tuple $T \triangleq (\Sigma, S, \Delta, S_0)$, where

- Σ is the *alphabet*,
- S is a nonempty set of *states*,
- $\Delta \subseteq S \times \Sigma \times S$ is the *transition relation*, and
- $S_0 \subseteq S$ is the set of *initial states*.
- Eft, $\mathsf{Lft} : \Delta \to \mathbf{T}$ are functions for the *earliest* and *latest firing times* of transitions, satisfying $\mathsf{Eft}(t) \leq \mathsf{Lft}(t)$ for all $t \in \Delta$;

Time transition systems generate *timed ω-words*: finite or infinite sequences of *events*. An event e is a tuple $e \triangleq (a, \tau)$, where $a \in \Sigma$ and $\tau \in \mathbf{T}$. Let $\sigma \triangleq (e_1, e_2, e_3, ...)$, where $e_i \triangleq (\sigma_i, \tau_i)$, be such an ω-word over Σ. We say that σ is *generated by* the timed transition system T, if there exists a sequence $s_0, t_1, s_1, t_1, s_2, ...$ of states and transitions such that

- $s_0 \in S_0$,
- for each $i > 0$ it holds that $t_i = (s_{i-1}, \sigma_i, s_i) \in \Delta$, and
- $\tau_i + \mathsf{Eft}(t_i) \leq \tau_{i+1} \leq \tau_i + \mathsf{Lft}(t_i)$.

Usually, the following *non-Zeno'ness* condition is imposed on generated words:

- Time increases beyond any bound: if σ is infinite, then for any $\tau \in \mathbf{T}$ there exists an index i such that $\tau_i > \tau$

A somewhat weaker requirement would be that in any finite time interval there are only finitely many transitions. The *generated language* of a timed transition system T is the set of all non-Zeno timed ω-words which are generated by T. One possibility to enforce non-Zeno'ness is the syntactical condition that in any loop in the transition graph the sum of earliest firing times is greater then zero. We will come back to this *progress condition* below.

A *timed automaton* [ACD90, Dil89a] is a finite automaton augmented with a finite set of real-valued *clocks*. A *location* λ of such an automaton is a tuple $\lambda \triangleq (s, \mathsf{clock})$, where $s \in S$ is a state and clock is an assignment of clock values to the time domain $\mathbf{T}$. In this model, there are two sorts of changes of location: transitions between states are instantaneous. Time can elapse when the automaton is in a state. A clock constraint, called a *guard*, is associated with each transition. The transition can only be taken if the current values of the clocks satisfy this clock constraint. A clock constraint is also associated with each state of the automaton. This constraint is called the *invariant* of the state. Time can elapse in a state only as long as the invariant of the state is true. When a transition occurs, some of the clocks may be reset to zero. At any instant, the reading of a clock is equal to the time that has elapsed since the last time the clock was reset. Moreover, time passes at the same rate for all clocks. Again, timed automata are usually assumed to be *non-Zeno*, i.e., only a finite number of transitions can happen within a finite amount of time.

Clearly, a time transition system can be thought of as a special case of a timed automaton, where each state $s \in S$ has a special clock c_s which is reset upon entering the state, and the transition constraints are given by $\mathsf{Eft}(t) \leq c_s \leq \mathsf{Lft}(t)$.

Unfortunately, concurrency can not be modeled directly by time transition systems or timed automata. On the other hand, *time Petri nets* were considered in [MF76], and used for timing verification in [BD91]. Time Petri

nets are an adequate model of timed concurrent systems, which generalizes other models in a natural way. Using time Petri nets, it is very easy to model, for example, logic gates with bounded delays or network protocols. The definitions here are based on [Sta90].

Formally, a *time Petri net* N is six–tuple, $N = (P, T, F, \mathsf{Eft}, \mathsf{Lft}, \mu_0)$, where

- $P = \{p_1, p_2, \ldots, p_m\}$ is a finite set of *places*;
- $T = \{t_1, t_2, \ldots, t_n\}$ is a finite set of *transitions* $(P \cap T = \emptyset)$;
- $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*;
- $\mathsf{Eft}, \mathsf{Lft} : T \to \mathbf{T}$ are functions for the *earliest* and *latest firing times* of transitions, satisfying $\mathsf{Eft}(t) \leq \mathsf{Lft}(t)$ for all $t \in T$;
- $\mu_0 \subseteq P$ is the *initial marking* of the net.

For any transition t, $\bullet t = \{p \in P \mid (p,t) \in F\}$ and $t\bullet = \{p \in P \mid (t,p) \in F\}$ denote the *preset* and the *postset* of t, respectively. To simplify the presentation, we require that $\bullet t \cap t\bullet = \emptyset$ for every transition t; however, this requirement is not essential for our results.

A *marking* μ of N is any subset of P. A transition is *enabled* in a marking μ if $\bullet t \subseteq \mu$ (all its input places have tokens in μ); otherwise, it is *disabled*. Let $\mathsf{enabled}(\mu)$ be the set of transitions enabled in μ.

A *location* λ of a time Petri net is a pair (μ, clock), where μ is a marking and clock is a function $T \to \mathbf{T}$. The *initial location* λ_0 is $(\mu_0, \mathsf{clock}_0)$, where $\mathsf{clock}_0(t) = 0$ for all $t \in T$.

The locations of time Petri nets change, if time passes or if a transition fires. In location $\lambda = (\mu, \mathsf{clock})$, time $\tau \in \mathbf{T}$ *can pass*, if for all $t \in \mathsf{enabled}(\mu)$, $\mathsf{clock}(t) + \tau \leq \mathsf{Lft}(t)$. In this case, location $\lambda' = (\mu', \mathsf{clock}')$ is *obtained by passing* τ from λ, if

1. $\mu = \mu'$, and
2. for all $t \in T$, $\mathsf{clock}'(t) = \mathsf{clock}(t) + \tau$.

In location $\lambda = (\mu, \mathsf{clock})$, transition $t \in T$ *can fire*, if $t \in \mathsf{enabled}(\mu)$, and $\mathsf{clock}(t) \geq \mathsf{Eft}(t)$. In this case, location $\lambda' = (\mu', \mathsf{clock}')$ is *obtained by firing* t from λ, if

1. $\mu' = (\mu \setminus \bullet t) \cup t\bullet$, and
2. for all $\hat{t} \in T$, $\mathsf{clock}'(\hat{t}) = \begin{cases} 0 & \text{if } \hat{t} \in \mathsf{enabled}(\mu'), \hat{t} \notin \mathsf{enabled}(\mu) \\ \mathsf{clock}(\hat{t}) & \text{else .} \end{cases}$

Intuitively, this can be interpreted as follows: Firing a transition t consumes no time, but updates μ and clock such that the clocks associated with newly enabled transitions (i.e. transitions which are enabled in μ' but not in μ) are reset to 0. Clock values of other transitions (i.e. transitions not affected by t) are left unchanged.

A *run* $\rho = (\lambda_0, \lambda_1, \lambda_2, \ldots)$ of N is a finite or infinite sequence of locations such that λ_0 is the initial location, and λ_{i+1} is obtained from λ_i by passing time τ and then firing transition t. We write $\lambda_i(\rho)$ for the i–th location of ρ, and similarly $\mu_i(\rho)$ and $\mathsf{clock}_i(\rho)$, and omit the argument (ρ) whenever appropriate. A run is *maximal*, if it is infinite or in its last location there is no enabled transition. The *behavior* $B(N)$ of N is the set of all maximal runs of N.

Given any run ρ and $i \geq 0$, we define $\mathsf{time}_i(\rho)$ to be the sum of all times τ passed between $\lambda_0(\rho)$ and $\lambda_i(\rho)$; that is, $\mathsf{time}_0(\rho) = 0$ and $\mathsf{time}_{i+1}(\rho) = \mathsf{time}_i(\rho) + \mathsf{clock}_{i+1}(t) - \mathsf{clock}_i(t)$ for some t which is not newly enabled in μ_{i+1}.

A location λ is *reachable* if there exists a finite run whose last location is λ. A time Petri net is *one–safe*, if for every location $\lambda = (\mu, \mathsf{clock})$ obtained by passing time from any reachable location λ', and for every transition t which can fire in λ, $t\bullet \cap \mu = \emptyset$. The restriction to one–safe nets simplifies both the analysis of time Petri nets and the reduced state space generation.

Further, for the proof of the finiteness of the graphs introduced in Sect. 8.1, we need the *progress condition* mentioned above [AH92]: The sum of earliest firing times of transitions forming any loop in N is positive. More precisely, for every set $\{t_1, t_2, \ldots, t_n\}$ of transitions such that $t_1\bullet \cap \bullet t_2 \neq \emptyset$, $t_2\bullet \cap \bullet t_3 \neq \emptyset$, ..., $t_n\bullet \cap \bullet t_1 \neq \emptyset$ it holds that $\mathsf{Eft}(t_1) + \mathsf{Eft}(t_2) + \cdots + \mathsf{Eft}(t_n) > 0$. This guarantees that in any infinite run time is increasing beyond any bound.

In the sequel, a *net* will always be a one–safe time Petri net satisfying the progress condition.

Fig. 7.1 shows an example net N_x. Pairs of numbers after transition names represent earliest and latest firing times, respectively. Since, for example, t_2 can fire at any time between 1 and 3 after being enabled, the behavior $B(N_x)$ contains an infinite number of runs. Furthermore, since $\mathsf{Eft}(t_0) > \mathsf{Lft}(t_1)$, t_0 can never fire, and thus every run of N_x is infinite.

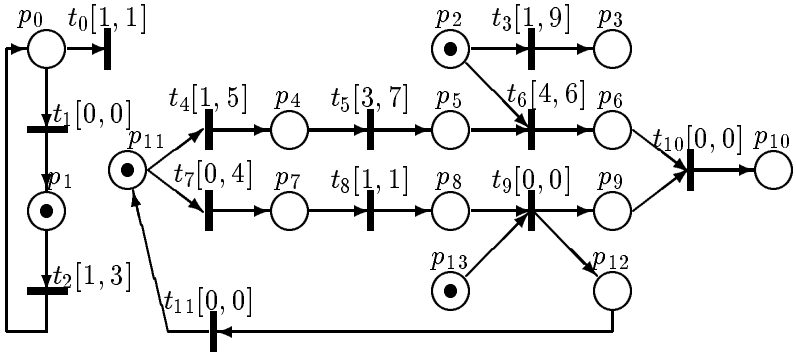


Figure 7.1: An example of a time Petri net : N_x

7.3 Time Net Logic

In order to specify and verify real–time systems, languages for reasoning about quantitative timing properties are necessary. Many timed temporal logics have been proposed to express such properties [AH92, ACD90, and others]. But again, for practical applications, state explosion is a big problem. There are only a few reports on the avoidance of state explosion in the case of real–time systems.

Subsequently, we develop an efficient model checking algorithm for the verification of real–time systems based on the partial order approach. The given real–time system is modeled by a time Petri net. For the specification of properties and time constraints of the time Petri nets we use a suitably extended linear temporal logic. The language is designed such that it fits to the partial order analysis. Automatic verification is achieved by generating a reduced state space of the net, which is big enough to evaluate the given formula, and by traversing the reduced state space with the given formula.

In this section, we propose a temporal logic for the specification of net properties. On one hand, every such logic should be expressive enough to be capable of formalizing "interesting" properties including quantitative time requirements, and on the other hand there should exist an efficient model checking algorithm for the logic avoiding the state explosion problem. With branching time logics such as **CTL** it seems to be more difficult to use the parallelism in the net to reduce the average time complexity of the model checking problem; therefore, we focus on linear time temporal logic.

Given a net N and formula φ, we want to find whether there exists a run ρ of N satisfying φ (written $\rho \models \varphi$). In general there are infinitely many runs of N, therefore we group these into a finite number of equivalence classes $[\rho_1]$, $[\rho_2]$, ..., $[\rho_c]$, such that the existence of a satisfying run ρ implies that every element of the equivalence class $[\rho]$ satisfies φ. Thus we only have to

check a finite number of equivalence classes, and a coarser partition yields a better algorithm.

Consider a set of atomic propositions $\{p_1, \ldots, p_k\}$ of a logic, such that the notion of *validity* $((\rho, i) \models p_j)$ of an atomic proposition p_j in a location λ_i of a run ρ is defined. Two runs ρ and ρ' are *strongly equivalent* with respect to $\{p_1, \ldots, p_k\}$, if $(\rho, i) \models p_j$ iff $(\rho', i) \models p_j$ for all $i \geq 0$ and all atomic propositions $p_j \in \{p_1, \ldots, p_k\}$.

A location λ_{i+1} in a run ρ is *stuttering* with respect to $\{p_1, \ldots, p_k\}$, if $(\rho, i) \models p_j$ iff $(\rho, i+1) \models p_j$ for all $p_j \in \{p_1, \ldots, p_k\}$. Two runs ρ and ρ' are *stuttering equivalent* w.r.t. $\{p_1, \ldots, p_k\}$, if the two sequences obtained by eliminating all stuttering locations from ρ and ρ' are strongly equivalent w.r.t. $\{p_1, \ldots, p_k\}$. Define a formula φ to be *stuttering invariant*, if for any two runs ρ and ρ' which are stuttering equivalent with respect to the atomic propositions in φ it holds that $\rho \models \varphi$ iff $\rho' \models \varphi$.

Stuttering invariance allows to group all stuttering equivalent runs into the same equivalence class, thereby reducing the average complexity of the model checking. In particular, all runs which differ only in the interleaving of independent transitions are stuttering equivalent with respect to places not connected to these transitions.

Unfortunately, most formulas of existing real–time logics are not stuttering invariant. Firstly, uncautious use of a "next–state" operator inhibits stuttering invariance. Moreover, if the logic allows to directly refer to the time associated with a location in a run, then a similar effect as with a "next–state" operator can result. In other words, classical real–time logics are inappropriate for our purpose. Therefore, our logic only refers to *differences* of firing times of transitions.

Our logic, which we call **TNL**, is formally defined as follows. Given any net $N = (P, T, F, \mathsf{Eft}, \mathsf{Lft}, \mu_0)$, let $\mathcal{P} = \{p^\bullet \mid p \in P\} \cup \{p^\circ \mid p \in P\}$ be the set of *time variables*. The set of *propositional variables* is P. The *formulas* of **TNL** are defined inductively:

- If $x, y \in \mathcal{P}$ and $c \in \mathbf{Q}$, then $x - y \leq c$ is a formula.
- Every propositional variable is a formula.
- $\bot$ is a formula.
- If φ_1 and φ_2 are formulas, then $(\varphi_1 \rightarrow \varphi_2)$ and $(\varphi_1 \ \mathbf{U}^* \ \varphi_2)$ are formulas.

$\bot$, propositional variables, and $x - y \leq c$ for $x, y \in \mathcal{P}$ and $c \in \mathbf{Q}$ are called *atomic propositions*. Additional boolean connectives **true**, $\neg$, $\wedge$, $\vee$, $\rightarrow$, and temporal connectives $\mathbf{G}^*$, $\mathbf{F}^*$ can be defined as shown in Chapter 2. Also

formulas $x-y \sim c$, where $\sim$ is any relation from $\{<, =, \geq, >\}$, can be defined in an obvious way.

In order to define the semantics of **TNL**, the value of time variables in a location of a run has to be defined. Intuitively, $p^\bullet$ and $p^\circ \in V$ represent the time when the place p got or lost the latest token, respectively.

Let ρ be a run of N, $i \geq 0$, and let $x \in \mathcal{P}$.

$$eval_i(x) = \begin{cases} 0 & \text{if } i = 0 \\ \mathsf{time}_i(\rho) & \text{if } x = p^\bullet,\ p \in \mu_i,\ p \notin \mu_{i-1} \\ \mathsf{time}_i(\rho) & \text{if } x = p^\circ,\ p \notin \mu_i,\ p \in \mu_{i-1} \\ eval_{i-1}(x) & \text{otherwise} \end{cases}$$

Validity of a **TNL** formula φ in a run ρ at point $i \geq 0$, denoted by $(\rho, i) \models \varphi$, is now defined by induction on φ as usual:

1. $(\rho, i) \models x - y \leq c$ iff $eval_i(x) - eval_i(y) \leq c$
2. $(\rho, i) \models p$ iff $p \in \mu_i$ for $p \in P$
3. $(\rho, i) \not\models \bot$
4. $(\rho, i) \models (\varphi_1 \rightarrow \varphi_2)$ iff $(\rho, i) \models \varphi_1$ implies $(\rho, i) \models \varphi_2$
5. $(\rho, i) \models (\varphi_1\ \mathbf{U}^*\ \varphi_2)$ iff there exists $j \geq i$ such that $(\rho, j) \models \varphi_2$, and for all k such that $i \leq k < j$, $(\rho, k) \models \varphi_1$

ρ *satisfies* φ, denoted by $\rho \models \varphi$, if $(\rho, 0) \models \varphi$. φ is *satisfiable* in N if there exists a (maximal) run $\rho \in B(N)$ such that $\rho \models \varphi$.

Consider our example net from Fig. 7.1. Then the formula $\Diamond p_{10}$ is satisfiable if the place p_{10} is reachable, which is the case, and $\Diamond(p_{10} \wedge p_{10}^\bullet - p_{10}^\circ \leq 8)$ is satisfiable if it can be reached within 8 time units, which is not the case (Note that $eval_i(p_{10}^\circ) = 0$ for all i). $\Box\Diamond(p_1^\circ - p_1^\bullet > 2)$ means that t_2 may infinitely often need more than 2 time units to fire.

Chapter 8

State Space Techniques for Real-Time

8.1 Model Checking for Time Nets

In general, there exist infinitely many runs of a given net N. In this section, we will construct a finite graph G such that the paths through G represent exactly the runs of N, and that every node in G determines the truth value of all atomic propositions appearing in the given **TNL** formula. Thus, the **TNL** model checking problem is reduced to the **LTL** model checking problem, for which an algorithm can be found in Chapter 6.

Basically, we use a set of inequalities to represent a number of different clock functions. By an *inequality* we mean any string of the form "$x-y \sim c$", where x and y are from a designated set of variables, $c \in \mathbf{Q}$ and $\sim$ is a relation symbol from $\{\leq, <, =, >, \geq\}$. If I is a set of inequalities, then $var(I)$ denotes the set of variables that I contains; we say that I is a set of inequalities *over* $var(I)$.

Let I be a set of inequalities over $\{x_1, x_2, \ldots, x_n\}$. A *feasible vector* for I is a tuple $(c_1, c_2, \ldots, c_n)$ of constants $c_i \in \mathbf{Q}$, such that every inequality obtained by replacing every x_i by c_i $(1 \leq i \leq n)$ in any inequality from I holds in the theory of rational numbers. The *solution set* of I is the set of feasible vectors for I. A set of inequalities is *consistent* if its solution set is nonempty. Two sets of inequalities are *isomorphic*, if they have the same solution set.

The *closure* of a **TNL**–formula φ, denoted by $Cl(\varphi)$, is the smallest set of inequalities such that for every inequality "$x - y \leq c$" appearing in φ, both "$x - y \leq c$"$\in Cl(\varphi)$ and "$x - y > c$"$\in Cl(\varphi)$. A *maximal consistent set* of

φ is a maximal set $F \subseteq Cl(\varphi)$ of inequalities which is consistent. Given any set I of inequalities, a *complete extension* $\hat{I}$ of I and φ is any consistent set $\hat{I} = I \cup I'$, such that I' is a maximal consistent set of φ. $CE(I, \varphi)$ denotes the set of all complete extensions of I and φ. Note that for consistent I, $CE(I, \varphi)$ is nonempty and finite.

In the previous section, time variables representing times when the corresponding places got or lost its latest token were introduced. In order to grasp the future behavior of the net, we introduce another sort of time variables, called *transition variables*, representing the next firing time of (enabled) transitions. Since there is no confusion, we use the set T to denote transition variables as well as transitions; all inequalities in this section will therefore use variables from $V = \mathcal{P} \cup T$. $\mathcal{P}_\varphi$ denotes the set of time variables appearing in φ.

An *atom* is a pair $\alpha = (\mu, I)$, where μ is a marking and I is a set of inequalities. The *initial* atom is $\alpha_0 = (\mu_0, \widehat{I_0})$, where μ_0 is the initial marking of the net, and $\widehat{I_0}$ is the unique complete extension of the following set I_0 of inequalities:

$$\begin{aligned} I_0 = \quad & \{\text{“}x - y = 0\text{”} \mid x, y \in \mathcal{P}\} \cup \\ & \{\text{“}t - x \geq \mathsf{Eft}(t)\text{”} \mid t \in \mathsf{enabled}(\mu_0), x \in \mathcal{P}\} \cup \\ & \{\text{“}t - x \leq \mathsf{Lft}(t)\text{”} \mid t \in \mathsf{enabled}(\mu_0), x \in \mathcal{P}\} \end{aligned}$$

The first line defines the initial values of all time variables to be equal. The second and third line give the timing constraints on the next firing of transitions enabled in the initial marking.

We are now going to describe how the set of successor atoms α' of an atom α can be computed. To this end we need the notion of *deletion* of a set U of variables from a set I of inequalities. For every such I and U there exists an (up to isomorphism) unique set $I' = (I \setminus\setminus U)$ of inequalities over $var(I) \setminus U$, such that the solution set of I' is equal to the solution set of I, projected on $var(I) \setminus U$. For example, if $I = \{\text{“}y - x \geq 2\text{”}, \text{“}y - x \leq 7\text{”}, \text{“}y - z < 3\text{”}, \text{“}z - y \leq 11\text{”}\}$, then $(I \setminus\setminus \{y\}) = \{\text{“}x - z < 1\text{”}, \text{“}z - x \leq 18\text{”}\}$. As shown in [JM87], I' can be computed by a graph–based algorithm in time $O(|var(I)|^3)$.

If $\alpha = (\mu, I)$ is an atom, then $\mathsf{firable}(\alpha) = \{t_f \mid t_f \in \mathsf{enabled}(\mu), I \cup \{\text{“}t - t_f \geq 0\text{”} \mid t \in \mathsf{enabled}(\mu)\}$ is consistent$\}$ is the set of transitions that can fire earlier than all other transitions in the given marking and timing properties. Let t_f be a transition in $\mathsf{firable}(\alpha)$, $\mu' = (\mu - {\bullet t_f}) \cup t_f{\bullet}$, and $U_f = \{p^\circ \mid p \in {\bullet t_f}\} \cup \{p^\bullet \mid p \in t_f{\bullet}\}$. We define the following sets of inequalities:

- $J_1 = I \cup \{\text{“}t - t_f \geq 0\text{”} \mid t \in \mathsf{enabled}(\mu)\}$
- $J_2 = (J_1 \setminus\setminus U_f)$
- $J_3 = J_2 \cup \{\text{“}x - t_f = 0\text{”} \mid x \in U_f\}$
- $J_4 = (J_3 \setminus\setminus \{t \mid t \notin \mathsf{enabled}(\mu')\})$
- $J_5 = J_4 \cup \{\text{“}t - x \geq \mathsf{Eft}(t)\text{”} \mid t \in \mathsf{enabled}(\mu'),\ t \notin \mathsf{enabled}(\mu),\ x \in U_f\}$
  $\cup \{\text{“}t - x \leq \mathsf{Lft}(t)\text{”} \mid t \in \mathsf{enabled}(\mu'),\ t \notin \mathsf{enabled}(\mu),\ x \in U_f\}$
- $J_6 = (J_5 \setminus\setminus \mathcal{P} \setminus \mathcal{P}_\varphi)$

Intuitively, this can be read as follows: J_1 describes that t_f fires first, i.e. earlier than other enabled transitions. J_2 is obtained from J_1 by eliminating all time variables U_f which have to be updated. This updating is then done in J_3 by fixing the value of these variables to be equal to the firing time of t_f. In J_4 the transition variables of disabled transitions are deleted. J_5 relates the transition variables of newly enabled transitions to the updated time variables. Finally, all irrelevant time variables are removed. Note that our definition of the J_i's contains some redundancies; e.g. J_6 can be computed by using the operation $(I \setminus\setminus U)$ only once. For any α and t_f, J_6 is uniquely determined (up to isomorphism); we say J_6 is *obtained by firing t_f from α*. $\alpha' = (\mu', I')$ is a *successor atom* of α, if $I' \in CE(J_6, \varphi)$ for some J_6 obtained by firing an enabled transition t_f from α. An *atom sequence* ϱ is a finite or infinite sequence $\varrho = (\alpha_0, \alpha_1, \alpha_2, \ldots)$, such that α_0 is the initial atom and α_{i+1} is a successor atom of α_i for any $i \geq 0$. The *atom graph* $G_\alpha(N, \varphi)$ consists of all atoms reachable by a finite atom sequence.

Given any atom sequence ϱ, satisfaction of φ in ϱ ($\varrho||\!\!-\varphi$) is defined in an obvious way. Moreover, it can be proved that for any run ρ there exists an atom sequence ϱ such that $\rho \models \varphi$ iff $\varrho||\!\!-\varphi$ and vice versa. Thus the question of whether there exists a run of N satisfying φ can be reduced to the question of whether there exists a satisfying atom sequence.

If φ contains no time variables, then $G_\alpha(N, \varphi)$ is finite as shown in [BD91]. Otherwise, however, an infinite number of different atoms may be reachable from the initial atom, because the difference $x - y$ between some time variables may become arbitrarily large, e.g. $\alpha_1 = (\mu, I \cup \{\text{“}x - y > 5\text{”}\})$, $\alpha_2 = (\mu, I \cup \{\text{“}x - y > 17\text{”}\})$, $\alpha_3 = (\mu, I \cup \{\text{“}x - y > 99\text{”}\})$, and so on. In this case, however, every atomic proposition $x - y \leq c$ and $y - x \leq c$ will eventually become constantly false and true, respectively, and thus all α_i in which $x - y$ surpasses a certain threshold value can be considered to be equivalent.

Let *max_const* be the absolute value of the maximal constant appearing in any subformula of φ, and let I be a set of inequalities. A time variable $x \in \mathcal{P}_\varphi$ is called *saturated in* I, if there is no transition variable $t \in var(I)$ such that the set $I \cup \{\text{“}t - x \leq max\_const\text{”}\}$ is consistent. For any two atoms $\alpha_1 = (\mu, I_1)$ and $\alpha_2 = (\mu, I_2)$, let $D = \{x \mid x \text{ is saturated in } I_1 \text{ and } I_2\}$. α_1 and α_2 are *equivalent*, denoted by $\alpha_1 \simeq \alpha_2$, if $I_1 \cap Cl(\varphi) = I_2 \cap Cl(\varphi)$ and $(I_1 \setminus\setminus D) = (I_2 \setminus\setminus D)$, that is, if the same maximal consistent set of φ is a subset of both I_1 and I_2 and the timing relations of I_1 and I_2 with respect to unsaturated variables are isomorphic.

From these definitions we prove in [YS97], using similar techniques as in [ACD90]:

Theorem 8.1 *1. $\simeq$ is a bisimulation; that is, for any α_1 and α_2 such that $\alpha_1 \simeq \alpha_2$, and for any α_1' which is a successor of α_1 there exists a successor α_2' of α_2 such that $\alpha_1' \simeq \alpha_2'$.*

2. $\simeq$ is an equivalence relation of finite index.

Therefore, there exists a finite set G of representative atoms such that for any atom α reachable from the initial atom there is an equivalent atom $\alpha' \in G$, and for any atom sequence $\varrho_1 = (\alpha_0, \alpha_1, \alpha_2, \ldots)$ there is a corresponding sequence $\varrho_2 = (\alpha_0', \alpha_1', \alpha_2', \ldots)$ in G such that $\alpha_i \simeq \alpha_i'$ $(i \geq 0)$ and thus $\varrho_1 || \!\!- \varphi$ iff $\varrho_2 || \!\!- \varphi$. G is constructed by depth–first–search from the initial atom, where the equivalence of atoms can be checked efficiently using hash–tables. Note, however, that in general the size of G is exponential in the size of the net.

Now, model checking can be performed by building the product of G with the set of all sets γ of subformulas of φ, eliminating from this product all pairs (α, γ) inconsistent with φ, and decomposing the resulting graph into maximal strongly connected components. φ is satisfiable by N iff there is a self–fulfilling strong component, i.e. one which contains with any pair (α_1, γ_1) and any formula $(\varphi_1 \ \mathbf{U}^* \ \varphi_2) \in \gamma_1$ also an pair (α_2, γ_2) such that $\varphi_2 \in \gamma_2$.

8.2 Stubborn Analysis

In this section we show how to reduce the size of the atom graph of a given net and formula without affecting the correctness of the model checking procedure. The reduced state space is obtained by considering a coarser equivalence on atoms than the one defined in the previous section. It satisfies the requirement that for any atom sequence in G there exists a stuttering

equivalent (w.r.t. atomic propositions in φ) atom sequence in the reduced state space, and vice versa.

Given any atom α_0 and transitions t, t' enabled in α_0, we say that t is *independent from* t' with respect to α and φ, if for any atom sequence $\varrho' = (\alpha_0, \alpha'_1, \alpha'_2, \ldots)$ such that α'_1 is obtained by firing t' from α_0 there exists a stuttering equivalent (w.r.t. atomic propositions in φ) atom sequence $\varrho = (\alpha_0, \alpha_1, \alpha_2, \ldots)$ such that α_1 is obtained by firing t from α_0. Otherwise, t is called *dependent on* t' (w.r.t. α, φ). Note that this relation is *not* symmetric!

If t is independent from t' we do not have to consider the firing of t' when generating the successors of α in the depth–first–search; there will be a stuttering equivalent sequence constructed by the firing of t.

However, the above definition is not effective; there is no efficient way to compute the set of independent transitions for a given t and α. Therefore, subsequently we give an algorithm to compute an approximation, that is a set $\mathit{dependent}(t, \alpha, \varphi)$, or $\mathit{dependent}(t, \alpha)$ in short, such that t is independent from all transitions not in this set. This idea is similar to the *stubborn set theory* of [Val90, ETV97, KV00] and the *interleaving set temporal logic* of [KP88]

Of course, $\mathit{dependent}(t, \alpha)$ should be as small as possible. For example, if the net N consists of two unconnected subnets N_1 and N_2, and φ mentions only places from N_1, then certainly all transition in N_1 are independent from any transition in N_2. E.g, we don't have to consider the different interleavings of t_2 with t_3, t_4 and t_7 in our example net N_x (shown in Fig. 7.1) for the formulas given at the end of Sect. 7.3

On the other hand, if for some t, t' which are in *conflict* (i.e. $\bullet t \cap \bullet t' \neq \emptyset$), both t and t' are firable in α, then the firing of t' inhibits that of t; thus t is not independent from t', and t' should be in the dependent set of t. So, in N_x, for every firing of t_4 also the alternative of firing t_7 should be considered.

Furthermore, disabled conflicting transitions t' can inhibit the firing of t if they can become enabled by the firing of other (enabled) transitions, and the firing of t' can overlap with that of t. In the example, although t_6 (in conflict with t_3) is disabled, it may inhibit the firing of t_3, since it can become enabled by the firing of t_4 and t_5. However, t_6 will not inhibit the firing of t_3 if t_6 becomes enabled too late. Thus, the dependency relation has to respect the timing in the net. This can be checked by examining the minimal time difference between the next firing times of t_4 and t_3. It takes at least $\mathsf{Eft}(t_5) + \mathsf{Eft}(t_6)$ $(= 7)$ time units to fire t_6 after the firing of t_4. Thus, t_4 can only inhibit the firing of t_3, if t_4 can fire 7 time units earlier than t_3. Hence, we include t_4 in the dependent set of t_3 only if

$I \cup \{\text{“}t_3 - t_4 \geq \mathit{diff}(t_4, t_6)\text{”}\}$ is consistent, where $\mathit{diff}(t, t')$ is the minimal value of sums of earliest firing times in the paths from t to t' ($\mathsf{Eft}(t)$ is not included).

Given any atom $\alpha = (\mu, I)$, firable transition t_f and disabled transition t, we therefore have to find a set of firable transitions such that the firing of any transition in this set could make t fire before t_f fires, and that the firing of t is preceded by the firing of at least one transition in this set. A set Υ of transitions is *necessary* for t, if $\Upsilon = \{t' \mid p \in t'\bullet\}$ for some $p \in \bullet t \setminus \mu$. $\mathit{necessary}^*(t, \alpha)$ is any set of transitions containing t which is transitively closed under necessity, that is, for any $t' \in \mathit{necessary}^*(t, \alpha)$ such that t' is disabled in μ there exists a set Υ of transitions necessary for t' with $\Upsilon \subseteq \mathit{necessary}^*(t, \alpha)$. For example, $\mathit{necessary}^*(t_6, \alpha_0) = \{t_6, t_5, t_4\}$ in Fig. 7.1.

A transition t_h in $\mathit{necessary}^*(t, \alpha)$ is *harmful for* t_f, if it is firable, and $I \cup \{\text{“}t_f - t_h \geq \mathit{diff}(t_h, t)\text{”}\}$ is consistent. If t is in conflict with t_f, then all harmful transitions for t_f in $\mathit{necessary}^*(t, \alpha)$ have to be fired as alternatives to the firing of t_f. The only transition which is harmful for t_3 in our above example is t_4.

There is still another class of dependent transitions. We want to obtain stuttering equivalence with respect to the atomic propositions of φ. Usually, φ contains only a few propositional and time variables. A transition t is *visible for* φ if $\bullet t \cup t\bullet$ contains any place p such that p or $p^\bullet$ or p° appears in φ. If t is visible, the firing order with other visible transitions is important. For example, both t_2 and t_3 are visible for the formula $(p_1\ \mathbf{U}^*\ p_3)$ in the example net, thus the firing order between t_2 and t_3 is relevant for the evaluation of $(p_1\ \mathbf{U}^*\ p_3)$, and t_2 should be in the dependent set of t_3, and vice versa. A visible transition can be regarded as being in conflict with all other visible transitions. Let $\mathit{conflict}^+(t)$ be the set $\{t' \mid \bullet t' \cap \bullet t \neq \emptyset\} \cup \{t\}$, if t is not visible, else $\mathit{conflict}^+(t)$ is $\{t' \mid \bullet t' \cap \bullet t \neq \emptyset\} \cup \{t' \mid t' \text{ is visible }\}$. Then $\mathit{dependent}(t_f, \alpha)$ is any set of transitions such that for every $t \in \mathit{conflict}^+(t_f)$ there exists a set $\mathit{necessary}^*(t, \alpha)$ such that all harmful transitions for t_f in $\mathit{necessary}^*(t, \alpha)$ are contained in $\mathit{dependent}(t_f, \alpha)$.

Finally, the set of transitions which are fired should be transitively closed under dependency; e.g., in our example, since t_4 is in the dependent set of t_3 and t_7 is in the dependent set of t_4, we have to fire t_7 as an alternative whenever we fire t_3 (p_{10} is only reachable by *first* firing t_7 and *then* t_4). Thus, let $\mathit{ready}(\alpha)$ be a smallest set of firable transitions, such that for any $t_f \in \mathit{ready}(\alpha)$, $\mathit{dependent}(t_f, \alpha) \subseteq \mathit{ready}(\alpha)$.

Now, we can prove:

Theorem 8.2 *For any atom sequence $\varrho \in G$ there exists a stuttering equivalent atom sequence $\varrho' = (\alpha_0, \alpha_1, \alpha_2, \ldots)$ such that for any $i \geq 0$, α_{i+1} is obtained by firing some transition in ready(α_i).*

Again, the full proof can be found in [YS97].

Thus, during the construction of the set of successor atoms of an atom we can neglect all firable transitions which are not ready. This results in a considerable average case reduction: For example, in Fig. 7.1, $\mathsf{firable}(\alpha_0) = \{t_2, t_3, t_4, t_7\}$, whereas $\mathit{ready}(\alpha_0) = \{t_2\}$.

We now give a formal description for constructing the reduced atom graph. The algorithm is an adaption of the untimed one given above.

Let $\alpha = (\mu, I)$ be an atom, t_f a transition in $\mathsf{firable}(\mu)$, $\mu' = (\mu - {\bullet t_f}) \cup t_f{\bullet}$, and $U_f = \{p^\circ \mid p \in {\bullet t_f}\} \cup \{p^\bullet \mid p \in t_f{\bullet}\}$. We define the following sets:

- $J_1 = I \cup \{\text{“}t - t_f \geq 0\text{”} \mid t \in \mathit{ready}(\alpha)\}$
- $J_2 = (J_1 \setminus\setminus U_f)$
- $J_3 = J_2 \cup \{\text{“}x - t_f = 0\text{”} \mid x \in U_f\}$
- $J_4 = (J_3 \setminus\setminus \{t \mid t \notin \mathsf{enabled}(\mu')\})$
- For $t \in T$ and $x \in \mathcal{P}$,
  $T_1(t, x) = \{\text{“}t - x \geq \mathsf{Eft}(t)\text{”}, \text{“}t - x \leq \mathsf{Lft}(t)\text{”}\} \cup \{\text{“}y^\bullet \leq x\text{”} \mid y \in {\bullet t}\}$
- $p \in P$ is called a *candidate of true parents* of a transition t_c in μ and I, if $p \in \mu \cap {\bullet t_c}$ and for some enabled transition t in μ, $I \cup \{\text{“}p^\bullet \geq t + \mathit{diff}(t, t_c) - \mathsf{Eft}(t_c)\text{”}\}$ is consistent.
- $T_2 = \prod_{t \in enabled(\mu'), t \notin enabled(\mu)} \{T_1(t, p^\bullet) \mid p$ is a candidate of true parents of t in μ' and $J_4\}$, where $\prod$ represents the Cartesian product.
- $J_5 = \{\hat{I} \mid \hat{I} = J_4 \cup a_1 \cup a_2 \cup \cdots \cup a_i, (a_1, a_2, \cdots, a_i) \in T_2,\ \hat{I}$ is consistent$\}$
- $J_6 = \{(\hat{I} \setminus\setminus \mathcal{P} \setminus \mathcal{P}_\varphi \setminus D) \mid \hat{I} \in J_5\}$, where $D = \{p^\bullet \mid p$ is a candidate of true parents of some disabled transition in μ' and $\hat{I}\}$

$\alpha' = (\mu', I')$ is a *successor atom* of α, if $I' \in \bigcup_{\hat{I} \in J_6} CE(\hat{I}, \varphi)$. The finite reduced atom graph G is constructed in the same way as shown in the end of Sect. 8.1.

Though the worst case complexity of the construction of the set $\mathit{ready}(\alpha)$ is $O(|P| \cdot |T|^2)$, usually this takes only $O(|T|)$ steps with a small constant of about two or three.

We have implemented both the basic model checking algorithm and its partial order improvement on a 17 MIPS UNIX workstation in C++. In this section, the performance of both algorithms with an example from [RB86] is demonstrated.

The verified system called PROWAY is a local area network linking stations by a shared hardware bus. The bus allocation procedure is based on a token bus access technique. Fig. 8.1 shows a Time Petri net model for station 1 of the PROWAY system in a four-station configuration.

Stations are logically distributed on a ring, and a baton goes around on the ring. When a station has the baton, it can transmit application messages, whereas the other stations can only listen to them. A token in p_1 means that the station is in the listening mode. A token in p_3 means that the station has a baton. If transition t_4 fires, the station first transmits application messages and then it passes a baton to the next station on the logical ring. Otherwise, the station only passes a baton without message transmission. On the transmission of messages, the station holds a baton for a longer time. (Compare firing intervals associated with t_9 and t_{10} in Table 8.1).

Each station has a recovery mechanism against a single fault. A station sets its frame interval timer $T1$ (represented by t_{17}) when it transmits a baton. If any activity on the bus (i.e., baton or message transmission from other stations) is listened a certain time later, the station gets into listening mode, resetting the timer. Otherwise, the frame interval timer times out. Suppose the station S_a transmits a baton to the station S_b. Time-out of the S_a's frame interval timer occurs when (i) a baton from S_a is lost, (ii) S_b is faulty, or (iii) the baton or messages from S_b are lost. In these cases, S_a transmits a new baton to another station S_c. Next time S_a has a baton, S_a tries to transmit the baton to S_b. If $T1$ of S_a times out again, S_a will ignore S_b from now on. p_8, p_9 and p_{10} represent how many times this time-out of $T1$ occurs.

A station sets its lost baton timer $T2$ (represented by t_2) when it gets into listening mode. The purpose of this timer is to initiate a new baton when a baton holder goes faulty, holding the current baton, and all other live stations are in the listening mode. The value of $T2$ is indexed with the station's address as shown in Table 8.1, in such a way that the live station with the smallest address monitors the recovery.

As example property, we verify if the next activity will always occur within some constant time units, say max, after a station finishes sending its message. This property holds in the system if the **TNL** formula $\neg\ \mathbf{G}^*\ [\mathit{finish} \rightarrow (\neg \mathit{activity})\ \mathbf{U}^*\ (\mathit{activity}^\bullet - \mathit{finish}^\bullet \leq \mathit{max})]$ is not satisfiable.

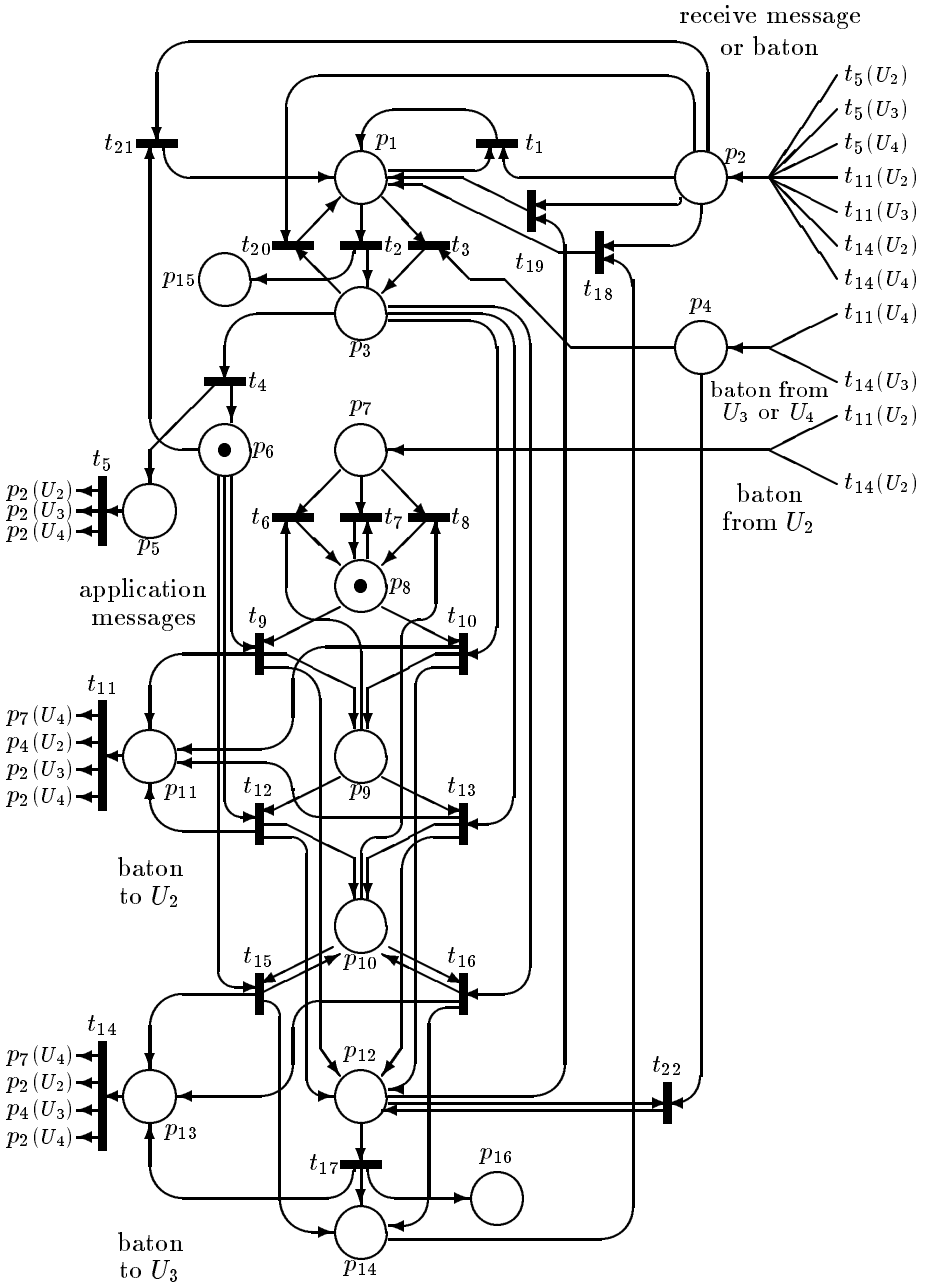
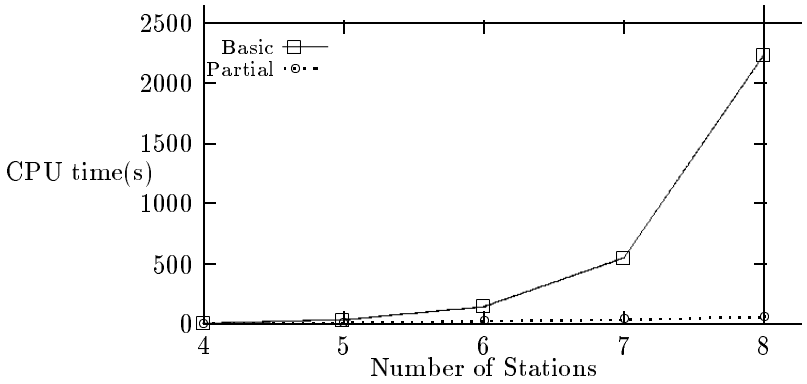


Figure 8.1: A time Petri net model for station 1 (U_1) of the PROWAY system in a four-station configuration.

Table 8.1: Timing constraints for transitions ($TC1$).

t_1	[0,0]	t_7	[0,0]	t_{13}	[16,24]	t_{19}	[0,0]
t_2	$[260, 300]^\dagger$	t_8	[0,0]	t_{14}	[0,10]	t_{20}	[0,0]
t_3	[0,0]	t_9	[50,100]	t_{15}	[50,100]	t_{21}	[0,0]
t_4	[16,24]	t_{10}	[16,24]	t_{16}	[16,24]	t_{22}	[0,0]
t_5	[0,10]	t_{11}	[0,10]	t_{17}	[50,53]		
t_6	[0,0]	t_{12}	[50,100]	t_{18}	[0,0]		

† [80i+180,80i+220] for station i



The Figure shows the CPU times for both implemented algorithms with this example. The size of the net is linear in the number n of stations; thus the basic algorithm is exponential in n. Since all stations operate more or less independently, parallelism also increases with n; therefore, the partial order method succeeds in reducing the complexity. This result is typical for a number of similar examples.

Chapter 9

Verification with Timed Traces

This chapter deals with a first industrial application of partial state-space analysis: Verification of asynchronous circuits. The context is the development of a complete asynchronous processor at TITech [TKI+97, Yon99]. Whereas this chapter presents a concrete algorithm used for the actual verification of parts of this processor, the next chapter reconsiders these techniques from a more theoretical perspective and compares different alternative definitions.

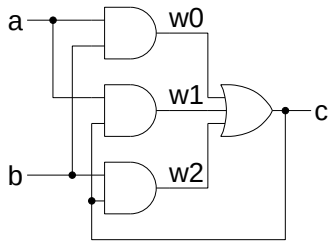
One of the main problems in the design of wafer-scale integrated circuits is the distribution of the global clock signal. Difficulties which arise in the design of large synchronous circuits are clock skews, clock delay estimation in layout design, etc. Therefore, *asynchronous processors* without a global clock are of increasing interest. However, asynchronous circuits are difficult to construct since the timing analysis often is very complex. Because of this reason, asynchronous circuits are usually modelled with a *speed independent model*, where the gate delays are unbounded, or are bounded by an unknown constant. Most of the research on design, synthesis, and verification of asynchronous circuits has been done under this model. Although the speed independent model is quite powerful, the possibility of unbounded delay can force the designer to add additional complexity to the circuit. For example, Muller's C element [MB59], defined by the truth table in Fig. 9(a), is implemented by the circuit of Fig. 9(b).

This implementation, however, is not correct under the speed independent model. Assuming that each gate can have an unbounded delay, there exists a signal transition sequence in which the output illegally goes down

a	b	c
0	0	0
0	1	P
1	0	P
1	1	1

P : previous value

(a)



(b)

Figure 9.1: Muller's C element: truth table and gate-level implementation.

before both inputs go down (suppose all wires initially have the value 0) :

$$a\uparrow\ b\uparrow\ w_0\uparrow\ c\uparrow\ b\downarrow\ w_0\downarrow\ c\downarrow .$$

The reason for this alleged fault is an extremely large delay of the gate with output w_1. With any well-processed VLSI, such a large delay should be impossible. In actual designs, the given circuit can be safely used to implement a C element. Thus, the speed independent model sometimes is not appropriate. In this paper we use a *bounded delay model* to model asynchronous circuits, where with each gate a lower and upper bound for the delay is associated.

In [Dil89b], an efficient verification method for speed independent circuits was proposed, which is based on *trace theory*. (Recall that a trace is the projection of an ω-word onto an observable alphabet.) The primary advantage of this method is the possibility of hierarchical verification, which greatly reduces the complexity of the verification procedure. However, this method is only suited for verifying safety properties.

Here, we adapt Dill's verification method to the bounded delay model. We show how trace theory can be extended to handle timed traces as well as certain timing requirements. Subsequently, we derive an algorithm to check whether an implementation, consisting of a set of I/O-modules, meets its specification. Then, we give some experimental results.

9.1 Timed Trace Theory

Let us briefly describe verification based on trace theory. To this end, we urge the reader to recall the notions of $I/O-automata$ and $I/O-modules$ from Section1.2.3. In Dill's method, the specification of a circuit is given

as an *I/O-automaton*, i.e. a finite automaton over an input alphabet Σ^{in} and output alphabet Σ^{out}. The implementation, which is supposed to be a set of I/O-modules, is given as a set of I/O-automata, each one representing the behavior of its related I/O-module. Then, special *composition* and *hiding* operations on I/O-automata are defined. The implementation *conforms* to the specification, if they agree on the input and output alphabets, respectively, and the implementation can be safely substituted for the specification in every context. This means, that the implementation causes a failure in an environment only if the specification also causes a failure in that environment.

A *failure* of an I/O-module in an environment is an output of the I/O-module which is not accepted by the environment, or an output of the environment which is not accepted by the I/O-module. By this definition, conformance can be expanded to the following requirements: the implementation should be able to handle every input that the specification can handle, and it never produces an output unless the specification can produce it. This in turn can be checked by considering the mirror of the specification, where all inputs are outputs and vice versa. The implementation conforms to the specification iff the result of hiding all internal signal transitions in the implementation and composing it with the mirror of the specification is failure-free.

The verification approach proposed here is the timed version of this method, where time Petri nets and timed ω-words are used instead of automata and ω-words. The extension to real-time makes it also possible to verify certain timing properties.

In the rest of this section, we define timed traces and their related notions, and the conformance relation between specification and implementation.

Let again Σ be the alphabet, and $\mathbf{T}$ the time domain: the set of non-negative rational numbers. Recall that for any $a \in \Sigma$ and $\mathsf{clock} \in \mathbf{T}$, the tuple (a, τ) is called an *event*. In an asynchronous circuit, (a, τ) represents the change of the value of wire a at time τ.

A *(timed) ω-word* σ over Σ is defined as a finite or infinite sequence of events $\sigma \triangleq e_1 e_2 \cdots$, where $e_i \triangleq (\sigma_i, \tau_i)$, such that the following properties are satisfied:

- Monotonicity: for all $0 < i < |\sigma|$, $\tau_i \leq \tau_{i+1}$.
- Progress: if σ is infinite, then for every $\tau \in \mathbf{T}$ there exists an index i such that $\tau_i > \tau$.

In this definition, $|x|$ denotes the length of ω-word x. If $|x| = 0$, then x is the *empty word* ε. For any finite ω-word x, ω-word y, and event e, the result of *appending* e or y to x is denoted by $x \circ e$ or $x \circ y$, respectively. x is a *prefix* of y if $y = x$ or $y = x \circ z$ for some ω-word z.

Recall that a trace is the projection of a (timed) ω-word onto the language of some transition system. Thus, the traces of an I/O-module can be regarded as the set of all maximal execution sequences of some transition system. However, trace structures are insensitive to nondeterminism; they can not distinguish between $a \circ (b + c)$ and $(a \circ b) + (a \circ c)$. In timed systems, usually the set of traces will be an infinite (or even uncountable) set of infinite sequences.

Now we consider the composition of several I/O-modules. Assume we are given a set $\mathcal{M} \triangleq \{M_1, \cdots, M_n\}$ of I/O-modules, where $M_k \triangleq (\Sigma_k^{in}, \Sigma_k^{out}, T_k)$, $\Sigma_k \triangleq \Sigma_k^{in} \cup \Sigma_k^{out}$, and $\Sigma_j^{out} \cap \Sigma_k^{out} = \emptyset$. That is, each wire is either an input, output, or both; in the latter case we say the wire is *internal*. Any wire can be an output of at most one I/O-module, and input of arbitrary many I/O-modules. Intuitively, I/O-modules are composed by soldering wires with the same name together. Output wires of one I/O-module are connected to input wires of other I/O-modules. However, in some cases this connection of wires may cause failures in the composed I/O-module.

A *safety failure* of $\mathcal{M}$ is any nonempty finite trace $x \triangleq y \circ (\sigma, \tau)$, where $\sigma \in \Sigma_k^{out}$ for some $k \leq n$, such that $\mathcal{M} \setminus \sigma \models x$, and $M_k \models x$, but $\mathcal{M} \not\models x$. Intuitively, a safety failure occurs if any I/O-module M_k tries to send an output, but some other I/O-module cannot receive this as internal input. $\mathcal{M}$ is *safety failure free*, if no safety failure can occur, i.e., if every output which may be produced by some I/O-module can be accepted by all other I/O-modules at the same time. Whenever an I/O-module can change the value on one of its output wires, all I/O-modules which have this wire connected as internal input must be able to process the signal immediately.

A *timing failure* of $\mathcal{M}$ is any nonempty finite trace $x \triangleq y \circ (\sigma, \tau)$, where $\sigma \in \Sigma_k^{in}$ for some $k \leq n$, such that $\mathcal{M} \setminus \sigma \models x$, and $M_k \models x$, but there is no $x' \triangleq y \circ (\sigma', \tau')$, where $\sigma' \in \Sigma_k^{in}$, and $\mathcal{M} \models x'$. Intuitively, a timing failure occurs if some I/O-module M_k expects an internal input from some other I/O-module which is not provided in time. $\mathcal{M}$ is timing failure free if whenever an I/O-module requests a signal on one of its internal input wires, there exists an I/O-module which can produce some signal as output within the required time interval. For any set $\mathcal{M} \triangleq \{M_1, \cdots, M_n\}$ of I/O-modules, $\mathsf{failure}(\mathcal{M})$ is the set of all safety and timing failures of $\mathcal{M}$. $\mathcal{M}$ is *failure-free* if $\mathsf{failure}(\mathcal{M}) = \emptyset$.

Next, we define a conformance relation between a system consist-

ing of a set of I/O-modules and a specification given as a single I/O-module. Consider a set $\mathcal{M}_C \triangleq \{M_1, \cdots, M_n\}$ of I/O-modules, where $M_k \triangleq (\Sigma_k^{in}, \Sigma_k^{out}, T_k)$, and an I/O-module $M_S \triangleq (\Sigma_S^{in}, \Sigma_S^{out}, T_S)$ such that $\Sigma_S^{in} \triangleq \bigcup \Sigma_k^{in} - \bigcup \Sigma_k^{out}$ and $\Sigma_S^{out} \subseteq \bigcup \Sigma_k^{out}$. I/O-module M_S can be thought of as an abstract specification of the concrete system M_C: all external inputs of the system M_C appear as inputs of the specification M_S, and some (but not necessarily all) outputs of the system M_C are visible in the specification M_S.

$\mathcal{M}_C$ *conforms to* M_S, if for any admissible environment $M_E \triangleq (\Sigma_S^{out}, \Sigma_S^{in}, T_E)$, whenever $\{M_S, M_E\}$ is failure-free, also $\mathcal{M}_C \cup \{M_E\}$ is failure-free. In other words, the system $\mathcal{M}_C$ may have a failure in the environment M_E only if the specification M_S allows a failure in the same context. This conformance relation is reflexive and transitive, but not symmetric: The system may be failure-free even in contexts in which the specification fails.

An I/O-module M is called *I/O-conflict free*, if for any trace x, and for all events $e_i \triangleq (\sigma_i, \tau_i)$ and $e_o \triangleq (\sigma_o, \tau_o)$ with $\sigma_i \in \Sigma^{in}$ and $\sigma_o \in \Sigma^{out}$ it holds that $M \models x \circ e_i$ and $M \models x \circ e_o$ implies $M \models x \circ e_i \circ e_o$ and $M \models x \circ e_o \circ e_i$. Since conflicts between inputs and outputs often indicate hazardous situations, specifications usually do not contain such conflicts. For a trace x and I/O-modules $M(= (\Sigma^{in}, \Sigma^{out}, T))$, M_1, and M_2, we say that M *admits* x ($M \models x$), if $\mathsf{project}(x, \Sigma^{in} \cup \Sigma^{out}) \in T$. An I/O-module M *can wait* after trace x *for at least* τ time units, denoted by $M \models\!\!\!\vdash x \circ \tau$, if $M \models x(w, \tau')$ for some $\tau' < \tau$ implies that $M \models x(w, \tau'')$ for some $\tau'' \geq \tau$. This means that after admitting x, I/O-module M can be inactive for τ time units without sending or receiving any events. We say that M is *unbounded*, if for any trace x such that $M \models x$, the I/O-module M can wait after x for an arbitrary amount of time. As we shall see, the unboundedness requirement facilitates the analysis considerably. We will come back to this assumption in the next chapter. Usually, specifications satisfy this additional requirement. Thus, henceforth we assume that all modules which are used as specifications in the verification procedure are unbounded and I/O-conflict free.

Correctness defined by conformance is different from other notions of correctness defined in previous chapters, e.g., trace inclusion, (bi-)simulation or logical implication. A common class of failures is that after a sequence of actions one module in a system produces an internal event but other modules can not accept it. Consider the modules M_{S1} and M_C shown in Fig. 9.2(a). M_C does not conform to M_{S1}, because there exists an I/O-module M_{E1} shown in Fig. 9.2(b) which has no such failure with M_{S1} but has a failure $((b,1)(x,2))$ with M_C (i.e., after receiving an input b at time 1,

M_C wants to produce the output x at time 2, but this can not be accepted by M_{E1}). On the other hand, consider M_{S2} in Fig. 9.2 (c). In this case, M_{S2} is not interested in receiving an input b. Therefore, M_{E1} now has a failure with M_{S2} (i.e., the output b of M_{E1} can not be accepted by M_{S2}). An I/O-module which has no failure with M_{S2} is, for example, M_{E2} in Fig. 9.2(d). It does not have any failures with M_C, either. Actually, any M_E which has no failures with M_{S2} has no failures with M_C as long as $\Sigma_E^{in} \subseteq \Sigma_{S2}^{out}$ and $\Sigma_E^{out} \supseteq \Sigma_{S2}^{in}$ holds. Hence, M_C conforms to M_{S2}. Note that M_C implements the additional behavior "bx" which M_{S2} does not care about. We consider M_C to be correct with respect to M_{S2}, because M_C correctly implements the behavior specified by M_{S2}, even if M_C is not a bisimulation of M_{S2}, and the trace set of M_C is not included in that of M_{S2}. This is an important property in circuit verification, because circuits usually implement some additional functionality for inputs not constrained by the specification.

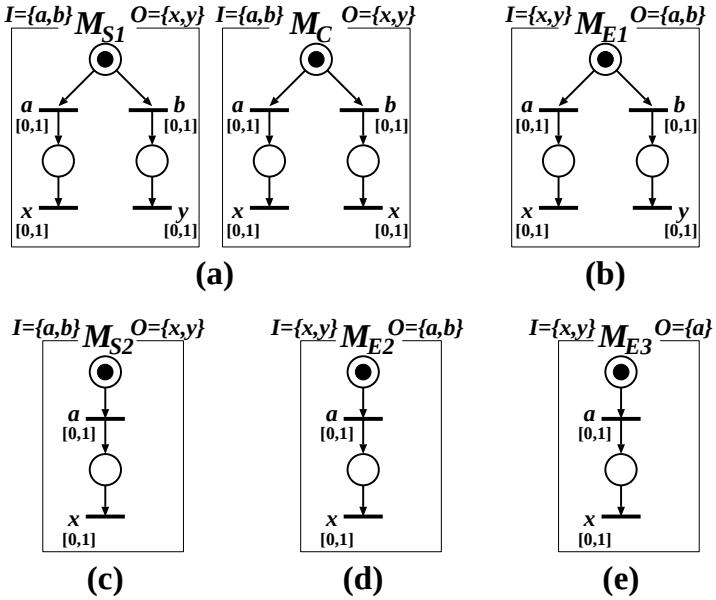


Figure 9.2: Examples of failures.

In the definition of conformance, only admissible environments ($\Sigma_E^{in} \subseteq \Sigma_S^{out}$ and $\Sigma_E^{out} \supseteq \Sigma_S^{in}$) are considered for the following reason. Consider the environment M_{E3} shown in Fig. 9.2(e). Note that M_{E3} does not have b as an output symbol, and, hence, is not admissible. Here, M_{E3} has the failure $((b,0)(x,1))$ with M_C: both M_C and M_{E3} admit the one-element trace $((b,0))$, because M_C accepts it, and M_{E3} ignores b. Thus, after the

trace $((b,0))$, I/O-module M_C may decide to produce an output x at time 1, but the environment M_{E3} can not accept it (note that M_{E3} must produce an a before being ready to receive x). On the other hand, $\{M_{S2}, M_{E3}\}$ is failure-free (M_{S2} does not accept the trace $((b,0))$). We conclude that without admissibility constraints on the environment, M_C would not conform to M_{S2}. In this sense, omitting these constraints in the definition of conformance would restrict the possibility of additional functionality for a correct implementation.

A major benefit of using conformance as a correctness criterion is that the verification can be done hierarchically. This allows us to verify rather large systems, which are built from component libraries in a hierarchical way.

The *mirror I/O-module* M^m of an I/O-module $M \triangleq (\Sigma^{in}, \Sigma^{out}, T)$ is the I/O-module $M^m \triangleq (\Sigma^{out}, \Sigma^{in}, T)$; that is, each input wire in M^m is an output wire of M and vice versa.

For any I/O-module M, the set $\{M, M^m\}$ is failure-free. Moreover, the following *hierarchy lemma* holds:

Lemma 9.1 *Consider three I/O-modules M_1, M_2 and M_3 such that $\Sigma_1^{in} = \Sigma_2^{in} = \Sigma_3^{in}$ and $\Sigma_1^{out} \supseteq \Sigma_2^{out} = \Sigma_3^{out}$. If $\{M_1, M_2^m\}$ is failure-free and $\{M_2, M_3^m\}$ is failure-free, then $\{M_1, M_3^m\}$ is failure-free.*

The proof can be found in [YZS99].

From the hierarchy theorem, the following *mirror theorem* can be obtained. It gives a similar characterization of conformance as in [Dil89b]:

Theorem 9.2 *$\mathcal{M}_C$ conforms to M_S iff $\mathcal{M}_C \cup \{M_S^m\}$ is failure-free.*

PROOF: Assume that $\mathcal{M}_C \cup \{M_S^m\}$ has a failure. Then for the environment $M_E = M_S^m$ we have that $\{M_S, M_E\}$ is failure-free, but $\mathcal{M}_C \cup \{M_E\}$ is not failure-free, i.e., $\mathcal{M}_C$ does not conform to M_S.
In the other direction, we have to show that failure-freeness of $\mathcal{M}_C \cup \{M_S^m\}$ implies that $\mathcal{M}_C$ conforms to M_S. Since M_S is a specification for the system $\mathcal{M}_C$, $\Sigma_S^{in} = \bigcup_k \Sigma_k^{in} - \bigcup_k \Sigma_k^{out}$ and $\Sigma_S^{out} \subseteq \bigcup_k \Sigma_k^{out}$. If $\mathcal{M}_C \cup \{M_S^m\}$ is failure-free, then the hierarchy lemma asserts that for any I/O-module M_E such that $\Sigma_E = \Sigma_S$ and $\{M_S, M_E\}$ is failure-free, $\mathcal{M}_C \cup \{M_E\}$ must also be failure-free. Thus, $\mathcal{M}_C$ conforms to M_S. □

To get an intuitive understanding of the conformance relation, consider the case of a single I/O-module $M_C \triangleq (\Sigma_C^{in}, \Sigma_C^{out}, T_C)$ conforming to $M_S \triangleq (\Sigma_S^{in}, \Sigma_S^{out}, T_S)$. This amounts to $\Sigma_S^{in} = \Sigma_C^{in}$, $\Sigma_S^{out} \subseteq \Sigma_C^{out}$, and for all traces x such that $\{M_C, M_S\} \models x$, and all events $i \triangleq (\sigma_i, \tau_i)$, $\sigma_i \in \Sigma_S^{in}$, and $o \triangleq (\sigma_o, \tau_o)$, $\sigma_o \in \Sigma_S^{out}$, the following holds:

- If $M_S \models x \circ i$, then $M_C \models x \circ i$,
- if $M_C \models x \circ o$, then $M_S \models x \circ o$,
- if $M_S \models x \circ o$, then there exists an $o' \triangleq (\sigma'_o, \tau'_i)$, $\sigma'_o \in \Sigma_S^{out}$ such that $\{M_S, M_C\} \models x \circ o'$, and
- if $M_C \models x \circ i$, then there exists a $i' \triangleq (\sigma'_i, \tau'_i)$, $\sigma'_i \in \Sigma_S^{in}$ such that $\{M_S, M_C\} \models x \circ i'$.

The first and second condition state that $\{M_C, M_S^M\}$ is safety failure-free: every input allowed by M_S is allowed by M_C, and every output allowed by M_C is allowed by M_S. The third condition reflects the definition of timing-failure: as long as M_S^m expects an input, that is, M_S requires an output, M_C should produce some output in time. The fourth condition is similar. If M_C is constructed as an implementation for the specification M_S, then this can be read as:

- The implementation can handle every input that the specification can handle,
- the implementation never produces an output unless the specification produces it,
- if the specification requires an output, the implementation produces it in time, and
- the implementation never expects an input unless the specification expects the input.

Therefore, our definition of the conformance relation includes not only safety properties, but also a certain timing property. In the case of bounded delay asynchronous circuits the absence of timing failure amounts to in-time-responsiveness, which is an important issue for verification. For example, consider the specification of an *or*-gate, where input a or b lead to output c within a certain time. Suppose that this specification is implemented erroneously by an *and*-gate. Then, after sending a to this system, it can not produce the output c. However, since the specification requires such an output, this situation leads to a timing failure.

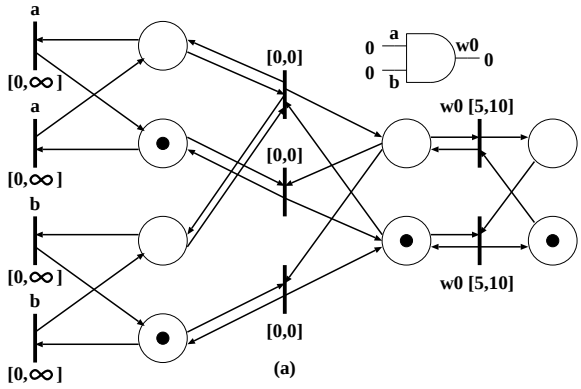
Note that we do not actually compose the I/O-modules constituting the implementation. Therefore, in our approach it is not necessary to eliminate so-called autofailures, which arise from internal communication errors in a composed I/O-module. Also we do not have an explicit hiding operation:

Failures resulting from the effect of hiding variables are transparent to the specification and will also be detected during the verification procedure. However, if we consider only safety-failures in untimed systems, then our notion of conformance is equivalent to the one in [Dil89b].

9.2 Analysis of Time Petri Nets

In the general setting of the previous section, there was absolutely no restriction posed on the set of traces of an I/O-module. To be able to give concrete algorithms, however, this set should at least be recursive, i.e., generated by some kind of automaton. In this section, we consider trace sets generated by one-safe time Petri nets.

Let wire be a function from a set of transitions in a time Petri net to the alphabet representing the wires of the asynchronous circuit. Every maximal run $\rho \triangleq \sigma_0 \stackrel{t_1}{\rightarrow} \sigma_1 \stackrel{t_2}{\rightarrow} \cdots$ of a net N *generates* the timed trace $((\mathsf{wire}(t_1), \mathsf{time}_1(\rho)), (\mathsf{wire}(t_2), \mathsf{time}_2(\rho)), \cdots)$. We also say that a net N *represents* the I/O-module consisting of all traces generated by maximal runs of N.



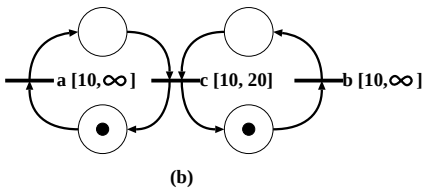


Figure 9.3: Nets specifying AND gate and C element

Bounded delay asynchronous circuits can be easily described by nets. For

example, an *and*-gate which has inputs a, b and an output w_0 with gate delay [5,10] can be represented by the net shown in Fig. 9.3(a). In this modelling, we do not distinguish between the change of a wire from 0 to 1 and from 1 to 0. An *or*-gate can be represented similarly. Even though it would be possible to give a more detailled description of gates (e.g., transistor level behavior), for most verification purposes the given net is an adequate representation.

The composition of several gates in a circuit can be described by simply putting together all nets representing single gates. Assuming that all wires in the circuit have unique names, for each transition the corresponding wire can be assigned. Then, the disjoint union of all these nets represents the complete circuit. Thus, the implementation of Muller's C element shown in Fig. 9(b) can be represented by a collection of nets which are similar to the one in Fig. 9.3(a).

This implementation works correctly under the following assumptions:

1. if an input changes, then the same input never changes again before an output changes, and

2. no input changes before some constant time passes after the change of the output.

The net shown in Fig. 9.3(b) specifies the behavior of a C-element with these assumptions. Verification consists in showing that the gate-level representation conforms to this specification. This is done by exploring the reachable states of the composed net.

We now describe an algorithm to generate these reachable states of time Petri nets. Since for time Petri nets the time domain consists of rational (not real) numbers, the state space can be finitely represented by sets of systems of inequalities. Basically, we use a system of inequalities to represent a number of different clock functions of time Petri nets. By an *inequality* we mean any string of the form "$x - y \sim c$", where x and y are from a designated set of variables, $c \in \mathbf{T}$ and $\sim$ is a relation symbol from $\{\leq, \geq\}$. If I is a set of inequalities, then $\mathsf{var}(I)$ denotes the set of variables that I contains; we say that I is a set of inequalities *over* $\mathsf{var}(I)$. Let I be a set of inequalities over $\{x_1, x_2, \cdots, x_m\}$. A *feasible vector* for I is a tuple $(c_1, c_2, \cdots, c_m)$ of constants $c_i \in \mathbf{T}$, such that every inequality obtained by replacing every x_i by c_i $(1 \leq i \leq m)$ in any inequality from I holds in the theory of rational numbers. The *solution set* of I is the set of feasible vectors for I. A set of inequalities is *consistent* if its solution set is nonempty. Two sets of inequalities are *isomorphic*, if they have the same solution set.

If the net $N \triangleq (P, T, F, \mathsf{Eft}, \mathsf{Lft}, \mu^0)$ represents the I/O-module $M \triangleq (\Sigma^{in}, \Sigma^{out}, T)$, we denote this by $M = (\Sigma^{in}, \Sigma^{out}, N, \mathsf{wire})$. An *abstract state* of the net is a pair (μ, I), where $\mu \subseteq P$ and I is a set of inequalities. Each abstract state denotes an equivalence class of reachable states of the net, namely all states for which the clock values form a feasible vector in the solution set of I. The initial abstract state of N is (μ^0, I_0), where $I_0 \triangleq \{\text{“}\mathsf{Eft}(t) \leq \underline{t} - v \leq \mathsf{Lft}(t)\text{”} \mid t \in \mathsf{enabled}(\mu^0)\}$. Here, $\underline{t}$ in I_0 is a variable to represent the next firing time of the transition t. The variable v indicates the initial time point.

The next step is to compute the set of abstract successor states σ' of an abstract state σ of N. This is similar to the calculation in the model checking algorithm on Page 164 and Page 169. The reader should recall the notion of *deletion* of a set U of variables from a set I of inequalities defined in Chapter 8. Let $\sigma \triangleq (\mu, I)$ be an abstract state of N, and $t_f \in \mathsf{enabled}(\mu)$. Then, $\mathsf{first}(\mu, t_f) \triangleq \{\text{“}\underline{t} - \underline{t}_f \geq 0\text{”} \mid t \in \mathsf{enabled}(\mu)\}$ is a set of inequalities describing that t_f is the first transition which fires in μ. $\mathsf{firable}(\sigma) \triangleq \{t_f \mid t_f \in \mathsf{enabled}(\mu), I \cup \mathsf{first}(\mu, t_f)$ is consistent$\}$ is the set of transitions that can fire earlier than all other transitions in the given marking.

- t_f is a transition in $\mathsf{firable}(\sigma)$.
- μ' is the marking of N obtained by firing transition t_f.
  That is, $\mu' \triangleq (\mu - {\bullet t_f}) \cup t_f \bullet$.
- R is a set of newly enabled transitions obtained by the firing of t_f.
  That is, $R \triangleq \mathsf{enabled}(\mu') - \mathsf{enabled}(\mu - {\bullet t_f})$.
- $J \triangleq \{\text{“}\underline{t} - \underline{t}_{out} \geq \mathsf{Eft}(t)\text{”} \mid t \in R\} \cup \{\text{“}\underline{t} - \underline{t}_{out} \leq \mathsf{Lft}(t)\text{”} \mid t \in R\}$.
- $J' \triangleq I \cup \mathsf{first}(\mu, t_{out}) \cup J$.
- $D \triangleq \{\underline{t} \mid t$ made some transition t' enabled, and t' is still enabled in $\mu'\}$.
- $I' \triangleq (J_3 \setminus\setminus \{\underline{t} \mid t \notin \mathsf{enabled}(\hat{\mu'})\} - D)$

Intuitively, J, J', D and I' can be read as follows: J relates the variables of newly enabled transitions to the variable of the fired transition t_{out}. J' is the union of I, J, and a set of inequalities representing that t_{out} fires earlier than others. Transitions related to variables in D are currently parents of enabled transitions in μ', and these variables are necessary to check the coverability between the firing domains of transitions. Finally, in I' the variables of disabled transitions except for those in D are deleted. We write $\sigma \xrightarrow{t_f} \sigma'$ if $\sigma' \triangleq (\mu', I')$ is a successor of the abstract state $\sigma \triangleq (\mu, I)$ with respect to t_f.

We now describe how conformance can be checked, using this successor relation between abstract states. We consider a set $\{M_0, M_1, \cdots, M_n\}$ of I/O-modules, where $M_i = (\Sigma_i^{in}, \Sigma_i^{out}, N_i, \mathsf{wire}_i)$, $N_i \triangleq (P_i, T_i, Eft_i, Lft_i, \mu_i^0)$, and assume that for $i \neq j$, $P_i \cap P_j = T_i \cap T_j = P_i \cap T_j = \emptyset$. Some I/O-module in the set is a mirror of a specification, and input transitions and output transitions must not be in conflict in the I/O-module. If there is no confusion, we use the notation wire instead of wire_i, and $t \in M_i$, when $t \in T_i$. Let $m(t)$ be the I/O-module number of t, i.e., $m(t) = i$, if $t \in M_i$. Transition t is called an *output transition* if $\mathsf{wire}_{m(t)}(t) \in \Sigma_{m(t)}^{out}$, and an *input transition* if $\mathsf{wire}_{m(t)}(t) \in \Sigma_{m(t)}^{in}$. If $\sigma_i \triangleq (\mu_i, I_i)$, $i \leq n$, are abstract states of the nets N_i, and K is a set of inequalities, we say that $s \triangleq (\sigma_0, \cdots, \sigma_n, K)$ is an abstract state of the I/O-module set $\{M_0, M_1, \cdots, M_n\}$.

The initial abstract state is $s_0 \triangleq (\sigma_0^0, \cdots, \sigma_n^0, \emptyset)$. We extend the definitions of $enabled(\mu)$ and $firable(\sigma)$ with respect to $s \triangleq (\sigma_0, \cdots, \sigma_n, K)$ as follows.

$$enabled(s) \triangleq \{t \mid t \in enabled(\mu_{m(t)})\}, \text{ and}$$

$$globally\_firable(s) \triangleq \{t \mid t \in enabled(s), \quad first(s,t) \cup \bigcup_{i=0}^{n} I_i \cup K \text{ is consistent}\},$$

where $first(s,t) \triangleq \{\text{“}t - t' \leq 0\text{”} \mid t' \in enabled(s)\}$. Furthermore, for an output transition t_O such that $t_O \in globally\_firable(s)$,

$$sync\_trans(t_O, s) \triangleq \{t \mid \mathsf{wire}(t) = \mathsf{wire}(t_O), t \in globally\_firable(s)\}.$$

When $\{M_0, M_1, \cdots, M_n\}$ is at $s \triangleq (\sigma_0, \cdots, \sigma_n, K)$, it moves to $s' \triangleq (\sigma_0', \cdots, \sigma_n', K')$ with respect to $t_O \in globally\_firable(s)$ by firing all transitions in $sync\_trans(t_O, s)$.

- for $1 \leq i \leq n$
  - if $t \in sync\_trans(t_O, s) \cap T_i$, then $\sigma_i \xrightarrow{t} \sigma_i'$, and
  - if $sync\_trans(t_O, s) \cap T_i = \emptyset$, then $\sigma_i' = \sigma_i$.
- $K' \triangleq K \cup \{\text{“}\underline{t} = \underline{t'}\text{”} \mid t, t' \in sync\_trans(t_O, s)\}$.

Let $s \xrightarrow{t_O} s'$ denote this state transition relation of the I/O-module set. For any transition t and abstract state σ, the variable $parent(t, \sigma)$ indicates which transition enabled t. Formally, if $\sigma \triangleq (\mu, I)$, $\sigma' \triangleq (\mu', I')$, $\sigma \xrightarrow{t} \sigma'$, and $t' \in enabled(\sigma')$, then

$$parent(t', \sigma') \triangleq \begin{cases} \underline{t}, & \text{if } t' \in enabled(\mu') - enabled(\mu - \bullet t) \\ parent(t', \sigma), & \text{otherwise.} \end{cases}$$

For a set I of inequalities, let $earlier(x, y, I)$ be the predicate expressing that $solution(\{\text{“}x > y\text{”}\} \cup I) = \emptyset$, i.e., $earlier(x, y, I)$ holds iff $x \leq y$ for every solution vector of I. We write $earlier(x, y, \sigma_i)$ for $earlier(x, y, I_i)$, where $\sigma_i \triangleq (\mu_i, I_i)$, and $earlier(x, y, s)$ for $earlier(x, y, \bigcup_{i=0}^{n} I_i \cup K)$, where $s \triangleq (\sigma_0, \cdots, \sigma_n, K)$. Let $t \in M_i$, $\sigma_i \triangleq (\mu_i, I_i)$, and $t \in enabled(s)$.

- $earliest\_firing\_time(s, t) \triangleq parent(t, \sigma_i) + Eft(t)$, and
- $latest\_firing\_time(s, t) \triangleq parent(t, \sigma_i) + Lft(t)$.

A state $s \triangleq (\sigma_0, \cdots, \sigma_n, K)$ is called *safe*, if for every output transition t_O such that $t_O \in globally\_firable(s)$, and for every I/O-module $M_j (0 \leq j \leq n)$ such that $\mathsf{wire}(t_O) \in \Sigma_j^{in}$, there exists an input transition t_I such that $\mathsf{wire}(t_I) = \mathsf{wire}(t_O), t_I \in enabled(s)$, $earlier(earliest\_firing\_time(s, t_I), t_O, s)$ holds, and either

1. $earlier(t_O, latest\_firing\_time(s, t_I), s)$, or
2. for some output transition t such that $t \in enabled(s)$, $earlier(t, latest\_firing\_time(s, t_I), s)$.

A state $s \triangleq (\sigma_0, \cdots, \sigma_n, K)$ is called *live*, if for every input transition t_I such that $t_I \in globally\_firable(s)$, there exists an output transition t (of an arbitrary I/O-module) such that $t \in globally\_firable(s)$.

Let I/O-modules $M_1, \cdots, M_n$ be represented by nets $N_1, \cdots, N_n$. A safety failure corresponds to a non-safe state in the reachable state space, and a timing failure occurs if a state can be reached which is not live. In other words, $\mathsf{failure}(M_1, M_2, \cdots, M_n)$ is empty, iff every state which is reachable from the initial state of $\langle N_1, \cdots, N_n \rangle$ is both safe and live. Therefore, the verification of conformance between I/O-modules can be done by traversing the state space of $\langle N_1, \cdots, N_n \rangle$ and checking if non-safe or non-live states are reachable.

Furthermore, it is possible to replace an abstract description of an I/O-module by a more concrete implementation. If $\{M_1, \cdots, M_{k-1}, M_k, M_{k+1}, \cdots, M_n\}$ conforms to M_S, $\{M_{k_1}, ..., M_{k_m}\}$ conforms to M_k, and $(\bigcup_{j=1}^{m} \Sigma_{k_j} - \Sigma_k) \cap \bigcup_{j=1}^{n} \Sigma_j = \emptyset$, then $\{M_1, \cdots, M_{k-1}, M_{k_1}, ..., M_{k_m}, M_{k+1}, \cdots, M_n\}$ conforms to M_S. The set of wires in a specification usually is much smaller than the set of wires in the implementation. Thus, the total computation cost to determine whether $\{M_1, \cdots, M_{k-1}, M_k, M_{k+1}, \cdots, M_n\}$ conforms to M_S and $\{M_{k_1}, \cdots, M_{k_m}\}$ conforms to M_k is significantly smaller than the computation of whether $\{M_1, \cdots, M_{k-1}, M_{k_1}, \cdots, M_{k_m}, M_{k+1}, \cdots, M_n\}$ conforms to M_S. This is the primary advantage of hierarchical verification.

9.3 Experimental Results

We have implemented the algorithm shown in the previous section on a UNIX workstation in C++. In this section, we present some experimental verification results.

First, our verifier shows that the implementation in Fig. 9(b) is correct with respect to the specification in Fig. 9.3(b) after traversing 51 states, which takes about one second on a 17 MIPS workstation.

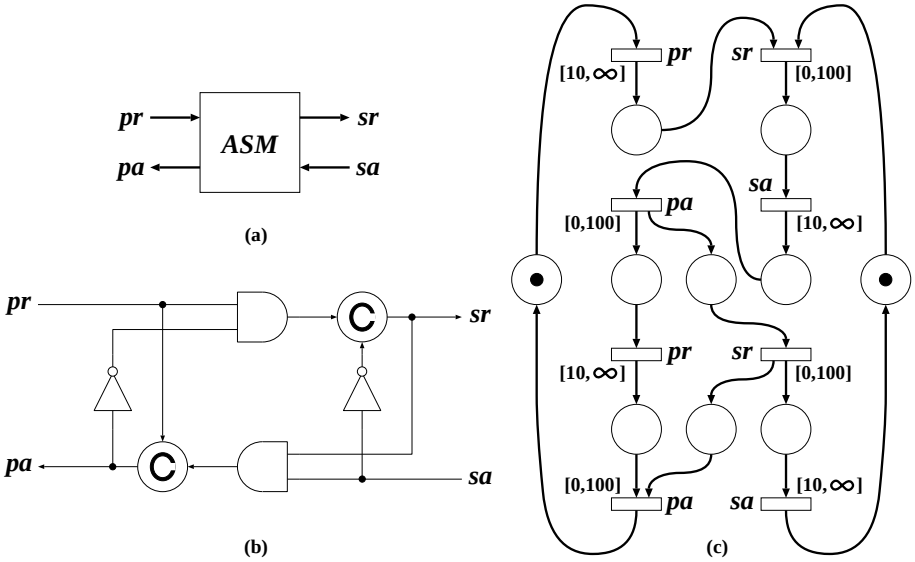


Figure 9.4: An automatic sweeping module, gate level implementation, and specification

The second example is a control circuit of the request-acknowledgement handshake mechanism for asynchronous circuits. This circuit called an automatic sweeping I/O-module (ASM, for short) has two inputs (a primary request pr, a secondary acknowledgement sa) and two outputs (a primary acknowledgement pa, a secondary request sr) (Fig. 9.4(a)). It has the following functionality:

1. When the primary request goes high with the secondary acknowledgement low, ASM sets the secondary request.

2. When the secondary acknowledgement becomes high, ASM resets the secondary request with setting the primary acknowledgement.

3. When the primary request becomes low, ASM resets the primary acknowledgement.

This functionality with almost the same assumptions as for the C-element is specified with a net as shown in Fig. 9.4(c). On the other hand, Fig. 9.4(b) was proposed as the gate level implementation of ASM. We assume that each gate has a delay [5,10].

Our verifier shows that this implementation is correct with respect to the specification in Fig. 9.4(c). In Table 9.1, the column *flat* shows the size of the nets, the number of states, and CPU times needed for this verification when C elements are expanded by using their gate level implementations shown in Fig. 9. The column *hierarchical* shows the results of the hierarchical verification. That is, the specification net shown in Fig. 9.3(b) is used for the verification of ASM. In this case, the total verification time is the sum of the verification times for both ASM and C-element. These results show the advantage of the hierarchical verification as well.

Table 9.1: Results of verification

	flat			*hierarchical*		
	size†	states	CPU(s)	size†	states	CPU(s)
C-element	–	–	–	p:30, t:34	51	1.3
ASM	p:78, t:90	391	81.8	p:34, t:34	58	1.2
Total	p:78, t:90	391	81.8	p:64, t:68	109	2.5

† : "p:" and "t:" number of places and transitions, respectively.

Chapter 10

Real-Time Conformance

Whereas the previous chapter described a particular algorithm for a very specific notion of timing correctness in the context of asynchronous circuit verification, in this chapter we compare different such notions.

Clearly, the notion of conformance and the mirror property depend on what we regard as a failure. For untimed systems, *safety failures* [Dil89b] are well-understood. In timed systems, additional failures may arise by wrong timing. However, it is much less clear what an intuitive and generally acceptable definition of timing failure could be.

One of the main problems in conformance checking for real-time systems is that the mirror property does not hold in general. Depending on the chosen definition of failure, it may or may not be possible to implement a conformance checking procedure by mirroring. We give several possible definitions of safety and timing failures, and in each case show why the mirror property fails, or give sufficient conditions under which it holds. Furthermore, we discuss possible ways to implement conformance checking algorithms without relying on the mirror property.

10.1 A Notion of Correctness

We briefly recall that a *failure* between I/O-modules is just a trace in the I/O-module's alphabet. Later on, we will define several types of failures between I/O-modules. Dependent on the respective definition of failure, *conformance* is defined as a notion of correctness of an implementation with respect to a specification.

Suppose that a system is modeled by a set $\mathcal{M}_C = \{M_1, \cdots, M_n\}$ of I/O-modules, and that an admissible specification for this system is given by the

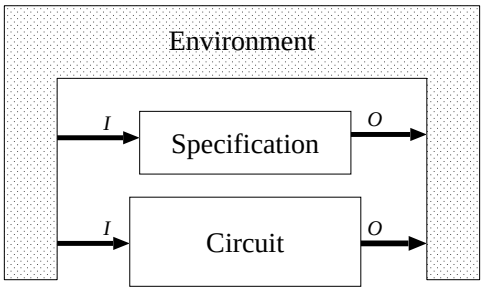


Figure 10.1: Correctness of a system with respect to a specification.

I/O-module $M_S = (\Sigma_S^{in}, \Sigma_S^{out}, T_S)$. Recall from the previous chapter that $\mathcal{M}_C$ *conforms to* M_S, if in any admissible environment M_E for M_S such that $\{M_S, M_E\}$ is failure-free, $\mathcal{M}_C \cup \{M_E\}$ is also failure-free. If $\mathcal{M}_C$ conforms to M_S, then $\mathcal{M}_C$ may fail in an environment only if the specifications allows a failure in this environment (See Fig.10.1).

To implement a conformance checking algorithm, the definition is not well suited: in general it is not possible to construct every admissible M_E which has no failure with M_S. Therefore, in the previous chapter we used the following

[**Mirror property**] $\mathcal{M}_C$ conforms to M_S, iff $\mathcal{M}_C \cup \{M_S^m\}$ is failure-free.

If the mirror property holds, then it is easy to implement a conformance checking procedure without considering all possible environments M_E: we just have to construct the set of failures of $\mathcal{M}_C \cup \{M_S^m\}$. Unfortunately, for timed system verification, the mirror property does not hold in general. In the following two sections, we will discuss necessary and sufficient conditions for the mirror property.

10.2 Checking Safety Properties

Recall that for a trace x and I/O-modules $M(= (\Sigma^{in}, \Sigma^{out}, T))$, M_1, and M_2, the notion of admittance of x by M was defined as follows: $M \models x$, if $\mathsf{project}(x, \Sigma^{in} \cup \Sigma^{out}) \in T$, and $M_1 \cap M_2 \models x$, if $M_1 \models x$ and $M_2 \models x$. For example, in Fig. 10.2 it holds that $M_S^m \cap M_I \models ((w,5)(v,11))$ and $M_I \models ((a,1)(u,9))$. Note that in the latter example, $(a,1)$ is projected out since $a \notin \Sigma_I^{in} \cup \Sigma_I^{out}$.

Also, recall from the previous chapter that for a given set $\mathcal{M} = \{M_1, \cdots, M_n\}$ of I/O-modules, where $M_k = (\Sigma_k^{in}, \Sigma_k^{out}, T_k)$, a *safety fail-*

ure of $\mathcal{M}$ is a finite trace $x = y(w,\tau)$, where $w \in \Sigma_k^{out}$ for some $k \leq n$, such that $M \models y$, $\mathcal{M} \mid\!\!\sim y \circ \tau$, and $M_k \models x$, but $\mathcal{M} \not\models x$. We denote the class of all safety failures of $\mathcal{M}$ by $\mathsf{failure0}(\mathcal{M})$. A set $\mathcal{M}$ of I/O-modules is *safety failure-free* if $\mathsf{failure0}(\mathcal{M}) = \emptyset$. If $\mathcal{M} = \{M_0, M_1, M_2\}$ and $\mathcal{M}' = \{M_1, M_2\}$, we write $\mathsf{failure0}(M_0, M_1, M_2)$ or $\mathsf{failure0}(M_0, \mathcal{M}')$ for $\mathsf{failure0}(\mathcal{M})$ etc.

Intuitively, a safety failure occurs if after trace x one I/O-module M_k sends an output w at time τ, but some other I/O-module cannot receive the corresponding input. We do not regard $y(w,\tau)$ as a safety failure of $\mathcal{M}$ if y is not a trace of $\mathcal{M}$, or if after trace y some I/O-module M_j of $\mathcal{M}$ must admit some symbol w' before τ (i.e., if $\mathcal{M}$ can not wait after y for τ time units): In this case, the event (w', τ') will change the state of M_j and $\mathcal{M}$ may be able to accept (w,τ) after $y(w',\tau')$.

In the example in Fig. 10.2 below, the one-element trace $((u,8))$ is a safety failure of $\{M_E, M_I\}$: note that u is an output of M_I ($u \in \Sigma_I^{out}$), initially $\{M_E, M_I\}$ can wait for 8 time units ($\{M_E, M_I\} \mid\!\!\sim \varepsilon \circ 8$), and M_I can send symbol u at time point 8 ($M_I \models ((u,8))$), but at that time u is not enabled in M_E ($M_E \not\models ((u,8))$). On the other hand, $((u,8))$ is not a safety failure of $\{M_S^m, M_I\}$, since M_S^m can not wait for 8 time units in the initial state ($\{M_S^m, M_I\} \not\mid\!\!\sim \varepsilon \circ 8$). Actually, in $\{M_S^m, M_I\}$ symbol w will be sent by M_S^m before M_I can send u. After receiving w, no u–labelled transition is enabled in M_I. Instead, M_I sends v, which is safely received by M_S^m. Thus, there are no safety failures in $\{M_S^m, M_I\}$.

From this example it follows that for the class of all safety failures, the mirror property does not hold. Since in $\{M_E, M_S\}$ no outputs are enabled and safety failure only occur when an output is sent, $\mathsf{failure0}(M_E, M_S) = \emptyset$. As shown above, $((u,8)) \in \mathsf{failure0}(M_E, M_I)$. From the definition of conformance, it follows that M_I does not conform to M_S. However, $\mathsf{failure0}(M_S^m, M_I) = \emptyset$. Hence, the mirror property fails for this example. (Note that in the previous section, we had additional constraints which render this example invalid.)

A sufficient condition that the mirror property holds with respect to safety failures is that specifications are unbounded. This corresponds to system specifications with unbounded delays. That is, we have the following theorem.

Theorem 10.1 *If the specification is unbounded, then the mirror property defined by* failure0 *holds.*

In order to prove this theorem, we first prove two lemmata, which will be needed also in the next section. Let $M_S = (\Sigma_S^{in}, \Sigma_S^{out}, T_S)$, $M_E =$

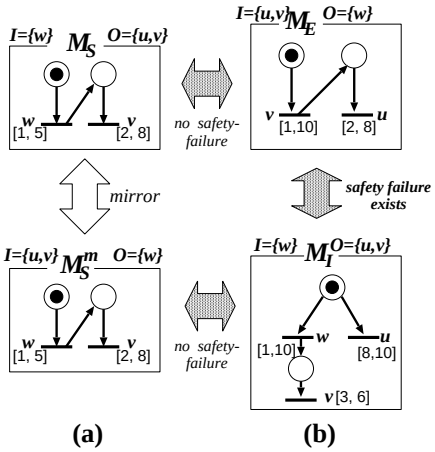


Figure 10.2: An example in which the mirror property defined by failure0 does not hold.

$(\Sigma_E^{in}, \Sigma_E^{out}, T_E)$, and $\mathcal{M}_C = \{M_1, M_2, \cdots, M_n\}$ with $\Sigma_C^{out} = \bigcup_{k=1}^{n} \Sigma_k^{out}$, $\Sigma_C^{in} = \bigcup_{k=1}^{n} \Sigma_k^{in} \setminus \Sigma_C^{out}$ such that $\Sigma_E^{in} \subseteq \Sigma_S^{out}$, $\Sigma_E^{out} \supseteq \Sigma_S^{in}$, $\Sigma_C^{out} \cap \Sigma_E^{out} = \emptyset$, $\Sigma_S^{out} \subseteq \Sigma_C^{out}$, $\Sigma_S^{in} \supseteq \Sigma_C^{in}$, and $\Sigma_S^{in} \cap \Sigma_C^{out} = \emptyset$ hold. For simplicity, we use $\mathcal{M}_C$ in several notations as if it were one I/O-module. We assume that those notations are extended appropriately for a set of I/O-modules.

Lemma 10.2 *If $\mathcal{M}_C$ conforms to M_S, then* $\mathsf{failure}(M_S^m, \mathcal{M}_C) = \emptyset$ *holds.*

PROOF: From the definition of the conformance, if $\mathcal{M}_C$ conforms to M_S, then $\mathsf{failure}(M_E, \mathcal{M}_C) = \emptyset$ holds for any M_E such that $\mathsf{failure}(M_E, M_S) = \emptyset$. Since $\mathsf{failure}(M_S^m, M_S) = \emptyset$, $\Sigma_S^{out} \subseteq \Sigma_S^{out}$, $\Sigma_S^{in} \subseteq \Sigma_S^{in}$, and $\Sigma_C^{out} \cap \Sigma_S^{in} = \emptyset$ hold, we can consider M_S^m as M_E, and have $\mathsf{failure}(M_S^m, \mathcal{M}_C) = \emptyset$. □

Lemma 10.3 *Suppose that* $\mathsf{failure}(M_E, M_S) = \emptyset$, $\mathsf{failure}(M_S^m, \mathcal{M}_C) = \emptyset$, *and $M_E \cap \mathcal{M}_C \models y$ hold. If M_S is unbounded, then $M_S \models y$ holds.*

PROOF: If $M_S \models y$ does not hold, then there must exist a longest prefix $z(u, \tau_u)$ of y such that $u \in \Sigma_S^{in} \cup \Sigma_S^{out}$, $M_S \models z$, but $M_S \not\models z(u, \tau_u)$. From $M_E \cap \mathcal{M}_C \models y$, we have $M_E \cap \mathcal{M}_C \models z(u, \tau_u)$. Since M_S is unbounded, $M_S \mid\!\!\!\vdash z \circ \tau_u$ follows from $M_S \models z$. If $u \in \Sigma_S^{out}$, then $u \in \Sigma_C^{out}$ holds from $\Sigma_S^{out} \subseteq \Sigma_C^{out}$. $M_S \mid\!\!\!\vdash z \circ \tau_u$ is equivalent to $M_S^m \mid\!\!\!\vdash z \circ \tau_u$. Thus, from $u \in \Sigma_C^{out}$, $M_S^m \mid\!\!\!\vdash z \circ \tau_u$, $\mathcal{M}_C \models z(u, \tau_u)$, but $z(u, \tau_u) \notin M_S^m$ (from $z(u, \tau_u) \notin M_S$), there must exist a safety failure in $\{M_S^m, \mathcal{M}_C\}$, but this contradicts $\mathsf{failure}(M_S^m, \mathcal{M}_C) = \emptyset$.

Similarly, if $u \in \Sigma_S^{in}$, a contradiction can be derived from the assumptions $u \in O_E$, $M_S \mid\!\!\vdash z \circ \tau_u$, $M_E \models z(u, \tau')$, $M_S \not\models z(u, \tau_u)$, and $\mathsf{failure}(M_E, M_S) = \emptyset$. Hence, we have shown that $M_S \models y$ holds. □

Now we are in a situation to prove Theorem 10.1.

PROOF: The direction "$\mathcal{M}_C$ conforms to $M_S \Rightarrow \mathsf{failure0}(M_S^m, \mathcal{M}_C) = \emptyset$" follows from Lemma 10.2. For the other direction "$\mathsf{failure0}(M_S^m, \mathcal{M}_C) = \emptyset \Rightarrow \mathcal{M}_C$ conforms to M_S", we assume that $\mathcal{M}_C$ does not conform to M_S, even if $\mathsf{failure0}(M_S^m, \mathcal{M}_C) = \emptyset$ holds. That is, there exists an I/O-module M_E such that $\mathsf{failure0}(M_E, M_S) = \emptyset$ but $\mathsf{failure0}(M_E, \mathcal{M}_C) \neq \emptyset$. Let $x = y(w, \tau)$ be a safety failure of M_E and $\mathcal{M}_C$. There are two cases: $w \in \Sigma_C^{in} \cap \Sigma_E^{out}$ and $w \in \Sigma_C^{out}$.

In the first case, from $w \in \Sigma_C^{in} \cap \Sigma_E^{out}$, $M_E \models x$ but $\mathcal{M}_C \not\models x$ hold as well as $M_E \mid\!\!\vdash y \circ \tau$ and $\mathcal{M}_C \mid\!\!\vdash y \circ \tau$. Since M_S is unbounded, $M_S \models y$ holds from Lemma 10.3, and also $M_S \mid\!\!\vdash y \circ \tau$ holds. From this together with $w \in \Sigma_E^{out}$, $M_E \models x$, and $\mathsf{failure0}(M_E, M_S) = \emptyset$, we have $M_S \models x$, i.e., $M_S^m \models x$. From $w \in \Sigma_C^{in} \subseteq \Sigma_S^{in}$, w is an output of M_S^m. Hence, from $\mathcal{M}_C \mid\!\!\vdash y \circ \tau$, $M_S^m \models x$, and $\mathsf{failure0}(M_S^m, \mathcal{M}_C) = \emptyset$, we have $\mathcal{M}_C \models x$, but this contradicts the hypothesis.

In the second case, if $M_E \not\models x$, then it implies $w \in \Sigma_E^{in}$, and the remaining proof is very similar to the above one (i.e., $M_E \models x$, which is a contradiction, is derived from $M_E \mid\!\!\vdash y \circ \tau$, $M_S \models x$, and $\mathsf{failure0}(M_E, M_S) = \emptyset$). Otherwise, $\mathcal{M}_C \not\models x$ holds, that is, for some $M_k, M_j \in \mathcal{M}_C$ such that $w \in \Sigma_k^{out} \cap \Sigma_j^{in}$, $M_k \models x$ but $M_j \not\models x$ hold. Also in this case, a contradiction ($\mathsf{failure0}(M_S^m, \mathcal{M}_C) \neq \emptyset$) is derived similarly from $\{M_S^m, \mathcal{M}_C\} \mid\!\!\vdash y \circ \tau$, $M_k \models x$, but $\{M_S^m, \mathcal{M}_C\} \not\models x$.

Since both cases lead to a contradiction, the assumption that $\mathcal{M}_C$ does not conform to M_S is incorrect. □

10.3 Timing Verification

If a system conforms to a specification with respect to safety failures, the system accepts any input that the specification accepts. However, the system may produce no output even if the specification requires it. That is, even if it is safety failure-free, a system might not satisfy desired timing properties ("it is always safe to do nothing"). Reconsider the example of Fig. 10.2: In the system $\{M_S, M_E\}$, initially M_E waits for input v from M_S, and M_S waits for input w from M_E. Clearly, this deadlock situation should be considered as a failure. Thus, we need a notion of failure which occurs when some I/O-module expects an input, and the corresponding output is not produced in

time. In this section, in addition to safety failures, we define six different versions of timing failures, which might be useful in practice. Here, we make no claim as to which of these definitions is most intuitive or adequate; our intention is to establish for each of these possible definitions whether the mirror property holds or not. We believe that the ultimate decision on what should be regarded as a timing failure depends on the specific application domain and modelling formalism.

First, for I/O-module M, trace y such that $M \models y$, and symbol w, define $latest(y, w, M)$ to be the maximal time τ such that $M \models y(w, \tau')$ for all $d \leq \tau' \leq \tau$, where d is the duration of y. Intuitively, $latest(y, w, M)$ represents the latest time when w can occur in M after y, if it is not in conflict with other transitions. If $M \not\models y(w, d)$, we define that $latest(y, w, M) = \infty$.

Given a set $\mathcal{M} = \{M_1, \cdots, M_n\}$ of I/O-modules, where $M_k = (\Sigma_k^{in}, \Sigma_k^{out}, T_k)$, a *type* $\boldsymbol{n}$ *timing failure* of $\mathcal{M}$ is a finite trace $x = y(w, \tau)$, where $w \in \Sigma_k^{in}$ for some $k \leq n$, such that $\mathcal{M} \models y$, $M_k \models x$, and Condition $\boldsymbol{n}$ is satisfied.

Condition 1: $\mathcal{M} \mid\!\!- y \circ \tau$, but $\mathcal{M} \not\models x$ holds.

Condition 2: $\mathcal{M} \mid\!\!- y \circ \tau'$ for every τ' such that $M_k \models y(w, \tau')$, but there does not exist τ'' such that $\mathcal{M} \models y(w, \tau'')$ holds.

Condition 3: There are no j, w', and τ' such that $w' \in \Sigma_j^{out}$ and $\mathcal{M} \models y(w', \tau')$

Condition 4: There are no j, w', and τ' such that $j \neq k$, $w' \in \Sigma_j^{out}$ and $\mathcal{M} \models y(w', \tau')$.

Condition 5: There are no j, w', and τ' such that $w' \in \Sigma_j^{out}$, $\mathcal{M} \models y(w', \tau')$, and $latest(y, w', M_j) \leq latest(y, w, M_k)$.

Condition 6: There are no j, w', and τ' such that $j \neq k$, $w' \in \Sigma_j^{out}$, $\mathcal{M} \models y(w', \tau')$, and $latest(y, w', M_j) \leq latest(y, w, M_k)$.

Similar as in the previous chapter, the intuition is that a timing failure occurs when an I/O-module expects an input, but it will not be given in time. Here, there are several interpretations on "in time". In type 1 timing failures, it is the exact time (τ) when the input is expected. That is, for each time point at which the receiver can accept a symbol the sender must be able to produce it. Thus, for example, $(u, 3)$ is a type 1 timing failures of M_1 and M_2 shown in Fig. 10.3(a), while $(u, 4)$ is not. Note that failure-freeness of $\{M_1, M_2\}$ with respect to both safety and type 1 timing failures is "almost" the same as trace equivalence of M_1 and M_2^m.

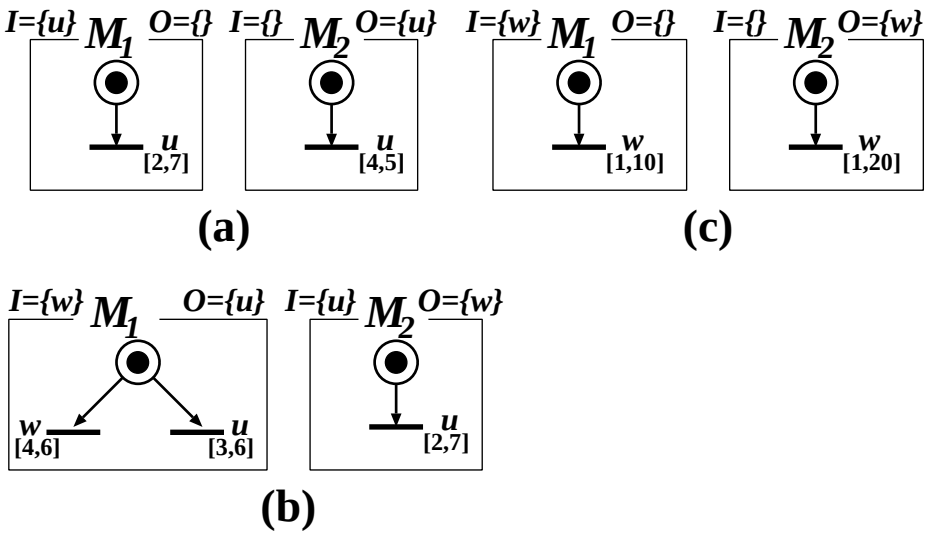


Figure 10.3: Examples of I/O-module sets.

In type 2 timing failures, it is the whole time period when the receiver can accept the input. That is, if the receiver expects an input within a certain time interval, the sender must be able to send it at least at one time point within this interval. If w can be produced at any one time point where M_k can accept it, then no type 2 timing failure occurs. Thus, the above M_1 and M_2 have no type 2 timing failures (M_1 expects input u any time between 2 and 7, and M_2 can send it, e.g., at time 5). As in the case of safety failures, if an I/O-module must change its state before the requested time, then there can not be a failure in this state. For example, $(u, 6)$ is not a type 1 timing failure of M_1 and M_2, because M_2 must make a transition at latest at time 5 and can not wait for 6 time units.

In Conditions 3 to 6, for avoiding a timing failure it is not necessary to send exactly the symbol w that the receiver expects . Instead, *some* output (w') must be produced and accepted in time. This reflects the consideration that in order to avoid a deadlock, it suffices to always be able to send at least one of the inputs the other partner is waiting for. However, if an alternative output (w') is sent and the former input (w) is still expected, the condition must hold again; therefore, timing failure freeness according to these conditions implies that if an input is continually expected it must finally be given. In this sense, the meaning of "in time" is similar to that of the type 2 timing failures. Since alternative outputs are allowed, we don't

have to require that it is possible to wait up to a certain time (i.e., that $\mathcal{M} \mid\!\!\vdash y \circ \tau'$).

In type 3 and 5, we allow that the alternative output is produced by *any* I/O-module (including the one which expects an input), whereas in type 4 and 6 the alternative symbol must be produced by some *other* I/O-module. The difference between Condition 2, 4 and 6, and Condition 3 and 5 is illustrated in Fig. 10.3(b), where the input w expected by M_1 can be disabled by sending a conflicting output u. In this example, $(w, 6)$ is a type 2 timing failure, since $\{M_1, M_2\} \mid\!\!\vdash \varepsilon \circ 6$ and $M_2 \not\models (w, \tau)$ for $\tau \leq 6$ (M_1 expects input w between time 4 and 6, but M_2 does not send it). Also, there is no alternative output sent by M_2, therefore it is also a timing failure according to Condition 4 and 6. On the other hand, $\{M_1, M_2\}$ is timing failure free according to Condition 3 or 5, since $M_1 \cap M_2 \models ((u, 6))$ and $latest(\varepsilon, u, M_1) = latest(\varepsilon, w, M_1) = 6$ (Intuitively, here we assume that if it doesnt get the expected input w in time, M_1 can choose to produce the output u instead). Clearly, if there are no conflicts between input and output within an I/O-module, then Condition 3 coincides with Condition 4, and Condition 5 coincides with Condition 6.

In Condition 5 and 6, we require an alternative output which *must* be sent before the latest time at which the input is expected. Intuitively, if a I/O-module expects an input within a certain time which is not provided, this is regarded as a failure even if an alternative output can occur at some later time. The difference between timing failures of type 3 and 4 and timing failures of type 5 and 6 is illustrated in Fig. 10.3(c). In this example, $((w, 5))$ is a type 5 and 6 timing failure, since $latest(\varepsilon, w, M_2) = 20 > 10 = latest(\varepsilon, w, M_1)$ (M_1 expects w before time 10, but M_2 may choose to send it only at time 20). However, $\{M_1, M_2\}$ has no timing failures of type 1 to 4, since for each time point $\tau \leq 10$ at which M_1 expects w, M_2 can produce w, and it is safety failure free, since $\{M_1, M_2\}$ can not wait for more than 10 time units.

There are still other possible definitions of timing failures which we considered; however, the above list seems to contain most of our intuitive ideas about the notion of timing failure in different application contexts. Let $\mathsf{failure}n(M_1, \cdots, M_n)$ be the union of all safety failures and type n timing failures of $\{M_1, \cdots, M_n\}$. Next, we will exhibit counterexamples to the mirror property for various failures.

Consider the example shown in Fig. 10.4. Neither $\{M_S, M_E\}$ nor $\{M_S^m, M_I\}$ has any safety failures, because w is accepted by the receiver, and M_S or M_S^m can not wait for 6 time units. $\{M_I, M_E\}$ has a safety failure $(u, 6)$, because $M_I \models (u, 6)$, $M_E \mid\!\!\vdash \varepsilon \circ 6$, but $(u, 6) \notin M_E$. Concerning tim-

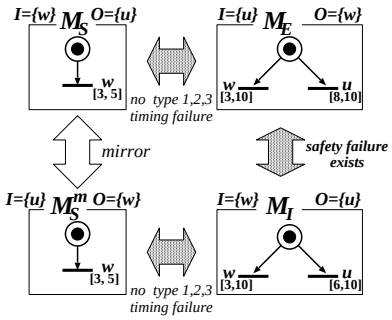


Figure 10.4: A counterexample to the mirror property for failure $1, 2, 3$.

ing failures, failure$n(M_S, M_E) = \emptyset$ holds for $n \in \{1, 2, 3\}$, because the input w of M_S is given by M_E in time, and for the input u of M_E, for example, $(u, 8)$, initially M_S can not wait for 8 time units. $(u, 8)$ is both a type 4 and 6 timing failure, because M_E itself produces the other output w. $(w, 4)$ is both a type 5 and 6 timing failure from $latest(\varepsilon, w, M_E) > latest(\varepsilon, w, M_S)$. For M_S^m and M_I, failure$n(M_S^m, M_I) = \emptyset$ holds for any n, because w of M_S^m is the only input and it is given by M_S^m in time. Although $\{M_I, M_E\}$ is timing failure-free, failure$n(M_I, M_E) \neq \emptyset$ for any n due to the safety failures of them (remember that failure$n$ includes safety failures). Hence, the mirror properties defined by failure$1, 2, 3$ do not hold in this example, because for $n \in \{1, 2, 3\}$, failure$n(M_S, M_E) = \emptyset$ and failure$n(M_I, M_E) \neq \emptyset$ (i.e., M_I does not conform to M_S), but failure$n(M_S^m, M_I) = \emptyset$.

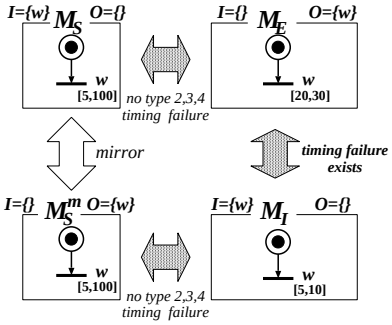


Figure 10.5: A counterexample to the mirror property for failure $2, 3, 4$.

In the example shown in Fig. 10.5, no safety failures exist in each pair of I/O-modules. As for $\{M_E, M_I\}$, note that M_I can not wait for 20 units or more. For $n \in \{2, 3, 4\}$, failure$n(M_S^m, M_I) =$ failure$n(M_S, M_E) = \emptyset$,

and $(w, 10)$ is in failure1(M_S, M_E), failure6(M_S^m, M_I), and failure5(M_S^m, M_I). The latter two are due to $latest(\varepsilon, w, M_S^m) > latest(\varepsilon, w, M_I)$. On the other hand, $(w, 10) \in$ failure$n(M_E, M_I)$ for $n \in \{1, 2, 3, 4, 5, 6\}$. Therefore, M_I does not conform to M_S with respect to failure$2, 3, 4$. Hence, the mirror properties defined by failure$2, 3, 4$ do not hold in this example.

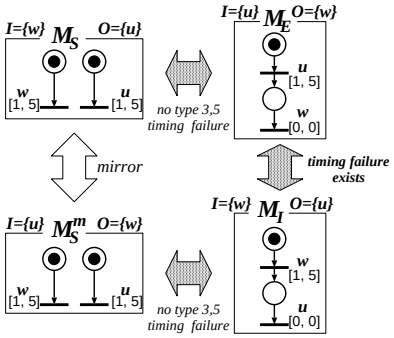


Figure 10.6: A counterexample to the mirror property for failure $3, 5$.

Fig. 10.6 shows an example in which the mirror properties defined by failure$3, 5$ do not hold. Similar to the previous example, here each pair of I/O-modules is safety failure free. Since no output symbols can change in M_E and M_I, $\{M_E, M_I\}$ has every type of timing failures (e.g. $(u, 1)$). On the other hand, failure$5(M_S, M_E) = \emptyset$ holds, because for the expected input w of M_S^m, some other output u is produced in time (i.e., $latest(\varepsilon, u, M_S^m) \leq latest(\varepsilon, w, M_S^m)$), and after receiving u, M_E immediately produce w, which is also in time again from $latest(\varepsilon, u, M_S^m) \leq latest(\varepsilon, w, M_S^m)$. This implies failure$3(M_S, M_E) = \emptyset$. Similarly, we can derive failure$n(M_S^m, M_I) = \emptyset$ for $n \in \{3, 5\}$. Hence, the mirror properties defined by failure$3, 5$ do not hold. failure$n(M_S, M_E) \neq \emptyset$ for $n \in \{1, 2\}$, because M_E can not produce w without receiving u, and failure$n(M_S, M_E) \neq \emptyset$ for $n \in \{4, 6\}$, because the same I/O-module M_S produces the other output u.

So far, we have given counterexamples to the mirror properties defined by failure1 to 5. Somewhat surprisingly, however, we have the following result.

Theorem 10.4 *The mirror property defined by* failure6 *holds.*

For the proof, note that in this case a similar lemma as for safety failures (Lemma 10.3) holds:

Lemma 10.5 *Suppose that* $\mathsf{failure6}(M_E, M_S) = \emptyset$, $\mathsf{failure6}(M_S^m, \mathcal{M}_C) = \emptyset$. *Then* $M_E \cap \mathcal{M}_C \models y$ *implies* $M_S \models y$.

PROOF: Literally as above, if $M_S \models y$ does not hold, then there must exist a longest prefix $z(u, \tau_u)$ of y such that $u \in \Sigma_S^{in} \cup \Sigma_S^{out}$, $M_S \models z$, but $M_S \not\models z(u, \tau_u)$. From $M_E \cap \mathcal{M}_C \models y$, we have $M_E \cap \mathcal{M}_C \models z(u, \tau_u)$. Again, first we show that $M_S \mid\!\!\vdash z \circ \tau_u$ holds. Assume that $M_S \mid\!\!\not\vdash z \circ \tau_u$. Then, there must exist $u_1 \in \Sigma_S^{in} \cup \Sigma_S^{out}$ such that $latest(z, u_1, M_S) < \tau_u$. If $u_1 \in \Sigma_S^{in}$, then from $\mathsf{failure6}(M_E, M_S) = \emptyset$, M_E has an output u_2 such that $latest(z, u_2, M_E) \leq latest(z, u_1, M_S)$, (Note that this holds only if n is equal to 6. For example, in the case of $n = 5$, it is not necessary that M_E has $u_2 \in \Sigma_E^{out}$ as above (i.e., M_S may have an output u_2' without violating $\mathsf{failure5}(M_S, M_E) = \emptyset$).) This contradicts $M_E \models z(u, \tau_u)$ because $latest(z, u_2, M_E) < \tau_u$. If $u_1 \in \Sigma_S^{out}$, then u_1 is an input of M_S^m. Thus, from $\mathsf{failure6}(M_S^m, \mathcal{M}_C) = \emptyset$, $\mathcal{M}_C$ has $u_3 \in \Sigma_C^{out}$ such that $latest(z, u_3, \mathcal{M}_C) \leq latest(z, u_1, M_S^m)$, (this also holds only in case $n = 6$) but this contradicts $\mathcal{M}_C \models z(u, \tau_u)$ because $latest(z, u_3, \mathcal{M}_C) < \tau_u$. Hence, we have $M_S \mid\!\!\vdash z \circ \tau_u$. The rest of the proof is exactly as in Lemma 10.3. □

From Lemma 10.5, Theorem 10.4 can be shown as follows.

PROOF: The direction "$\mathcal{M}_C$ conforms to $M_S \Rightarrow \mathsf{failure6}(M_S^m, \mathcal{M}_C) = \emptyset$" follows directly from Lemma 10.2. For the other direction "$\mathsf{failure6}(M_S^m, \mathcal{M}_C) = \emptyset \Rightarrow M_C$ conforms to M_S", we assume that M_C does not conform to M_S, even if $\mathsf{failure6}(M_S^m, \mathcal{M}_C) = \emptyset$ holds. That is, there exists an I/O-module M_E such that $\mathsf{failure6}(M_E, M_S) = \emptyset$ but $\mathsf{failure6}(M_E, \mathcal{M}_C) \neq \emptyset$. Let $x = y(w, \tau) \in \mathsf{failure6}(M_E, \mathcal{M}_C)$. There are two cases: $w \in \Sigma_C^{in} \cap \Sigma_E^{out}$ and $w \in \Sigma_C^{out}$. Here we give the proof for the first of these cases, the second being analogous.

1. In the case that x is a safety failure:
   From the definition of safety failures, we have $\mathcal{M}_C \mid\!\!\vdash y \circ \tau$, $M_E \models x$, but $x \notin \mathcal{M}_C$. From $\mathsf{failure0}(M_E, M_S) = \emptyset$, $w \in \Sigma_E^{out}$, and $M_E \models x$, either $M_S \models x$ or $M_S \mid\!\!\not\vdash y \circ \tau$ must hold. If $M_S \models x$, then from $w \in \Sigma_C^{in} \subseteq \Sigma_S^{in}$, $\mathcal{M}_C \mid\!\!\vdash y \circ \tau$, $\mathsf{failure0}(M_S^m, \mathcal{M}_C) = \emptyset$, we have $\mathcal{M}_C \models x$, but it contradicts the hypothesis. If $M_S \mid\!\!\not\vdash y \circ \tau$ holds, then from $M_S \models y$ (from Lemma 10.3) for some $w_s \in \Sigma_S^{in} \cup \Sigma_S^{out}$, $latest(y, w_s, M_S) < \tau$ must hold. If $w_s \in \Sigma_S^{in}$, then from $\mathsf{failure6}(M_S, M_E) = \emptyset$, for some $w_e \in \Sigma_E^{out}$, $latest(y, w_e, M_E) \leq latest(y, w_s, M_S)$ holds. (This holds only if $n = 6$.) This contradicts $M_E \models x$ from $latest(y, w_e, M_E) < \tau$. If $w_s \in \Sigma_S^{out}$, then from $\mathsf{failure6}(M_S^m, \mathcal{M}_C) = \emptyset$, for some $w_c \in \Sigma_C^{out}$, $latest(y, w_c, \mathcal{M}_C) \leq latest(y, w_s, M_S)$ holds. (Again, this holds only if $n = 6$.) This contradicts $\mathcal{M}_C \mid\!\!\vdash y \circ \tau$ from $latest(y, w_c, \mathcal{M}_C) < \tau$.

2. In the case that x is a type 6 timing failure:
   From $w \in \Sigma_C^{in}$, $\mathcal{M}_C \models x$ must hold. From $\mathsf{failure6}(M_S^m, \mathcal{M}_C) = \emptyset$, $\mathsf{failure6}(M_E, \mathcal{M}_C) \neq \emptyset$, and $M_S \models y$ (from Lemma 10.3), for some $w_s \in \Sigma_S^{in}$, $latest(y, w_s, M_S) \leq latest(y, w, \mathcal{M}_C)$ holds. (This holds even if $n \neq 6$.) From this and from $\mathsf{failure6}(M_E, M_S) = \emptyset$, for some $w_e \in \Sigma_E^{out}$, $latest(y, w_e, M_E) \leq latest(y, w_s, M_S)$ holds. (This holds only if $n = 6$.) From the above discussion, $\mathsf{failure0}(M_E, \mathcal{M}_C) = \emptyset$ holds. This and $latest(y, w_e, M_E) \leq latest(y, w, \mathcal{M}_C)$ derives $\mathcal{M}_C \models y(w_e, \tau_e)$. However, this means that we have $w_e \in \Sigma_E^{out}$, $latest(y, w_e, M_E) \leq latest(y, w, \mathcal{M}_C)$, and $M_E \cap \mathcal{M}_C \models y(w_e, \tau_e)$, which contradicts x is a type 6 timing failure.

Since both cases lead to contradictions, the assumption that $\mathcal{M}_C$ does not conform to M_S must be incorrect. □

The rest of this chapter deals with extensions and variations of these Theorems. We omit the proofs; they can be found in the extended version of [ZYS01]. First, we discuss additional conditions such that the mirror property holds for the various notions of failure. Since type 1 timing failures are very similar to safety failures, Theorem 10.1 indicates that unboundedness of the specification plays an important role. In fact, the following theorem holds.

Theorem 10.6 *If the specification is unbounded, then the mirror property defined by* failure1 *holds.*

Unboundedness of the specification is not sufficient for the mirror properties of the other failures. Actually, in the examples in Fig. 10.5 and Fig. 10.6, even if the nonzero latest firing times of u and w in each I/O-module are increased up to infinity, still $\mathsf{failure}n(M_S, M_E) = \mathsf{failure}n(M_S^m, M_I) = \emptyset$ and $\mathsf{failure}n(M_E, M_I) \neq \emptyset$ holds for $n \in \{2, 3, 4, 5\}$.

If the inputs and outputs in a specification cannot be fired simultaneously, i.e., for any y such that $M_S \models y$ there do not exist $u \in \Sigma_S^{in}$ and $w \in \Sigma_S^{out}$ such that both $M_S \models y(u, \tau)$ and $M_S \models y(w, \tau')$, then we say that the specification is *I/O conflict free*. I/O conflict freeness of a specification implies that some other I/O-module has to produce an output for the input expected by the specification. Thus, this is very similar to condition 6 in the definition of timing failures. However, for the mirror property to hold it is sufficient to require this condition only at the specification level:

Theorem 10.7 *If the specification is I/O conflict free, then the mirror property defined by* failure5 *holds.*

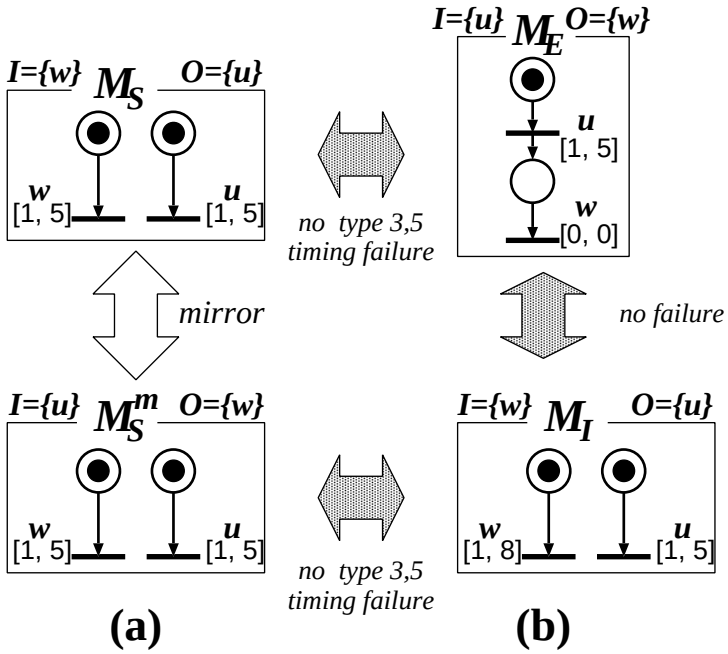


Figure 10.7: A correct implementation.

For the verification of timed asynchronous circuits, it can be argued that timing failures of type 5 are most appropriate. However, in this context we can not always guarantee that specifications are I/O conflict free. Thus, in [ZYS01] we started to develop techniques for conformance checking (w.r.t. failure5) without mirroring. We only briefly discuss the ideas; the interested reader is referred to [ZYS01].

For a set $\mathcal{M} = \{M_1, \cdots, M_n\}$ of I/O-modules and a trace x, we call $w \in \Sigma_j^{out}$ a *limited output*, if $latest(x, w, M_j) \leq latest(x, w', M_k)$ holds for any k and $w' \in \Sigma_k^{out}$. That is, w decides the latest time point up to which $\mathcal{M}$ can wait after x, if the type 5 timing failures do not exist (i.e., each input can wait longer). Furthermore, if such limited outputs exist only in one I/O-module M_i, then we call them *real* limited outputs of M_i. More than one limited output can be real limited as long as it exists in the same I/O-module. For example, in Fig. 10.6 w is a real limited output of M_S^m for $\{M_S^m, M_I\}$ and a trace ε. In Fig. 10.7 w is a limited output of M_S^m for $\{M_S^m, M_I\}$ and a trace ε, but not a real limited output because u of M_I is also a limited output and it is not in M_S^m.

Given a set $\mathcal{M} = \{M_1, \cdots, M_n\}$ of I/O-modules, where $M_k =$

$(\Sigma_k^{in}, \Sigma_k^{out}, T_k)$, and another I/O-module $M_0 = (\Sigma_0^{in}, \Sigma_0^{out}, T_0)$, a *pseudo timing failure* of $\{M_0\} \cup \mathcal{M}$ with respect to M_0 is a finite trace $x = y(w, \tau)$, where $w \in \Sigma_0^{in}$, such that $\mathcal{M} \models y$, $M_0 \models x$, and $latest(y, w, M_0) = latest(y, u, M_0)$ holds for some real limited output u of M_0.

In the example of Fig. 10.6, $(u, 5)$ is a pseudo timing failure of $\{M_S^m, M_I\}$ with respect to M_S^m, because w is a real limited output of M_S^m and has the same latest value as the input u (i.e., $latest(\varepsilon, u, M_S^m) = latest(\varepsilon, w, M_S^m)$ holds). On the other hand, $\{M_S^m, M_I\}$ in Fig. 10.7 has no pseudo timing failure with respect to M_S^m, because w is not a real limited. Note that a pseudo timing failure is not included in the type 5 timing failures. Let $\mathsf{failure}P(M_0; \mathcal{M}_C)$ be the union of $\mathsf{failure}5(M_0, \mathcal{M}_C)$ and the set of pseudo timing failures of $\{M_0\} \cup \mathcal{M}_C$ with respect to M_0. With these definitions, we can prove the following theorem.

Theorem 10.8 *Suppose that conformance is defined by* failure5. *Then* $\mathcal{M}_C$ *conforms to* M_S *iff* $\mathsf{failure}P(M_S^m; \mathcal{M}_C) = \emptyset$.

According to this theorem, conformance checking for failure5 can be done by constructing $\mathsf{failure}P(M_S^m; \mathcal{M}_C)$. In the case of time Petri nets, an algorithm similar as the one given in the previous chapter can be used to generate and compare the respective state spaces.

Part IV

Debugging and Testing

Chapter 11

Model Checking of Program Runs

In the previous chapters, we have argued that with partial order techniques, model checking can be used to verify finite-state systems of considerable size. Because of the state explosion problem, in many cases a complete state space analysis of large parallel and distributed programs is impossible. Generally, the number of reachable states is exponential in the number of state variables of the system. Especially, often the size of the input space of a parallel or distributed algorithm is too large for a complete traversal. The input space is defined by the number of input variables and their ranges. When a system has n input variables, each of which has domain D, then the size of the input space is $|D|^n$. For example, in a sorting algorithm for a field of 100 integer variables, where each integer is represented by 32 bits, the size of the input space is $(2^{32})^{100} \simeq 10^{1000}$. Even with symbolic methods, a complete traversal of the states of the unabstracted sorting algorithm is not possible.

In this chapter, we therefore focus on partial order *debugging* techniques for parallel programs. Debugging can be seen as an alternative way of partial state space traversal: In contrast to verification and testing, the debugging process is started whenever it has been recognized that the software contains a bug, and the task is to locate the error. Thus, the search can be restricted to those parts which are involved in the occurrence of the error.

We develop an on-the-fly algorithm for model checking of temporal logic safety properties on partially ordered occurrence net structures. This algorithm is then used for the automated debugging of parallel programs. During the monitoring of a program run, a state action net is constructed

from the program trace. Temporal specifications are evaluated on-the-fly with respect to this net. The specifications can express e.g. that an error has occurred, or that certain control locations have been reached. When the specified condition can occur, the execution is halted. Since we use a partial order logic, specification violations can be detected even if they did not actually occur in the particular interleaving of the program run. The results of this chapter are a joint work with M. Frey from Viag Interkom GmbH, Munich [FS98].

The "result" of a reactive program run often is modelled as a finite or infinite trace or sequence of events. For parallel programs, it is sometimes more appropriate to model the program behaviour as a *partial order*. Model checking can be used to determine whether a specific property formulated in an appropriate temporal logic holds for the given program run. This checking can either be done after termination or abortion of the program, or "on-the-fly" during the execution.

Several authors [BW83, Bat95, BFV86] have proposed to use formal specifications for the debugging of parallel programs. In contrast to testing, during the debugging process it is possible to observe and modify internal variables of the system. Model checking can be used to determine the possible causes of an error. Hypotheses about the causes of an error are specified in temporal logic. If the hypotheses are not satisfied, a counter example is generated to explain why the specification fails.

A first implementation of this idea was integrated in the debugging tool IDD [HHK85]. Requirements on the sequence of events of handling a common communication device of a distributed system were specified in a temporal interval logic. These requirements were automatically checked during runs of the system. In [GYK90] a linear time temporal logic for the specification of debugging assumptions is used. Events occurring during a run of the system are recorded together with a time stamp. Then the linear sequence of events, where the events are totally ordered by their time stamp, is used for model checking.

Both of the above approaches use a linear representation of parallel system runs. The ordering of events by time stamps introduces dependencies between events, which are not inherent in the run. For example, assume that the specification says that event A should happen before event B. If there are causally independent occurrences of A and B in a run, where by chance A's time stamp smaller than that of B, then a linear time approach will not detect the possible error of B happening before A.

Therefore, in [GW94] a partially ordered models to debug distributed program runs is introduced. The authors propose an efficient algorithm for

automatic model checking of a limited temporal logic during the program run. Thus, some errors can be detected even if they do not occur in the arbitrary interleaving of independent events during the program run. Frey and Weininger [FW94] extend this approach to the full partial order temporal logic of Reisig[Rei88]. The corresponding models are special kinds of finite causal nets, and the logic describes properties of global states of these models. Since models must be finite, model checking is applied "post mortem". To debug nonterminating reactive programs, the execution is aborted after some random time interval.

Here, we extend the results of [Fre96] to allow model checking of (potentially) infinite runs "on-the-fly". Thus, the debugging process can run in parallel to the system under development. Execution is halted when an error is found, and variable values can then be inspected and modified. We give conditions for formulas which can be checked during the program run, and develop an on-the-fly algorithm for the evaluation of these formulas on partial order models which are generated by the program run.

11.1 Debugging by Model Checking

Figure 11.1 shows an overview of our approach [Fre96] for checking whether an execution of a parallel program satisfies a temporal logic specification.

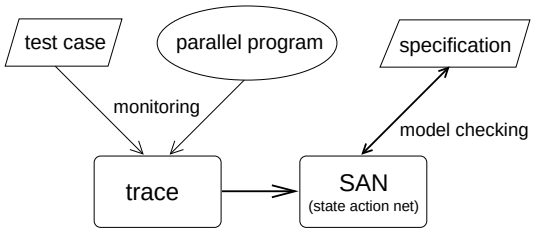


Figure 11.1: Specification-based debugging

We are given a parallel program, a test case of the program, and a temporal logic specification of properties for this test case. We execute the program and record a *trace* of this execution. Whenever during the test run synchronizations, communications, or accesses to variables are taking place, whenever executions of methods are starting or ending, and whenever new threads are generated or terminated, informations about these events is included in the trace. If the execution of the program is nonterminating, we abort it as soon as the trace reaches a certain length. The trace is then used to generate a special causal net called *state action net*, which is an abstract

model of the program run. We then apply model checking to determine whether the specification is satisfied by the state action net. If we find that the specification is not valid, we exhibit the events which lead to the fatal situation.

The construction of a partial order model from the linear trace gives an important advantage. Only those dependencies between threads are included in the model which represent synchronizations in the program run. Thus, errors can be detected even if they did not actually occur in the particular scheduling or interleaving of the program run.

11.1.1 State Action Nets

State action nets (SANs) are a special kind of finite causal nets, consisting of nodes and transitions. In SANs, transitions are called *actions*. Each action represents an atomic step in the program, i.e., the execution of a single program statement or a set of synchronized program statements. SAN nodes are called *local states*. Each local state is a description of one thread of control at a particular moment in the execution of the program. It contains the identification of the thread, the name of the method executed at the preceding action, and the values of variables in this thread.

Formally, a state action net is a tuple $N = (S, A, R)$, where

- S is a nonempty and finite set of local states,
- A is a nonempty and finite set of actions, and
- $R \subset (S \times A) \cup (A \times S)$ is the causal dependency relation.

State action nets can be seen a a special form of elementary Petri nets, cf. Chapter 1. The causal dependency relation satisfies the following conditions:

- the transitive closure of R is acyclic, and
- the preset and postset of each local state contains at most one action.

Figure 11.2 shows an example of a state action net. At each local state the executed methods are annotated. The variable **st** contains all executed methods in the order in which they have been called. The prefixes "S." and "T." indicate that the execution of a method is starting or terminating, respectively. Otherwise, the prefix is "I.".

State action nets can be generated from traces, which are recorded during a particular program run. Each traced event consists of an *event name*

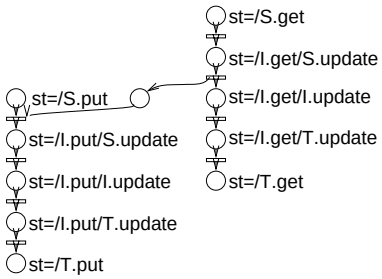


Figure 11.2: A state action net

and a set of *parameters*. In order to relate an event to a specific thread, each event contains the identification of the thread as a parameter. Further parameters describe dependencies between pairs of events of different threads. For example, the generation of a thread can be described by two events: the event `create_thread(x.i)` for the creator, and the event `new_thread(x.i)` for the created thread. Both events contain a parameter `x.i` which uniquely identifies this generation of threads. This unique identification is done by the thread identification `x` of the creator and a unique number `i` with respect to the creator.

Other events are traced whenever shared variables are generated, deleted, read or written. Synchronizations are also described by read and write access to shared variables, together with an event describing that the read access was done during examination of a waiting condition. Furthermore, there are events which describe the entering and leaving of method bodies, and events to describe the beginning of the execution of a new statement.

SANs are generated from traces by examining the traced events and reconstructing from them causal dependencies in the program run. Sequential dependencies between actions and local states within a single thread are generated according to the order in which they occur in the trace. Dependencies between actions of different threads are generated by synchronizing the traced events of both threads according to the causal dependency within the program run. For example, each event which describes the reading of a variable value is causally dependent on the event where this value was written. Thus, in the SAN a dependency is introduced between the corresponding actions.

Since the SAN of a program run is generated by a linear traversal of the program trace, it can be generated on-the-fly, during the program run.

11.1.2 A Temporal Logic for Debugging

Our temporal logic provides means to express local properties of single threads in an execution, as well as temporal dependency operators. For the intended application in debugging, we extended classical temporal logic in three ways:

- The logic consists of two tiers: a local and a global tier.
  - The local tier describes requirements on local states. We can express that variable values satisfy specific predicates and relations within a local state, for example, that $x = 3$ or $x \leq y$. Moreover, the predicate **in.***m* expresses that a methods m is executed in the action preceding the local state, and the starting state and the terminating state of the execution of m can be distinguished by the predicates **start.***m* and **term.***m*, respectively.

    Relational terms and predicates can be combined using the logical operators $\neg$ and $\wedge$.
  - The global tier has formulas of the local tier as atomic formulas. In addition to boolean operations it contains the temporal connectives $\mathbf{X}$, $\mathbf{F}^+$ and $\mathbf{U}^+$.
- To handle unknown variable values we use a three-valued interpretation. A formula containing a variable whose value is not visible in a certain state has the truth value *undefined*. The semantics of boolean operations is given by Lukasiewicz' classical three-valued interpretation of boolean algebra [RU71].
- To describe requirements of programs with dynamic generation of threads, we use *thread identifier variables* (TIVs) as placeholder for threads. Each atomic formula of the global tier is prefixed with a TIV. Formulas can be relativized by equality and other conditions between TIVs: (p **if** *cond*(*t*,*t'*)). The semantics of TIVs is defined by a *thread identifier variable assignment* (TIVA) which assigns a thread $\mathfrak{T}(t)$ in the program run to every TIV t.

Subsequently, we introduce on-the-fly model checking of program runs with our temporal logic. Therefore, we give the formal definition of the three-valued semantics of the global tier. A *slice* is a maximal set of local states which are not causally ordered. The step-relation between slices is defined by firing of transitions. Formally, the global model of an SAN N is defined by $G_N = (\mathfrak{S}, \mathfrak{R}, \mathfrak{s}_I, \mathfrak{s}_T)$, where $\mathfrak{S}$ is the set of all slices of N, and $\mathfrak{R} = \mathfrak{S} \times \mathfrak{S}$

describes the firing of actions. Here $(l, l') \in \mathfrak{R}$, if an action a exists, which is activated in l and l' is generated from l by firing a. The initial slice is $\mathfrak{s}_I$, and the terminal slice is $\mathfrak{s}_T$. The existence of initial and terminal slice is guaranteed since the slice-graph is a finite lattice.

For a given slice l and TIVA $\mathfrak{T}$, a formula $\mathtt{p}$ of the global tier can be either *true* ($(l, \mathfrak{T}) \models \mathtt{p}$), or *false* ($(l, \mathfrak{T}) \not\models \mathtt{p}$) or *undefined* ($(l, \mathfrak{T}) \models\!\!\!\uplus \mathtt{p}$). Formally, these three relations are defined as follows:

- $(l, \mathfrak{T}) \models t : \mathtt{p}_l$, if a local state $s \in l$ exists where $\mathfrak{T}(t) = tid(s)$, and $s \models \mathtt{p}_l$;
  $(l, \mathfrak{T}) \not\models t : \mathtt{p}_l$, if $s \in l$ exists where $\mathfrak{T}(t) = tid(s)$, and $s \not\models \mathtt{p}_l$;
  $(l, \mathfrak{T}) \models\!\!\!\uplus t : \mathtt{p}_l$, otherwise.

- $(l, \mathfrak{T}) \models \neg\mathtt{p}$ if $(l, \mathfrak{T}) \not\models \mathtt{p}$;
  $(l, \mathfrak{T}) \not\models \neg\mathtt{p}$ if $(l, \mathfrak{T}) \models \mathtt{p}$;
  $(l, \mathfrak{T}) \models\!\!\!\uplus \neg\mathtt{p}$, otherwise.

- $(l, \mathfrak{T}) \models (\mathtt{p} \wedge \mathtt{q})$ if $(l, \mathfrak{T}) \models \mathtt{p}$ and $(l, \mathfrak{T}) \models \mathtt{q}$;
  $(l, \mathfrak{T}) \not\models (\mathtt{p} \wedge \mathtt{q})$ if $(l, \mathfrak{T}) \not\models \mathtt{p}$ or $(l, \mathfrak{T}) \not\models \mathtt{q}$;
  $(l, \mathfrak{T}) \models\!\!\!\uplus (\mathtt{p} \wedge \mathtt{q})$, otherwise.

- $(l, \mathfrak{T}) \models \mathbf{X}\ \mathtt{p}$ if a slice l' exists where $(l, l') \in \mathfrak{R}$ and $(l', \mathfrak{T}) \models \mathtt{p}$;
  $(l, \mathfrak{T}) \not\models \mathbf{X}\ \mathtt{p}$, otherwise, i.e., if
  for all slices l' with $(l, l') \in \mathfrak{R}$ either $(l', \mathfrak{T}) \not\models \mathtt{p}$ or $(l', \mathfrak{T}) \models\!\!\!\uplus \mathtt{p}$.

- $(l, \mathfrak{T}) \models \mathbf{F}^+\ \mathtt{p}$ if a slice l' exists such that $(l, l') \in \mathfrak{R}^+$ and $(l', \mathfrak{T}) \models \mathtt{p}$;
  $(l, \mathfrak{T}) \not\models \mathbf{F}^+\ \mathtt{p}$, otherwise.

- $(l, \mathfrak{T}) \models (\mathtt{p}\ \mathbf{U}^+\ \mathtt{q})$ if a slice l' exists where
  $[(l, l') \in \mathfrak{R}^+$, and $(l', \mathfrak{T}) \models \mathtt{q}]$, and for all slices l'' where
  $[(l, l'') \in \mathfrak{R}^+$ and $(l'', l') \in \mathfrak{R}^+]$, either $(l'', \mathfrak{T}) \models \mathtt{p}$, or $(l'', \mathfrak{T}) \models\!\!\!\uplus \mathtt{p}$;
  $(l, \mathfrak{T}) \not\models (\mathtt{p}\ \mathbf{U}^+\ \mathtt{q})$, otherwise.

- $(l, \mathfrak{T}) \models$ **(**$\mathtt{p}$ **if** $cond(\boldsymbol{t}, \boldsymbol{t'})$**)** if $(l, \mathfrak{T}) \models \mathtt{p}$ and $cond(\mathfrak{T}(t), \mathfrak{T}(t'))$;
  $(l, \mathfrak{T}) \not\models$ **(**$\mathtt{p}$ **if** $cond(\boldsymbol{t}, \boldsymbol{t'})$**)** if $(l, \mathfrak{T}) \not\models \mathtt{p}$ and $cond(\mathfrak{T}(t), \mathfrak{T}(t'))$;
  $(l, \mathfrak{T}) \models\!\!\!\uplus$ **(**$\mathtt{p}$ **if** $cond(\boldsymbol{t}, \boldsymbol{t'})$**)**, otherwise.

According to this definition, all formulas $\mathbf{X}\ \mathtt{p}$, $\mathbf{F}^+\ \mathtt{p}$ or $(\mathtt{p}\ \mathbf{U}^+\ \mathtt{q})$ are either true or false in any slice l and TIV $\mathfrak{T}$. A formula $\mathtt{p}$ is defined to be *valid* in an SAN N, if $\mathbf{G}^+\ \mathtt{p}$ is true in the initial slice of N, where $\mathbf{G}^+\ \mathtt{p}$ is $\neg\ \mathbf{F}^+\ \neg\mathtt{p}$ as usual (cf. Chapter 1). Since the truth value of $\mathbf{G}^+\ \mathtt{p}$ in any slice is not *undefined*, the notion of validity in state action nets is two valued: A formula $\mathtt{p}$ is valid in an SAN N if for all l and $\mathfrak{T}$ of N either $(l, \mathfrak{T}) \models \mathtt{p}$ or $(l, \mathfrak{T}) \models\!\!\!\uplus \mathtt{p}$.

For example, consider the formula

$$((t1\text{:}(\textbf{in.}\textit{get} \wedge \textbf{term.}\textit{update})\ \textbf{before}\ t2\text{:}(\textbf{in.}\textit{put} \wedge \textbf{start.}\textit{update}))\ \textbf{if}\ t1 \neq t2)$$

Here ($\mathtt{p}$ **before** $\mathtt{q}$) $\triangleq \neg(\neg\mathtt{p}\ \mathbf{U}^+\ \mathtt{q})$ (for similar abbreviations, cf. Chapter 2). The formula specifies a precedence order on executions of the method *update* which has to be true if *update* is executed in parallel by several threads. It turns out that this formula is not valid in the SAN of Figure 11.2. In the next section, we describe an algorithm for automatically checking such properties.

11.1.3 Model Checking Debugging Logic

Our model checking procedure is a global and bottom-up evaluation of formulas. The checking is done by calculating all slices and TIVAs in which the formula is false. The formula is valid if the result is empty; otherwise the calculated slice and TIVA indicates where the formula is not satisfied. Since it is difficult to calculate slices and TIVAs in which subformulas of the given formula are undefined, we avoid the negation during model checking by further transformation of the input formulas. We use the dual temporal operators $\vee$, ~~$\mathbf{X}$~~, and $\mathbf{G}^+$, and replace negated subformulas by the respective dual formulas. For example, $\neg\ \mathbf{F}^+\ \mathtt{p}$ is replaced by $\mathbf{G}^+\ \neg\mathtt{p}$.

The slices are calculated bottom-up using the syntactical structure of the negated and transformed formula. This syntactical structure can be described by the syntax tree. To avoid the calculation of slices in which subformulas are undefined, we slightly change the syntax tree into a *formula tree*. The difference between both trees is that a child node of a node labelled by $\mathbf{G}^+\ \mathtt{p}$, **all_next** $\mathtt{p} \vee$ ($\mathtt{p}$ **before** $\mathtt{q}$) is labelled by the transformed formula logically equivalent to $\neg\mathtt{p}$. Leaf nodes of the formula tree may contain arbitrary formulas of the local tier. Figure 11.3 shows the formula tree of our example formula.

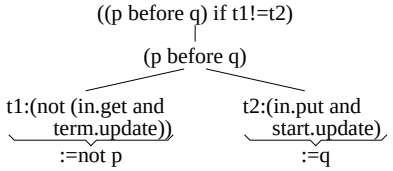


Figure 11.3: A formula tree

In [Fre96], model checking is done "post mortem", after generating the

whole SAN of the program run. Therefore, the formula tree can be evaluated bottom-up. All slices and TIVAs of the child nodes of a node in the formula tree have to be calculated before the slices and TIVAs of a node can be calculated. To do so, we need the global structure of the net. The generation of the SAN can be done on-the-fly during the program run. For checking infinite runs on-the-fly, only part of the global structure is available. Therefore it is necessary to restrict the logic appropriately.

11.2 Safety Requirement

We want to extend the post mortem model checking approach to reactive programs with potentially infinite executions. We propose to execute the program an unlimited amount of time and to check the validity of the formula for this run on-the-fly. This means that model checking can be done simultaneously during the program run. If a slice and a TIVA is generated for the root node of the formula tree, the run is stopped and user interaction can take place. Unfortunately, not all properties can be checked in this manner.

As described above, SANs can be generated on-the-fly. For our purposes, this creation has to satisfy the following two properties: Firstly, the information in local states must be completed before successors of the local state are generated. Otherwise, the valuation of formulas of the local tier may change depending on the changing values of the state. Thus, we could not guarantee that properties which are false will be remain false during all continuations of the run. Secondly, any continuation of an execution must lead to an extension of the SAN, in which all parts of the previously generated net are unchanged. Otherwise, new slices could be introduced in the old part of the net. This could lead to a change in the sequences of slices of the old part which could affect the valuation of formulas.

Both conditions on the generation of nets can be fulfilled in the generation of state action nets, if all traced variable values belong to different threads of the program. This condition is trivially satisfied in message passing programs. In shared memory programs, we can trace "local copies" of variables shared between several threads.

In addition to the conditions on SANs, we now exhibit a condition for the properties which are to be model checked on-the-fly. Only those properties can be examined which, after they are found to be false in a slice and a TIVA during part of the run, can not become true by the same slice and TIVA in a continuation of this run. Recall from Chapter 3, that a formula

φ is called a *safety property*, if for every SAN N,

$$N \models \varphi \text{ if } (\forall M < N)(\exists N' > M)\ N' \models \varphi$$

In this definition, M is an initial subnet of N, and N' is any continuation of this subnet M. In other words, φ is a safety property if for every net *not* satisfying φ there is a slice and TIVA such that $(l, \mathfrak{T}) \not\models \varphi$ and there is no extension N' such that $(l, \mathfrak{T}) \models \varphi$ in N'. Stated differently, for every net dissatisfying φ something "bad" must have happened after some finite amount of time which cannot be remedied by any future good behaviour.

For linear and branching time temporal logics, in Chapter 3 we discussed various syntactical characterizations of safety and liveness properties. However, these characterizations are not appropriate for our logic, since it is interpreted on partial order structures.

All formulas describing properties of finite parts of an infinite computation are safety-properties. If the length of the finite part is bounded by the property, then only those slices of a net extension have to be checked for which a succeeding sequence of slices exists which is bounded by the property. In particular, each formula of the local tier is a safety property.

If φ is a safety property, then the formula $\mathbf{G}^+ \varphi$, which is equivalent to $\neg\, \mathbf{F}^+ \neg\varphi$, is a safety property. In general, $\mathbf{F}^+ \varphi$ is not a safety property: if in a net no slice exists in which φ is satisfied, then $\mathbf{F}^+ \varphi$ is false in the initial slice. However, if an extension is constructed in which φ is true, then $\mathbf{F}^+ \varphi$ is true in the initial slice. Similarly, the formula $(\varphi$ **before** $\psi)$ is a safety property, but $(\varphi\ \mathbf{U}^+\ \psi)$ is not a safety property.

As an example for a formula φ such that neither φ nor $\neg\varphi$ is a safety property, consider $\varphi = \mathbf{F}^+\ (\mathtt{p} \wedge \mathbf{G}^+\ \mathtt{p})$. Since $\mathbf{F}^+\ \mathtt{p}$ is not a safety property, φ is not a safety property. The formula $\neg\varphi = \mathbf{G}^+\ (\neg\mathtt{p} \vee \mathbf{F}^+\ \neg\mathtt{p})$ is also not a safety property: $\mathbf{G}^+\mathbf{F}^+\ \mathtt{p}$ may be true in the initial slice of a net extension, because the terminal slice of the extension is the only one satisfying $\neg\mathtt{p}$. In all initial subnets, $\mathbf{G}^+\ (\neg\mathtt{p} \vee \mathbf{F}^+\ \neg\mathtt{p})$ is false, because in all of its slices $\mathtt{p}$ is satisfied.

To describe the formulas specifying safety properties, we characterize the set of formula trees of the negated and transformed formulas. A negated and transformed formula in the formula tree is called *slice-stable*, if for every SAN N, for every slice l and for every TIVA $\mathfrak{T}$ of N

$$(l, \mathfrak{T}) \models_N \varphi \text{ if } \forall M > N (l, \mathfrak{T}) \models_M \varphi$$

The relation $\models_N$ is defined with respect to the SAN N; $\models_M$ is defined with respect to an extension M of N. Formulas $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $\mathbf{X}\ \varphi$, $\mathbf{X}\ \varphi'$, $\mathbf{F}^+\ \varphi$

and $(\varphi' \ \mathbf{U}^+ \ \psi)$ are slice-stable if φ and ψ are slice-stable and φ' specifies a property of a finite part of the SAN. If φ is slice-stable, then its negation specifies a safety property. Thus, slice-stability is a sufficient criterion for the algorithm described in the next section to be applicable.

11.3 On-the-Fly Model Checking

In this section, we give a parallel algorithm for model checking safety properties on-the-fly. The evaluation of the specification can run in parallel with the program to be debugged, and independent subformulas can be evaluated in parallel.

In principle, the bottom-up model checking procedure described above could be applied. Whenever a new subnet is generated during the construction of the SAN from a trace, we could evaluate the formula tree from scratch on this subnet. However, with this procedure, for each net extension and each node of the formula tree all slices and TIVAs calculated in the previous net extension would have to be recalculated. This is not necessary: we can reuse the information from the previous subnet. For each extension and formula-tree node we have to deal only with those slices, which arise from the addition of further local states to the net.

For each node and each new slice we have to determine whether the respective subformula is false in the slice. In contrast to the post-mortem evaluation of the SAN, slices and TIVAs of all children of a node can be calculated at the same time. Conceptually, all nodes of the formula tree can be examined independently and, on a multiprocessor system, in parallel. This can be done by starting a thread for each node which calculates the slices of this node. To avoid recalculation, for each node we use a queue of slices and TIVAs, in which they are inserted in a consecutive order.

If all nodes are processed independently, it is necessary to synchronize the evaluation. Version numbers are assigned to each newly generated net extension. For each node, evaluation of the slices of a new net extension can begin when all child nodes have finished the calculation of these slices.

The method in Figure 11.3 describes the synchronization protocol for each node.

For each node n a new thread is generated executing the method `calc_slices`. The object *san* contains the state action net, with a boolean variable *terminated* which indicates when the program run is interrupted or terminated. In this case the variable *last_version* contains the final version number of the program run. The variable *version* contains the version

```
thread calc_slices(n)
    version := 0;
    while (not san.terminated and version<san.last_version)
        await(version < san.version);
        await(version < n.childs.version);
        calc_operation(n,version+1);
        version++;
```

Figure 11.4: Synchronization part of the model checking algorithm

number of the last generated net extension.

The first statement of the loop waits for the generation of a new net extension. The second statements enforces waiting until all child nodes have finished calculating the slices of the new net extension. Then, the method specified by the logical operator in the node is called. Additional slices and TIVAs of the net extension defined by the value of *version*+1 are calculated. If a slice and TIVA falsify the subformula of a node, then they are inserted in the corresponding queue. Finally, *version* is incremented to signal the parent node that the slices of the net extension are calculated for the node n.

In the following we describe the methods which are called by `calc_slices` to calculate the new value of the queue of a node. We give only the cases for atomic formulas, conjunction, and until-formulas; the other cases are similar. They can be found in [FS98].

Nodes labelled with atomic formulas

Leaf nodes of the formula tree are labelled with atomic formulas. When the method `calc_slices` is called for a leaf node, there are no waiting conditions for the child nodes. After a new net extension is generated, the method `calc_leaf` is called with node and version number of the net extension as parameters.

New slices and TIVAs can arise from two different sources. Firstly, new slices have to be built if the extension contains a state which is concurrent (not causally ordered) to a local state of a previous extension. Secondly, if the extension contains a state in which the local formula of the node is satisfied, all slices with this state have to be added. Therefore, the leaf

```
procedure calc_leaf(n, version)
    for all s ∈ san.new_states[version] do
        for all s' ∈ satisfying_states do
            if (s || s') then
                sl := states2slices({s, s'}, san.all_states[version]);
                for all l ∈ sl do
                    n.queue[version] := n.queue[version]∪
                        {(l, make_tiva(s'.tid, n.var))};
            if check_ll(s) then
                new_sat_states := new_sat_states ∪ {s}
                sl := states2slices({s}, san.all_states[version]);
                for all l ∈ sl do
                    n.queue[version] := n.queue[version]∪
                        {(l, make_tiva(s.tid, n.var))};
            satisfying_states := new_sat_states;
```

Figure 11.5: Calculation of leafs

node contains a variable *satisfying_states* which contains all states of previous extensions in which the formula of the node is satisfied. The variable *san.new_states* is an array of sets of states containing the new states of all extensions. The variable *san.all_states* is also such an array of sets of states containing all local states of a version. The method *states2slices* calculates all slices which must contain the states of the first parameter, and all other states of the slices are states of the second parameter. The method `make_tiva` generates the TIVA which assigns to the first parameter a TIV of the second parameter. $n.var$ is the set of TIVs of the node n, and $s'.tid$ is the thread identification of the local state s'. Method `check_ll`(s) checks whether in the local state of the parameter the formula of the node is satisfied.

Nodes labelled with $(\mathtt{p} \wedge \mathtt{q})$

Slices of a node labelled with $(\mathtt{p} \wedge \mathtt{q})$ are calculated by an intersection of slice sets of both nodes. This intersection has to take care of the TIVA associated with a slice. The parameters of `simple_and` are the set of variables and the set of slices and TIVAs of both child nodes. To calculate the intersection,

it has to be checked whether the TIVA of the slice of one child node and the TIVA of the slice of the other child node can be merged together. This is similar to the calculation of a most general unifier in automated theorem proving. We have to check that no TIV exists to which both TIVAs assign a different thread identification. This check is done by the method `comp`. The merging of TIVAs is done by the method *merge*.

For an on-the-fly generation of the slices of nodes labelled with $(\mathtt{p} \wedge \mathtt{q})$ it is not sufficient to simply calculate the intersection of the new slices of one extension. One of the child nodes could be the node of a temporal operator. A new slice can be generated for a new extension even if the slice itself did occur in prior extensions. As an example, consider the formula $\mathbf{F}^{+}$ `p`: If a slice of the new extension satisfies `p`, all slices which are predecessors of it satisfy $\mathbf{F}^{+}$ `p`. Some of these slices can also be slices of prior extensions. This leads to the method for an on-the-fly calculation shown in Fig. fig:alg-and. The method consists of two symmetrical parts, where each part calculates the intersection between the new slices of one node and all slices of the other node. It can be easily generalized to deal with arbitrary finite conjunctions.

```
procedure simple_and(var_p, var_q, sl_p, sl_q, sl_res)
    for all (l, T) ∈ sl_p do
        for all (l', T') ∈ sl_q ) do
            if l = l' and comp(T, T', var_p ∪ var_q) then
                T" = merge(T, var_p, T', var_q);
                sl_res := sl_res ∪ (l, T");

procedure calc_and(n, version)
    n.child_p.slice_set := n.child_p.slice_set∪
        n.child_p.queue[version];
    n.child_q.slice_set := n.child_q.slice_set∪
        n.child_q.queue[version];
    simple_and(n.child_p.var, n.child_q.var,
        n.child_p.queue[version], n.child_q.slice_set, sl);
    simple_and(n.child_p.var, n.child_q.var,
        n.child_p.slice_set, n.child_q.queue[version], sl1);
    n.queue[version] := sl ∪ sl1;
```

Figure 11.6: Calculation of intersection of slices

Nodes labelled with ($\mathtt{p}\ \mathbf{U}^+\ \mathtt{q}$)

There is only one source of new slices for a node labelled ($\mathtt{p}\ \mathbf{U}^+\ \mathtt{q}$): The slices l and TIVAs $\mathfrak{T}$ of a new extension of the child node labelled with $\mathtt{q}$ can lead to new slices which are a subset of the predecessors of l w.r.t. $\mathfrak{R}^+$.

The subset is given by all slices l' and TIVAs $\mathfrak{T}'$ where $\mathfrak{T}'$ is a TIVA for all TIVs of the node and it is an extension of $\mathfrak{T}$. Furthermore, no slice l'' exists between l and l' such that $(l'', \mathfrak{T}')$ is an element of the child node labelled with $\neg\ \mathtt{p}$. To check this condition, it is not necessary to know the slices and TIVAs of the child node labelled with $\neg\mathtt{p}$ of further net extensions, because only those slices of the child node which are predecessors of l are necessary. New predecessors of l cannot be generated in further net extensions because of the conditions on state action nets stated in section 3. The method in Figure 11.3 calculates the slices and TIVAs of a node labelled with ($\mathtt{p}\ \mathbf{U}^+\ \mathtt{q}$) on-the-fly.

To calculate the slices for the dual operator $\mathbf{W}^+$ is somewhat more complicated; for more information, the reader is referred to [FS98]. On the other hand, nodes labelled with formulas ($\mathtt{p}$ **if** $cond(t, t')$) pose no extra difficulties; such formulas constrain only the values of TIVAs. To calculate the slices and TIVAs satisfying a condition only the new slices and TIVAs of a new net extension are necessary for on-the-fly model checking.

Experimental applications

On-the-fly model checking allows to check potentially infinite executions of concurrent and distributed programs. Monitoring of traces, generation of state action nets and model checking can be done simultaneously in a pipelined execution order. If a slice and a TIVA is generated in the root node of the formula tree, the specification is not valid for this run. In this case, the program run is stopped, and the user has several choices:

- variable values, control locations and generated slices can be inspected to find the cause of an error,
- values and locations can interactively be changed by assigning new values to them, and
- the program run can be resumed with the same specification and history, to reach the same condition again, or
- the specification can be changed, the history purged and the execution restarted from the current state.

```
procedure calc_until(n, version, length)
    sl_all := gen_slices(l_I, san.final_slice[version], length);
    tiva_all := generate_tiva(empty, san.all_tiv[version], n.var);
    n.child_p.slice_set := n.child_p.slice_set ∪ n.child_p.queue[version];
    for all (l, 𝔗) ∈ n.child_q.queue[version] do
        for all (l',𝔗') ∈ sl_all × tiva_all do
            if comp(𝔗, 𝔗', n.child_q.var) and (l', l) ∈ ℜ+
            then
                C := true;
                for all (l", 𝔗") ∈ n.child_p.slice_set do
                    if comp(𝔗', 𝔗", n.child_p.var) and
                        (l', l") ∈ ℜ+ and (l'', l) ∈ ℜ+
                    then C := false;
                if C and l ∈ sl_all then
                    n.queue[version] := n.queue[version] ∪ (l', 𝔗');
```

Figure 11.7: Checking the partial order until-operator

If no slice and TIVA falsifying the formula is found, the program could run an unlimited amount of time. However, for each program event all slices containing this event have to be stored. Thus, an unlimited amount of memory would be needed for this purpose. Therefore, the program can only be run for a limited number of steps. If no error occurs within this time, the program has to be stopped and the history must be purged.

A typical application is as follows. Assume that the user wants to debug a parallel sorting algorithm. There is one thread T_s executing a method *split* which splits the input into even and odd numbers, and two threads T_e and T_o executing a method *sort* to sort the even and odd numbers, respectively. To distinguish both threads *sort* has a boolean parameter *is_e* which has the value true for T_e and the value false for T_o during the execution of *sort*. When T_e and T_o are finished, the resulting sequences are merged by T_m. To control that the splitting is done correctly, the user would specify

$$(t1{:}((\textbf{start.}sort \wedge is\_e) \rightarrow \text{even}(input)) \wedge$$
$$\ t2{:}((\textbf{start.}sort \wedge \neg\ is\_e) \rightarrow \text{odd}(input))).$$

Thus, whenever e.g. the thread for even numbers gets an odd input the execution is stopped. The cause of this error can then be found by examination

of the sequence of actions leading to this situation. The following formula assures that T_e must be terminated before T_m is started:

$$(t{:}(\textbf{start.}\mathit{split}) \rightarrow\ (t'{:}(\textbf{term}.\mathit{sort} \wedge \mathit{is\_e}) \\ \textbf{before}\ t''{:}(\textbf{start}.\mathit{merge}))).$$

If the synchronization between T_e and T_m is faulty, the program is stopped on the preliminary start of the merging process. To debug the merging, the user would halt the program when both separate sorting processes are finished:

$$\neg(t1{:}(\textbf{term}.\mathit{sort} \wedge \mathit{is\_e}) \\ \wedge\ (t2{:}\textbf{term}.\mathit{sort} \wedge\neg\ \mathit{is\_e})).$$

The thread for merging can be advanced a single step with the specification

$$(t{:}(\textbf{start.}\mathit{merge}) \rightarrow\ \textbf{all\_next}\ t{:}(\textbf{term.}\mathit{merge})).$$

The whole program can be advanced n steps with the following formula:

$$(t{:}(\textbf{start.}\mathit{split}) \rightarrow\ \textbf{all\_next}^n\ t'{:}(\textbf{term.}\mathit{merge})).$$

Since all of these specification formulas follow a fixed scheme, we have investigated the possibility of standard templates for them.

We experimented with a preliminary implementation of our algorithm. One of the main limiting factors in the implementation of our algorithm is the size of the slice sets for each node in the formula tree. Therefore, we need a good representation for large sets of slices; furthermore, we need heuristics on when to discard irrelevant parts of the SAN for garbage collection.

We also extended the approach to a real-time logic similar to the logic **TNL** described in Section 7.3. In this setting, the application which is debugged and the debugger doing the model checking are running on different machines. The application sends debugging information to the model checker, which analyzes the timing constraints expressed by the logic. Here, the speed of the model checking and the transmission of protocol data over the local network is important. A solution which overcomes some of the timing bottlenecks is the automated testing approach described in the next chapter.

Chapter 12

Testing Reactive Real-Time Systems

Testing can be seen as another way of partially traversing the state space of a software system. By providing appropriate input stimuli, the system is forced to visit certain states, and responds with output signals which reveal some of the state information. Depending on whether this output allows to determine the internal state of the system, current methods for testing embedded real-time control software can be classified as structural or specification-based. *Structural testing* methods try to execute as many different parts of the *program code* as possible, where coverage is measured in terms of statements, conditionals, branches, function calls, and so on. *Specification based* methods treat the system under test as a black box and focus on testing the required *properties* of the system.

In contrast to verification or debugging, in testing we refrain from the idea of finding all errors or a certain known error in the system. The goal is to find as many errors as possible with a limited amount of effort. Therefore, it is important to develop techniques which allow to test as many state sequences of the system as can be reasonably expected. Several commercial tools for the systematic generation and execution of test cases exist. In these tools, the expected result of each test case usually is specified by a table. For parallel and distributed reactive systems this is not appropriate, since the correctness not only depends on the functional value, but also on the relative order of events. Recently, a number of researchers have proposed testing based on *formal specifications*. In particular, the testing theories of Hennessy [DH84, CH89, CH98], Tai [Tai85, TO86, TA87, TC96] and Peleska [Pel96a, Pel96b, PS96, PZ99, PS97] allow to generate test sequences

from formal modelling languages. In their approach, for each test case the result of the program run is checked whether it matches a description in some formal specification language. This method is closely related to conformance checking which is developed in Chapter 9. There, an I/O-module representing the specification is compared with an I/O-module representing the implementation; the comparison is done by connecting outputs of the (mirrored) specification with inputs of the implementation and vice versa. In specification based testing, not an abstract model but the actual embedded system is used for comparison with the specification. The specification is written in a mirrored fashion – outputs of the specification being direct stimuli for the system under test, and outputs of the system being inputs for the specification which can be used to detect nonconformance.

In Peleska's method, deterministic CSP terms (cf. Page 25) represent specifications of the behaviour of the system and its environment. They are compiled into a transition graph. Since the graph is deterministic, test sequences can be generated automatically and on the fly. The complete embedded system (hard- and software) is tested with these sequences. A test driver creates inputs for the system, while a test oracle checks the correctness of the system's outputs. In contrast to the usual test-script approach mentioned above, further continuation of each test is determined by these outputs and by previously tested sequences. A test monitor guarantees that all relevant test sequences are covered. Since the method is completely automatic, a high test coverage can be achieved.

In this chapter we report on experiences with this method on two practical examples — a satellite controller and a protocol stack layer. It turns out that the main problem in the practical use of formal methods in industrial examples is interfacing to an ever changing and constantly underspecified system. The satellite controller example is joint work with O. Meyer, the protocol stack example is joint work with J. Bredereke, both from TZi Univ. Bremen.

12.1 Description of the Systems

12.1.1 The ABRIXAS–PTC

The ABRIXAS X-ray satellite was built by OHB-System GmbH, Bremen, in cooperation with various scientific laboratories in Germany. Its mission was the first complete scan of the sky in the medium energy X-ray range up to 10 KEV. The system consisted of several modules: bus control, attitude control, camera control, and power and thermal control. The design was

highly redundant, each hardware component having at least one backup. Even the software was multiply loaded into different storage areas. However, this gives only protection from hardware-faults; correctness of the software is of equal importance. For example, a bug in the battery charge algorithm within the power and thermal controller could gradually reduce the capacity of the battery and thus eventually lead to a loss of the satellite. Therefore, extensive quality assurance was considered necessary in the construction of the software.

The main task of the ABRIXAS–PTC was to guarantee power supply of all active consumers. During the sun phase the energy should be generated by the solar strings; during the shade phase the battery should be used as a power source. The PTC had to control discharge and recharge of the battery. It further influenced power supply and temperature of various components (e.g., battery, mirror system, camera, attitude control, antennas). Another task was the central acquisition of a significant number of electrical and thermal data, and their transmission to the tracking station at the ground station. An overview on the components controlled by the PTC is given in Fig. 12.1.

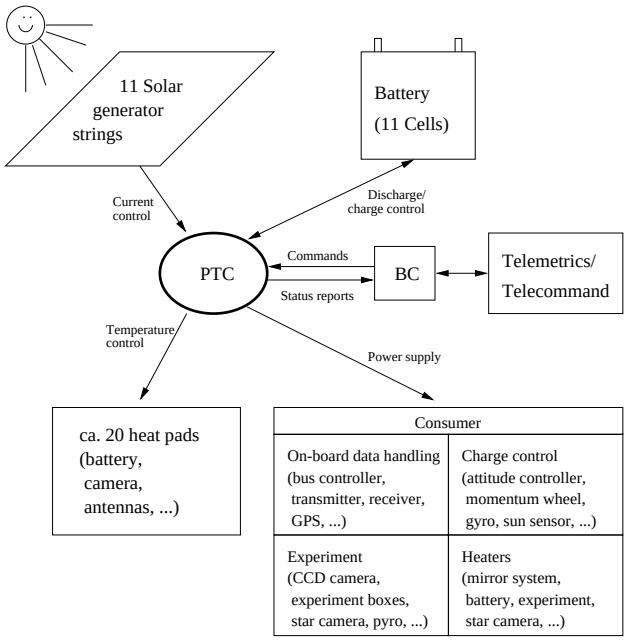


Figure 12.1: Environment of the ABRIXAS-PTC

The PTC was realized as a separate box consisting of a motherboard (PTC controller, PTCC) with several additional cards for data acquisition, energy distribution etc. The PTCC held a standard (space approved) 80C31 processor, which could access data from all other cards via a digital bus. The additional cards consisted of special circuitry to record voltage, current, pressure and temperature of various parts of the satellite.

The PTC software was coded in C and Assembler. It was configured with a set of tables written into the ROM. Input of the PTC were approximately 260 signals, i.e. analog measured data. The PTC controlled approximately 100 switches via the bus, and 70 open collector wires. It could receive approximately 20 different types of telecommands from the ground station, and transmit error and diagnostic reports. Furthermore, it had the possibility of latch-up control via the bus, and could communicate with the second (redundant) PTC. Fig. 12.2 gives an overview of the PTC interfaces.

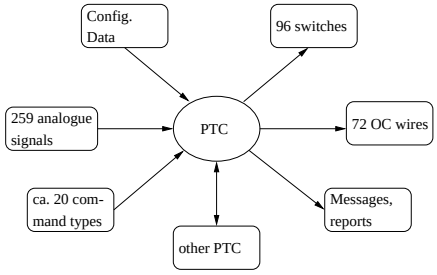


Figure 12.2: Interfaces of the ABRIXAS-PTC

The PTC software consisted of approximately 36 modules. A typical control task was to keep the temperature of a mirror system within a constant interval around 20°C. Another task was to control the charging of the battery during the sun phase, until the amount of energy discharged during the preceding shade phase was recharged and a certain battery pressure was reached. Since the hardware was highly redundant, this specification had to be satisfied even in case of certain hardware faults.

One problem in building software for space applications is that it is impossible to test the software in its real environment. For the purpose of testing the PTC the so called Power-SCOE (**S**ubsystem **C**heck-**O**ut **E**quipment) was built, which gave a hardware simulation of the environment of the controller. Basically, it consisted of an adjustable power supply as battery simulator, adjustable power sinks as consumer simulators, a special hardware solar generator simulator, and two PCs (COSMI and ADMEG) to compute the simulation values and for generating and measuring data.

12.1.2 The UMTS RLC layer

UMTS is (universal mobile telecommunication system) is one of the major new standards in digital telecommunication which will become the basis for third generation mobile phones in Europe. These systems will offer an increased bandwidth for high-speed transfer of multi media data. UMTS is being developed within the International Telecommunications Union (ITU IMT-2000) and is being standardized by the Third Generation Partnership Project (3GPP), in which all major telecommunication companies as well as the European Telecommunications Standards Institute (ETSI) take part.

The UMTS protocol stack software is supposed to be written in accordance with this open standard. The standard suggest a layered architecture, where each layer is defined by service primitives to and from upper and lower levels, respectively, and by the (virtual) peer-to-peer connection to the same layer of the other communication partner. Layer 1 is the physical layer: it guarantees services provided by the hardware, for example modulation of frequencies, multiplexing and demultiplexing of channels, and so on. Layer 2 consists of the media access control (MAC) and radio link control (RLC). MAC provides unacknowledged data transfer, physical reallocation of radio resources, and measurements such as traffic volume and quality indication. RLC has to guarantee services such as safe handover, connection establishment and connection termination. During operation, it exists in several instances. Layer 3 is responsible for services such as establishment and release of a connection and transfer of messages. It contains a module for radio resource control (RRC) which broadcasts information from the network to all reachable user devices and vice versa. Upper layers contain modules such as call control (CC) and mobility management (MM). An overview of this architecture is given in Figure 12.3.

In general, UMTS user equipment and the base stations are developed at different sites. For such systems, specification based testing is more appropriate than structural testing: in order to ensure inter-operability between devices from different providers, test suites should be based solely on the UMTS standard plus additional site-specific requirements, rather than on individual program code. The 3GPP standard is written partly in English and partly in SDL. From these souces, Siemens AG, Salzgitter, derives code for the RLC protocol stack layer of the user equipment with suitable software development tools. Independently from this, we formalized the requirements on the protocol for testing.

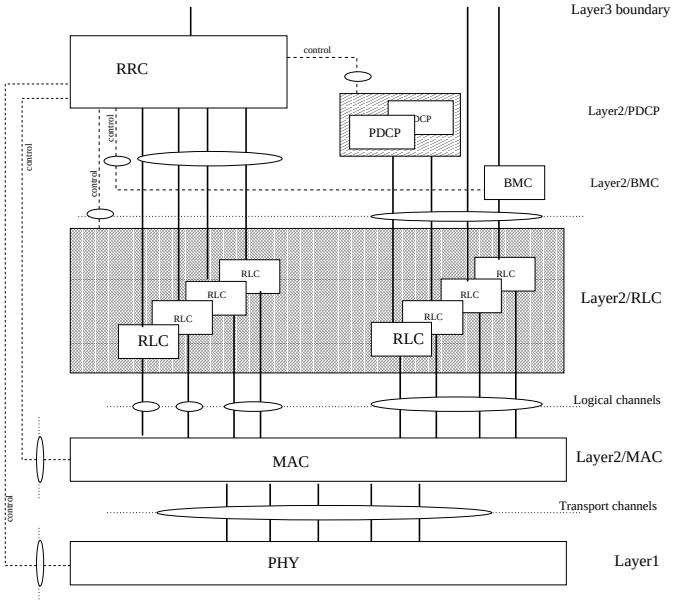


Figure 12.3: UMTS protocol stack architecture

12.2 Testing Setup

For the tests, in both cases the informal requirement documents were translated into a CSP description of the target system and its environment. From such specification the tool RT-TESTER (formerly VVT-RT) generates test sequences, which are executed by providing a stream of inputs and monitoring the corresponding outputs of the target.

For speed reasons, it is advantageous to have the test system running on a machine separate from the target. Usually, these are connected via specialized interfaces.

In the ABRIXAS project, a K6-200 PC was connected via standard ethernet with the COSMI-PC of the Power-SCOE. The COSMI PC interfaced the testing machine with the target and the hardware simulation of the environment (see Fig. 12.4). Licence terms forced us to generate the transition graph on a Solaris workstation. We then transferred the data to the VVT-RT PC, which executed and evaluated the tests under Linux. The adaptation of the TCP/IP based communication protocol of the testing machine to the specific formats used in the ABRIXAS project was the main effort in the project.

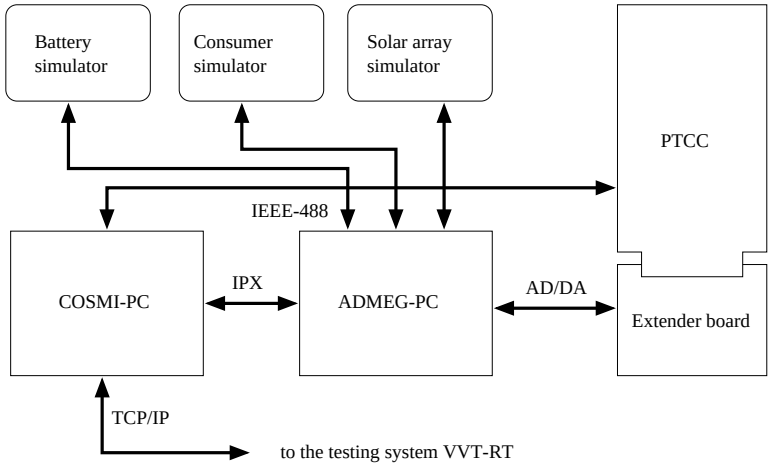


Figure 12.4: Power system checkout equipment

For the UMTS project, the same test specifications were to be used both for the development (host) platform in various integration stages and for the embedded target. Therefore, the setup was even more involved. Again, a special hardware simulation of the environment was considered necessary for high frequency wavetable emulation ("R&S-Tester"). Between the test engine and the target, we conceived a router which interfaced testing machine, target, and other equipment. The router connects to the system under test with the protocol stack running in an interpreter, in an emulated operating system, or on the actual embedded target. This setup is shown in Figure 12.5.

12.3 Formalization of the Requirements

In order to formally specify the system requirements, we first identified and classified properties describing the complete capacity of the system. Each of these properties corresponds to a class of functionalities to be tested. In both projects one of the main problems was that the requirements were constantly subject to change.

In the PTC case, changes of the requirements were usually verbally communicated and required manual adaption of the CSP terms. In some cases the required behaviour of the PTC was not yet defined. For example, the reaction to some combination of hardware faults was unspecified. Many of the

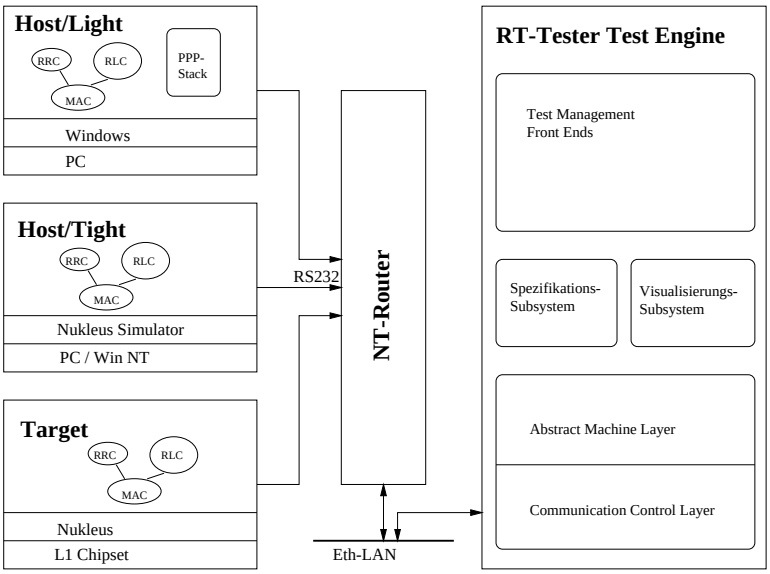


Figure 12.5: Testing setup for the UMTS example

problems we found in the initial phase were due to such incomplete specifications and, resulting from that, unpredictable control behaviour of the PTC. A feedback process with the system designers led to the documentation of requirements and an overall improvement of the implementation.

In the RLC case, for political reasons there were significant last-minute-changes to the 3GPP standard expected during the whole development process. Similarly, some details for the machine representation of data at the interfaces were not fixed until necessary. We therefore designed the testing environment to be highly flexible by

- defining the interface in terms of SDL signals and data structures instead of low level descriptions,
- performing an automated consistency check between the SDL description of the interface and the formal CSP specification of the interface, and
- modularizing the formal behaviour specifications into largely independent functional requirements.

We developed a generator tool that takes the SDL and the CSP descriptions of the system under test, and automatically generates the code for the

interface adapter of the testing tool. If the machine representation of the code changes due to a different selection of hardware, then it is sufficient to re-compile the interface code. If the requirements in the UMTS standard change, then it is sufficient to adapt the descriptions in SDL and CSP, and to re-generate the interface code. The new test scripts then are generated automatically. The generator also performs consistency checks between the interface descriptions; e.g., it checks whether the parameter types of SDL and CSP signals are consistent. With the RLC protocol layer this feature is especially important: this module uses large and complex data structures as signal parameters, which are difficult to keep in sync manually. There are signals with more than one kilobyte of heavily structured parameter data; comparing their definitions manually would be extremely tedious and error-prone.

Another preparative to cope with changing requirements is a modular structure of the formal behaviour specification. CSP allows to specify different aspects of a system separately, and to combine these specifications by suitable composition operators. We therefore strived to describe each property of the system under test in a separate requirements module. If a change of requirements affects only one particular aspect of the system, the necessary adaptions in the CSP code are restricted to a single module. This improved the maintainability of the test suites. Furthermore, some crucial aspects of the system under test had to be tested more thoroughly than others. In the RLC case, we also composed suitably tailored instances of the above modules to complex test suites, and run them using the testing tool. In all these cases, the actual test scripts are generated automatically on the fly.

We now describe the formalization of requirements for the ABRIXAS PTC project in more detail. These examples are from [SMH99]. There were three groups of requirements:

- requirements for switching functions,
- requirements for the energy control, and
- requirements for the thermal control.

As an example for a switching requirement we mention the telecommand to turn a consumer on or off. This command is delivered to the PTC via the bus controller. The PTC checks the command for syntactical and semantical consistency. Usually, in order to turn a consumer on, several switches have to be toggled. The PTC executes the command by putting a sequence

of appropriate signals onto the bus. It then checks via voltage and current sensors whether the operation was correctly performed, and sends an acknowledgement or error message to ground control. A simple switching requirement is given in Fig. 12.6.

At any given moment, it is possible to send telecommands for turning any consumer on or off. All switching operations neccessary to activate or deactivate this consumer must be performed within a given time constant `T_Swtch_Consumer`.

Figure 12.6: Requirement for Switching Functionality

An example requirement for the energy control is the central part of the battery charge control. The satellite battery is highly delicate and must be charged and decharged according to certain manufacturer instructions. Charging is done in two phases: main charge and supplementary charge. During main charge it is important to quickly recharge the battery with a high current, while during supplementary charge the average pressure and temperature of the battery influence the charging. The requirement for the main charge phase is given in Fig. 12.7.

During the sun phase the battery is charged with a charge current of `I_Charge`, until the amount of energy consumed during the last shade phase is recharged, and the minimal absolute pressure `P_minabs` is reached or the sun phase ends.

Figure 12.7: Requirement for the Energy Control

A third example is the requirement for the thermal control of the CCD camera. The PTC has to regulate the temperature with electric heat pads, such that it is as stable as possible; in any case it must remain within a certain interval. In Fig. 12.8 the normal behaviour is described. This simplified requirement could be implemented with a simple hardware switch. The complete requirement, which includes the desired behaviour in case of hardware faults and insufficient power supply, requires a nontrivial switching logic and is beyond the scope of this book.

In order to automatically generate a (usually very large) number of test cases and test data from these requirements, we formulated them in the specification language CSP. Although some expert knowledge is required to

The temperature of the CCD camera is always within fixed bounds. If the camera is turned on, its heater must be deactivated. If the camera is turned off, the following rules apply: If the temperature of the CCD is less than H_CCD_opt-H_CCD_tol, the corresponding heater is switched on. If the temperature of the CCD is greater than H_CCD_opt+H_CCD_tol, the corresponding heater is switched off.

Figure 12.8: Requirement for the Thermal Control

write such formal specifications, the process is comparable to programming in a high level programming language and could be done by system engineers. First, the interfaces of each requirement were listed in a systematic way. Then dependencies between interfaces were located and the requirements were grouped in several classes. For each analog channel bounds were defined, such that the transgression of these bounds leads to the generation of an event. The event mapping library was programmed and the communication software connecting VVT-RT to the Power-SCOE was implemented.

```
SPEC = ( SWITCHCONS [|{| Tau_nextTC |}|] TCTIM ) |||  TIMCHK

SWITCHCONS =  Tau_nextTC -> ( (Com_PYRO_PWR_CONS_ON -> setTimSwt
        -> Swt_BS_ON_MAIN_ON -> Swt_PYRO_PRE_MAIN_ON
        -> Swt_PYRO_PWR_MAIN_ON -> resTimSwt -> SWITCHCONS)
 |~| (Com_PYRO_PWR_CONS_OFF -> setTimSwt -> Swt_PYRO_PWR_MAIN_OFF
        -> resTimSwt -> SWITCHCONS)
 |~|  ... )

TIMCHK = elaTimSwt -> errorSwitchTimer -> TIMCHK

TCTIM = Tau_nextTC -> setTimTick -> elaTimTick -> TCTIM
```

Figure 12.9: CSP-code for the requirement in Fig. 12.6

Then the requirements were formalised in CSP. As examples we give the CSP code for the above requirements. Fig. 12.9 shows part of the formulation of the requirement in Fig. 12.6. The formal specification SPEC consists of three parallel subprocesses SWITCHCONS, TIMCHK and TCTIM. Process SWITCHCONS nondeterministically chooses a consumer to be turned on or off. (In the figure, we only display the consumer PYRO which is used to open the cover of the mirror system.) The process then generates an event Com_PYRO_PWR_CONS_ON or Com_PYRO_PWR_CONS_OFF, which is translated

by the event mapping library into an appropriate telecommand for turning the pyro on or off. Then a timer `TimSwt` is activated which supervises the required time bound `T_Swtch_Consumer`. For turning on the pyro, the PTC has to switch the three consecutive switches `Swt_BS_ON_MAIN_ON`, `Swt_PYRO_PRE_MAIN_ON` and `Swt_PYRO_PWR_MAIN_ON`. Process `SWITCHCONS` waits for the corresponding events. If they arrive in time, it resets the timer `TimSwt`. To turn the pyro off it suffices to switch `Swt_PYRO_PWR_MAIN_OFF`. All tests are done in an endless loop, which is realised by a recursive call. Process `TIMCHK` is executed interleaved with process `SWITCHCONS`. If the timer `TimSwt` elapses, VVT-RT generates the event `elaTimSwt` which is delivered to this process. In this case, it generates an appropriate error event `errorSwitchTimer` for the test oracle. This way, it is possible to detect timing errors without stopping or restarting the PTC.

First experiments with this CSP code revealed that the PTC could not satisfy the requirement. The reason was that the test system generated telecommands too quickly. As soon as a switch was toggled by the PTC, the test system without delay asked for the next consumer to be switched. Sometimes the PTC failed to notice these commands. In reality, where commands are emitted from human operators at ground control, it is reasonable to assume that such a frequency of commands can not be reached. Therefore, the informal requirement that "at any given moment" telecommands must be accepted was supplemented. We assumed that at most one telecommand per second is sent. In the CSP specification, this assumption is realised by an additional timing process `TCTIM`, which is synchronized with process `SWITCHCONS` via the internal event `Tau_nextTC`. It delays the sending of new commands by the assumed bound. With this modified specification, the PTC was fast enough to pass the switching tests.

Fig. 12.10 shows the formalisation of the requirement for the main charge phase of the power control. Charging starts with the main charge phase as soon as the sun rises. Within the time bounds defined by timer `TimChargeControl` the PTC has to adjust the charge current to the value `I_Charge`. Whenever this value is reached (within a certain precision) VVT-RT generates the event `Evt_I_BATT_MAIN_IS_I_Charge`. If process `MAINCHARGE` receives this event before timer `TimChargeControl` elapses, it resets the timer; if the timer elapses before the required charge current is reached a test error is signalled. If during main charging the charge current leaves the specified precision range around `I_Charge` due to switching operations, the timer is set again.

It is required that main charging continues until the amount of energy discharged during the last shade phase is recharged and the minimal abso-

```
CHARGECONTROL = Tau_SUN_ON -> setTimChargeControl
                           -> MAINCHARGE(false, false)

MAINCHARGE(PressureOK, Recharged) =
    (Evt_I_BATT_MAIN_IS_I_Charge
       -> resTimChargeControl -> MAINCHARGE(PressureOK, Recharged))
 [] (elaTimChargeControl -> errorChargeControl
       -> setTimChargeControl -> MAINCHARGE(PressureOK, Recharged))
 [] (Evt_I_BATT_MAIN_ISNOT_I_Charge
       -> setTimChargeControl -> MAINCHARGE(PressureOK, Recharged))
 [] (Evt_P_BATT_PRESS_1_OK ->
       if Recharged then SUPPLEMENTARYCHARGE
       else MAINCHARGE(true, Recharged))
 [] (Evt_Recharged ->
       if PressureOK then SUPPLEMENTARYCHARGE
       else MAINCHARGE(PressureOK, true))
 [] (Tau_SUN_OFF -> DISCHARGECONTROL)
```

Figure 12.10: CSP-code for the requirement in Fig. 12.7

lute battery pressure is reached, or until the sun phase ends. In our CSP encoding of this requirement, the state variables "recharged" and "pressure O.K." are realised by boolean parameters of the process `MAINCHARGE`. The event `Evt_P_BATT_PRESS_1_OK` is generated by the test system whenever the required pressure range is reached. If this event reaches the process where the state variable `Recharged` is false, then the PTC should still be in the main charge phase. If the process is in state `Recharged`, then the supervision of supplementary charge begins. The test system generates the event `Evt_Recharged` if the integral of the charge current reaches the integral of the discharge current during the last shade phase. This way, arbitrary complex hybrid properties can be tested. The CSP code for the main charge is given in Fig. 12.10.

For conciseness, the formalisation of the requirement for the CCD camera thermal control is just sketched here. The temperature of the camera is simulated by an algorithm which runs in parallel with the test system. The CSP processes in Fig. 12.11 describe both the behaviour of the environment upon activation or deactivation of heat pads, and the required behaviour of the CCD heater control algorithm. Internal events (starting with `Evt_`) are used to synchronise the test system with the temperature simulation algorithm. For example, whenever at least one of both heaters (main or redundant) is active, the simulation has to choose a temperature curve which models the warm up of the camera (`Evt_warmer`). Similar to above,

```
ED = TC2_CCD(ok,false,false)
-- 1st parameter: temperature state, initially o.k.
-- 2nd parameter: heater MAIN state, initially off
-- 3rd parameter: heater REDU state, initially off
-- Thermal Control Testcase2, Inner1

TC2_CCD(temp,main,redu) =
      Swt_CCD_HEAT_MAIN_ON -> Evt_warmer -> TC2_CCD(temp,true,redu)
   [] Swt_CCD_HEAT_REDU_ON -> Evt_warmer -> TC2_CCD(temp,main,true)
   [] Swt_CCD_HEAT_MAIN_OFF ->
       (if (not redu) then Evt_colder -> SKIP else SKIP);
       TC2_CCD(temp,false,redu)
   [] Swt_CCD_HEAT_REDU_OFF ->
       (if (not main) then Evt_colder -> SKIP else SKIP);
       TC2_CCD(temp,main,false)
   [] Evt_too_warm -> errorTooWarm -> setTim1
       -> TC2_CCD(temp,main,redu)
   [] Evt_warm -> setTim1 -> TC2_CCD(warm,main,redu)
   [] Evt_ok -> resTim1 -> TC2_CCD(ok,main,redu)
   [] Evt_cold -> setTim1 -> TC2_CCD(cold,main,redu)
   [] Evt_too_cold -> errorTooCold -> setTim1 -> TC2_CCD(temp,main,redu)
   [] elaTim1
       -> if ( (temp==warm and (main or redu))
            or (temp==cold and (not (main or redu))))
          then (errorTooLate -> setTim1 -> TC2_CCD(temp,main,redu))
          else TC2_CCD(temp,main,redu)
   [] Swt_EXP_PWR_MAIN_ON
       -> (  (Swt_CCD_HEAT_MAIN_OFF -> Swt_CCD_HEAT_REDU_OFF -> SKIP)
          [] (Swt_CCD_HEAT_REDU_OFF -> Swt_CCD_HEAT_MAIN_OFF -> SKIP));
       Evt_constOK -> TC2_CCD_ON(main)
   [] Swt_EXP_PWR_REDU_ON
       -> (  (Swt_CCD_HEAT_MAIN_OFF -> Swt_CCD_HEAT_REDU_OFF -> SKIP)
          [] (Swt_CCD_HEAT_REDU_OFF -> Swt_CCD_HEAT_MAIN_OFF -> SKIP));
             Evt_constOK -> TC2_CCD_ON(redu)

TC2_CCD_ON(MAIN_REDU) =
      Swt_CCD_HEAT_MAIN_ON -> errorHeaterOn -> TC2_CCD_ON(MAIN_REDU)
   [] Swt_CCD_HEAT_REDU_ON -> errorHeaterOn -> TC2_CCD_ON(MAIN_REDU)
   [] Swt_EXP_PWR_MAIN_OFF ->
       (if (MAIN_REDU==main)
        then (Evt_colder -> TC2_CCD(ok,false,false))
        else TC2_CCD_ON(MAIN_REDU))
   [] Swt_EXP_PWR_REDU_OFF ->
        (if (MAIN_REDU==redu)
         then (Evt_colder -> TC2_CCD(ok,false,false))
         else TC2_CCD_ON(MAIN_REDU))
```

Figure 12.11: CSP-code for the requirement in Fig. 12.8

VVT-RT checks that certain bounds are respected. For example, if the first upper limit (`H_CCD_opt` + `H_CCD_tol`) is reached, the simulation component generates the event `Evt_warm`. In this case, the specification requires the deactivation of both heat pads within a certain time. Since the temperature should always stay within certain bounds, the event (`Evt_too_warm`) is generated whenever the upper bound is reached and leads to a test error.

The temperature control should only be active when the camera is off. If the camera is turned on (`Swt_EXP_PWR_xxx_ON`), then the waste heat suffices to warm it. Process `TC2_CCD_ON` checks that in this case the heat pads are never on. Event `Evt_constOK` causes the simulation to assume that the temperature of the operating camera is constant.

12.4 Results of Testing

The CSP test procedures were automatically executed by the test driver with the above configuration. The test results could both be interactively compared to the expected results and automatically evaluated.

We already mentioned that in the PTC case a number of requirements were neither fixed nor documented by the designers. Similarly, for the RLC, in writing the formal requirements specification and comparing its interface to the implementation's interface, we found several ambiguities in the standards document. They can be subject to different, equally legal, incompatible interpretations. These spots will need special care to avoid interoperability problems with software developed at other sites or by different manufacturers. Furthermore, we also found a few discrepancies between the implementation's interface definition and the interface defined in the standard.

Small bugs were also found in table entries and conflicting versions of the configuration data. An example from the PTC is a wrong entry in a table which concerned the number of switches to be activated for opening the cover of the camera with the pyro. Thus, when executing the command to open the cover, only one of two necessary switches was toggled; therefore, additional user interaction would have been necessary to start the experiment.

There were several situations in which the RLC did not behave as expected. For example, in a certain state the RLC reacts to a certain signal where no reaction was intended. With structural testing, probably no explicit test script would have been written that checks for a non-reaction in this state. The systematic random exploration of the state space in our approach found this problem automatically. Furthermore, the tests revealed

interactions between different instances of the RLC protocol machines. The requirements allow different instances of these machines which behave completely independent. Each instance could be tested separately. But we also performed a test where several protocol machines were tested at the same time, each one against its own copy of the requirements specification. It turned out that in such a setup there was the possibility that the entire system under test could deadlock. Another interesting observation showed up only intermittently, after a certain history of input: even though the RLC always *should* have gone into the same state, it did not. The on-the-fly generation of the test scripts allowed us to run the test suite for a long period of time, which was necessary to detect this problem.

We also found problems in the C code of the PTC software. For example, a wrong sign in the statement calculating the necessary charge duration had as consequence that the difference between discharge and recharge amount was always negative. Thus, in contrast to the above specification, the minimal absolut battery pressure was the only criterion for the charging. In case of a fault of the pressure sensors there was no redundant way to determine the necessary charge duration.

Furthermore, in the PTC case an incompatibility between hardware and software was revelead: The software to store updated parameter tables or new software in the E^2proms during the mission was too fast for the hardware. Although the software timinig was correct with respect to the specificiation of the E^2proms, the actually written bytes where sometimes different from the source. It turned out that the employed E^2proms did not meet their specification; the necessary delays between consecutive write commands were longer than expected. This error demonstrates the importance of hardware-in-the-loop testing at system level even if the software is proven to work correctly: The communication between hardware and software might introduce new problems.

Specification-based testing thus mainly discovered problems in the exceptional behaviour, which "normally" never occurs. For safety critical applications it is required that the systems is reliable even in unforeseen circumstances. Most problems could be immediately solved. Incomplete specifications were updated in the requirements document and changed in the formal specification. Since the testing was completely automatic, regression tests could be performed for the improved software after compilation and loading without further efforts. In fact, in the PTC case in late stages of the project formal testing replaced debugging in the software development process.

Bibliography

[ABH+97] R. Alur, R.K. Brayton, T. Henzinger, S. Quadeer, and S.K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Proc. 9th Int. Conf. on Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 340–351, Haifa, Israel, June 1997. Springer.

[ABL98] L. Aceto, A. Bergueno, and K. Larsen. Model checking via reachability testing for timed automata. In B. Steffen, editor, *Proc. 4th Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, volume 1384 of *LNCS*, pages 281–297, Lisbon, Mar–Apr 1998. Springer.

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model–checking for real–time systems. In *Proc. 5th Ann. IEEE Symp. on Logic in Computer Science (LICS '90)*, pages 414–425. IEEE Comp. Soc. Press, June 1990.

[AD90] R. Alur and D. Dill. Automata for modelling real–time systems. In *Proc. 17th Int. Conf. on Automata, Languages and Programming (ICALP '90)*, volume 443 of *LNCS*, pages 322–335. Springer, 1990.

[AD97] P. Amthor and S. Dick. Test eines Bordcomputers für ein dezentrales Zugsteuerungssystem unter Verwendung des Werkzeuges VVT-RT. In *7. Kolloquium Software-Entwicklung - Methoden, Werkzeuge, Erfahrungen: Mächtigkeit der Software und ihre Beherrschung*, September 1997.

[AG87] M. Ajtai and Y. Gurevich. Monotone versus positive. *Journal of the ACM*, 34:1004–1015, 1987.

[AH92] R. Alur and T.A. Henzinger. Logics and models of real–time: A survey. In *Real–Time: Theory in Practice*, LNCS. Springer, 1992.

[AH94] R. Alur and T.A. Henzinger. A Really Temporal Logic. *Journal of the ACM*, 41(1):181–204, January 1994.

[AH96] M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *IEEE Real–Time Technology and Applications Symp. (RTAS'96)*, Boston MA, June 1996. IEEE Computer Society Press.

[AHK97] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time Temporal Logic. In *Proceedings of the Symposium on Foundations of Computer Science*, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison–Wesley, 1974.

[ALM96] M. Abadi, L. Lamport, and S. Merz. A TLA solution to the RPC–memory specification problem. In M. Broy, S. Merz, and K. Spies, editors, *Formal System Specification: The RPC–Memory Specification Case Study*, volume 1169 of *LNCS*, pages 21–66. Springer, 1996.

[Alu91] R. Alur. *Techniques for Automatic Verification of Real–Time Systems*. PhD thesis, Stanford University, 1991.

[Alu98] R. Alur. Timed automata. In *Verification of Digital and Hybrid Systems*, NATO ASI Summer School Series. Springer, 1998.

[AM96] M. Abadi and S. Merz. On TLA as a logic. In M. Broy, editor, *Deductive Program Design : Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, 26. July– 7. August 1994*, volume 152 of *NATO ASI series F, Computer and Systems Sciences*, pages 235–272, Berlin-Heidelberg-New York-Tokyo, 1996. Springer.

[And92] H.R. Andersen. Model checking and Boolean graphs. In B. Krieg-Brückner, editor, *Proceedings of 4th European Symposium on Programming (ESOP '92), Rennes, France, 26.–28. Februar 1992*, volume 582 of *Lecture Notes in Computer Science*, pages 1–19, Berlin-Heidelberg-New York-Tokyo,, 1992. Springer.

[And94] H.R. Andersen. On model checking infinite–state systems. In A. Nerode and Matiyasevich, editors, *Logic at St. Petersburg. Symp. on Logical Foundations of Computer Science (LFCS '94)*, volume 813 of *LNCS*, St. Petersburg, Russia, July 11–14, 1994. Springer.

[Anu95] A. Anuchitanukul. *Synthesis of Reactive Programs*. PhD thesis, Stanford, 1995.

[APP95] R. Alur, D. Peled, and W. Penczek. Model-checking of causality properties. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science, San Diego, California, 26.–29. Juni 1995*, pages 90–100, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[AS85] B. Alpern and F. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, October 1985.

[ASB+95] A. Aziz, V. Singhal, F. Balarin, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. It usually works – the temporal logic of stochastic systems. In Pierre Wolper, editor, *Proc. 7th Workshop on Computer Aided Verification (CAV '95)*, volume 939 of *LNCS*, pages 155–166. Springer, 1995.

[ASSB96] A. Aziz, K. Sanwal, V. Singhal, and R.K. Brayton. Verifying continuous Markov chains. In R. Alur and T. Henzinger, editors, *Proc. 8th Workshop on Computer Aided Verification (CAV '96)*, volume 1102 of *LNCS*, pages 269–276. Springer, 1996.

[ASW94] H.R. Andersen, C. Stirling, and G. Winskel. A compositional proof system for the modal μ–calculus. In *Proc. 9th Ann. IEEE Symp. on Logic in Computer Science (LICS '94)*, pages 144–153, Paris, France, July 1994. IEEE Computer Society Press. BRICS Report RS–94–34.

[BAMP83] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[BAN89] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. Technical Report 39, DEC Systems Research Center, February 1989.

[Bat88] P. Bates. Distributed debugging tools for heterogeneous distributed systems. In *Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, CA, 13.–17. Juni 1988*, pages 308–315, Los Alamitos, CA., 1988. IEEE Computer Society Press.

[Bat95] P.C. Bates. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behaviour. *ACM Tarns. on Computer Systems*, 13(1):1–31, February 1995.

[BBC+95] N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Zohar Manna, H.B. Simpa, and T.E. Uribe. STeP: The Stanford theorem prover – user's manual. Technical Report STAN–CS–TR–95–1562, Department of Computer Science, Stanford University, 1995.

[BBC+96] N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Zohar Manna, H.B. Sipma, and T.E. Uribe. STeP: Deductive–algorithmic verification of reactive and real–time systems. In *Proc. 8th Workshop Computer Aided Verification (CAV '96)*, volume 1102 of *LNCS*. Springer, 1996.

[BBLS92] S. Bensalem, A. Bouajani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In G.V. Bochmann and D.K. Probst, editors, *Proc. 4th Int. Conf. on Computer Aided Verification (CAV '92)*, 1992.

[BC86] M.C. Browne and E.M. Clarke. SML: A high level language for the design and verification of finite state machines. In *IFIP WG 10. 2 Int. Working Conf. from HDL Descriptions to Guaranteed Correct Circuit Designs*, Grenoble, France, September 1986. IFIP.

[BC95] G. Bhat and R. Cleaveland. Efficient local model-checking for fragments of the modal μ-calculus. In *Proceedings of the Tenth Annual Symposium on Logic in Computer Science (LICS'95), San Diego, Juni 1995*, pages 388–397, Los Alamitos, CA, 1995. IEEE Computer Society Press. Also in Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96), T. Margaria and B. Steffen, eds, Springer LNCS 1055, pp.107–126 1996.

[BCD85] M.C. Browne, E.M. Clarke, and D. Dill. Checking the correctness of sequential circuits. In *Proc. 1985 Int. IEEE Conf. on Computer Design*, Port Chester, New York, October 1985. IEEE.

[BCD86] M.C. Browne, E.M. Clarke, and D. Dill. Automatic circuit verification using temporal logic: Two new examples. In *Formal Aspects of VLSI Design*. Elsevier Science Publishers (North Holland), 1986.

[BCD+94] J.R. Burch, E.M. Clarke, D. Dill, D.E. Long, and K.L. McMillan. Symbolic model checking for sequential circuit verification. *IEEE Trans. on Computer Aided Design of Integrated Circuits*, 13(4):401–424, 1994.

[BCDM86] M. Browne, E.M. Clarke, D. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Trans. on Computers*, C-35(12):1035–1044, December 1986.

[BCFZ99] A. Biere, A. Cimatti, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36th ACM/IEEE Design Automation Conference (DAC '99)*, 1999.

[BCG88] M.C. Browne, E.M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1–2):115–131, July 1988.

[BCG89] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite–state processes. *Information and Computation*, 81(1):13–31, April 1989.

[BCG+92] J.R. Burch, E.M. Clarke, O. Grumberg, D.E. Long, and K.L. McMillan. Automatic verification of sequential circuit designs. *Phil. Trans.R. Soc. Lond. A*, 339:105–120, 1992.

[BCHG+97] C. Baier, E.M. Clarke, V. Hartonas-Garmhausen, M. Kwiatkowska, and M. Ryan. Symbolic model checking for probabilistic processes. In *Proc. Int. Conf. on Automata, Languages and Programming (ICALP '97))*, volume 1256 of *LNCS*, pages 430–437, 1997.

[BCJM96] S. Berezin, E.M. Clarke, S. Jha, and W. Marrero. Model checking algorithms for the μ–calculus. Technical Report CMU–CS–96–180, CMU, September 1996.

[BCL91a] J.R. Burch, E.M. Clarke, and D.E. Long. Representing circuits more efficiently in symbolic model checking. In *Proc. 28th ACM/IEEE Design Automation Conference (DAC '91)*, 1991.

[BCL91b] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *Proc. Int. Conf. on Very Large Scale Integration (VLSI '91)*, Edinburgh, Scotland, August 1991.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992. also in 5th IEEE LICS 90.

[BCMD90] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Sequential circuit verification using symbolic model checking. In *Proc. 27th ACM/IEEE Design Automation Conference (DAC '90)*, 1990.

[BCZ99] A. Biere, A. Cimatti, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, volume 1579 of *LNCS*. Springer, 1999.

[BD91] B. Berthomieu and M. Diaz. Modelling and verification of time dependent systems using Time Petri Nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, March 1991.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. KRONOS: A model-checking tool for real-time systems. In A.J. Hu and M.Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification*, Vancouver, BC, Canada, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550. Springer-Verlag, June/July 1998.

[BdRV01] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Elsevier, 2001.

[BE97] O. Burkart and J. Esparza. More infinite results. *Electronic Notes in Theoretical Computer Science*, 6, 1997. http://www. elsevier. nl/locate/entcs/volume6. html.

[Ber91] C.L. Bermann. Circuit width, register allocation and ordered binary decision diagrams. *IEEE Trans. on Computer–Aided Design*, 10(8):1059–1066, 1991.

[BFG+93] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proc. Int. Conf. on Computer Aided Design (ICCAD '93)*, pages 188–191, Santa Clara, November 1993.

[BFV86] F. Baiardi, N. De Francesco, and G. Vaglini. Development of a Debugger for a Concurrent Language. *IEEE Transactions on Software Engineering*, 12(4):547–553, April 1986.

[BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using qdds. In Rajeev Alur and T. Henzinger, editors, *Proc. 8th Workshop on Computer Aided Verification (CAV '96)*, volume 1102 of *LNCS*, pages 1–12. Springer, Jul./Aug. 1996.

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[Bie97] A. Biere. *Effiziente Modellprüfung des μ–Kalküls mit binären Entscheidungsdiagrammen.* PhD thesis, University of Karlsruhe, Germany, January 1997.

[BKPS97] B. Buth, M. Kouvaras, J. Peleska, and H. Shi. Deadlock analysis for a fault-tolerant system. *Lecture Notes in Computer Science*, 1349:60–75, December 1997.

[BLL+96] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and Wang Yi. UPPAAL: a tool suite for the automatic verification of real-time systems. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer-Verlag, 1996.

[BMS95] J. Bern, C. Meinel, and A. Slobodovà. Global rebuilding of BDDs – avoiding the memory requirement maxima. In Pierre Wolper, editor, *Proc. 7th Workshop on Computer Aided Verification (CAV '95)*, volume 939 of *LNCS*, pages 4–15. Springer, 1995.

[Bor97] A. Borälv. The industrial success of verification tools based on Stålmarck's method. In O. Grumberg, editor, *Proc. 9th Workshop on Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*. Springer, 1997.

[BPS99] B. Buth, J. Peleska, and H. Shi. Combining methods for the livelock analysis of a fault-tolerant system. *Lecture Notes in Computer Science*, 1548:124–139, January 1999.

[BRB90] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a BDD package. In *Proc. 27th ACM/IEEE Design Automation Conference (DAC '90)*, pages 40–45, 1990.

[Bry86] R.E. Bryant. Graph–based algorithms for Boolean function manipulation. *IEEE Trans. on Computers*, C–35(8):677–691, 1986.

[Bry91] R.E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Trans. on Computers*, 40(2):205–213, 1991.

[Bry92] R.E. Bryant. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–317, 1992.

[BS91] J. Bradfield and C. Stirling. Local model checking for infinite state spaces. In *Proc. 3rd Workshop on Computer Aided Verification (CAV '91)*, LNCS. Springer, 1991.

[BS00] B. Brard and L. Sierra. Comparing verification with HYTECH, KRONOS and UPPAAL on the railroad crossing example. Research Report LSV-00-2, Lab. Specification and Verification, ENS de Cachan, Cachan, France, January 2000. 14 pages.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Int. Congr. Logic, Method and Philosophy of Science 1960*, pages 1–12, Palo Alto, CA, USA, 1962. Stanford University Press.

[Büc65] J.R. Büchi. Transfinite automata recursions and weak second order theory of ordinals. In *Proc. Int. Congr. on Logic, Methodology and Philosophy of Science 1964*, pages 2–23, Amsterdam, 1965. North Holland.

[Bur74] M. Burstall. Program proving as hand simulation with a little induction. In *Proc. IFIP Congress, Stockholm*, pages 308–312. North Holland, 1974.

[Bur84] J. Burgess. Basic tense logic. In F. Guenthner D. Gabbay, editor, *Handbook of Philosophical Logic*, chapter II. 2, pages 89–134. Reidel, 1984.

[BW83] P.C. Bates and J.C. Wileden. High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach. *The Journal of Systems and Software*, 3(4):255–264, 1983.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th Ann. ACM Symp. on Principles of Programming Languages (POPL '77)*, 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. 6th Ann. ACM Symp. on Principles of Programming Languages (POPL '79)*, 1979.

[CD88] E.M. Clarke and I.A. Draghicescu. Expressibility results for linear time and branching time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 428–437. Springer, May 1988.

[CDK90] E.M. Clarke, I.A. Draghicescu, and R.P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω–automata. In A. Arnold and N.D. Jones, editors, *Proc. 15th Coll. on Trees in Algebra and Programming*, volume 407 of *LNCS*. Springer, 1990.

[CE81] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Proc. Workshop on Logic of Programs*, volume 131 of *LNCS*, Yorktown Heights, NY, 1981. Springer.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite–state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CFH97] E.M. Clarke, M. Fujita, and W. Heinle. Hybrid spectral transform diagrams. In *Proc. 1st Int. Conf. on Information, Communications and Signal Processing (ICICS '97)*, 1997.

[CFJ93] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In C. Courcoubetis, editor, *Proc. 5th Workshop on Computer Aided Verification (CAV '93)*, volume 697 of *LNCS*, Elounda, Crete, June 1993. Springer.

[CFM+93] E.M. Clarke, M. Fujita, P. McGeer, J. Yang, and X. Zhao. Multi–terminal binary decision diagrams: An efficient data structure for matrix representation. In *Proc. Int. Workshop on Logic Synthesis (IWLS '93)*, Tahoe City, May 1993.

[CFZ95] E.M. Clarke, M. Fujita, and X. Zhao. Hybrid decision diagrams —- overcoming the limitations of MTBDDs and BMDs. In *Proc. IEEE Int. Conf. on Computer Aided Design (ICCAD '95)*, pages 54–60. IEEE Computer Society Press, 1995.

[CFZ96] E.M. Clarke, M. Fujita, and X. Zhao. Multi–terminal binary decision diagrams and hybrid decision diagrams. In T. Sasao and M. Fujita, editors, *Representations of Discrete Functions*, chapter 4, pages 93–108. Kluwer academic publishers, 1996.

[CG87a] E.M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal model checking algorithms. In *Proc. 6th Ann. ACM Symp. on Principles of Distributed Computing*, pages 294–303, 1987.

[CG87b] E.M. Clarke and O. Grumberg. Research on automatic verification of finite–state concurrent systems. Technical Report CMU-CS-87-105, Carnegie Mellon University, January 1987.

[CGB86] E.M. Clarke, O. Grumberg, and M.C. Browne. Reasoning about networks with many identical finite-state processes. In *Proc. 5th Ann. ACM Symp. on Principles of Distributed Computing*, pages 240–248. ACM, August 1986.

[CGH+93] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ cache coherence protocol. In L. Claesen, editor, *Proc. 11th Int. Symp. on Computer Hardware Description Languages and their Applications*. North–Holland, 1993.

[CGH97] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10:47–71, 1997.

[CGJ95] E.M. Clarke, O. Grumberg, and S. Jha. Parametrized networks. In S. Smolka and I. Lee, editors, *Proc. 6th Int. Conf. on Concurrency Theory (CONCUR '95)*, volume 962 of *LNCS*. Springer, 1995.

[CGL93] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking. In M. Broy, editor, *Deductive Program Design*, pages 305–350. Springer NATO ASI series F, 1993.

[CGL94a] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994. also in 19th ACM POPL '92.

[CGL94b] E.M. Clarke, O. Grumberg, and D.E. Long. Verification tools for finite–state concurrent systems. In J.W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency – Reflections and Perspectives*, volume 803 of *LNCS*, pages 124–175. Springer, 1994. REX School/Symposium, Nordwijkerhout, The Netherlands, June 1993.

[CGMP99] E.M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reductions using partial order techniques. *Int. Journal on Software Tools for Technology Transfer*, 1999. to appear.

[CGMZ94] E.M. Clarke, O. Grumberg, Ken McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. Technical Report CMU–CS–94–204, Carnegie Mellon University, Pittsburgh, 1994.

[CGP99] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Boston, MA., 1999.

[CH80] A. Chandra and D. Harel. Computable queries for relational databases. *J. of Computer and System Sciences*, 21:156–178, 1980.

[CH89] R. Cleaveland and M. Hennessy. Testing equivalence as a bisimulation equivalene. Technical Report 4/89, Computer Science, School of Cognitive and Computing Sciences, University of Sussex, Brighton BN1 9QH, June 1989.

[CH98] I. Castellani and M. Hennessy. Testing theories for asynchronous languages. *Lecture Notes in Computer Science*, 1530:90ff, 1998.

[CJLM97] E.M. Clarke, S. Jha, Y. Lu, and M. Minea. Equivalence checking using abstract BDDs. manuscript, 1997.

[CKL+92] E.M. Clarke, S. Kimura, D.E. Long, S. Michaylov, S.A. Schwab, and J.P. Vidal. Symbolic computation algorithms on shared memory multiprocessors. In N. Suzuki, editor, *Shared Memory Multiprocessing*, pages 53–80. MIT Press, 1992.

[CKS92] R. Cleaveland, M. Klein, and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal μ-calculus. In G.v. Bochmann and D. Probst, editors, *Proceedings of Computer Aided Verification (CAV '92), Montreal, Canada, Juni–Juli 1992*, volume 663 of *Lecture Notes in Computer Science*, pages 410–422, Berlin-Heidelberg-New York-Tokyo, 1992. Springer.

[CKZ93] E.M. Clarke, K. Khaira, and X. Zhao. Word level model checking — a new approach for verifying arithmetic circuits. In *Proc. 30th ACM/IEEE Design Automation Conference (DAC '93)*. IEEE Computer society press, 1993.

[CL85] K.M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. on Computer Systems*, 3(1):63–75, February 1985.

[Cle90] R. Cleaveland. Tableau–based model checking in the propositional μ–calculus. *Acta Informatica*, 27(8):725–747, September 1990.

[CLM89] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proc. 4th Ann. IEEE Symp. on Logic in Computer Science (LICS '89)*, Asilomar, Calif., June 1989.

[CLM91] E.M. Clarke, D.E. Long, and K.L. McMillan. A language for compositional specification and verification of finite state hardware controllers. *Proc. IEEE*, 79(9):1283–1292, September 1991.

[CM84] E.M. Clarke and B. Mishra. Automatic verification of asynchronous circuits. In *Proc. Workshop on Logics of Programs*, volume 164 of *LNCS*, pages 101–115. Springer, 1984.

[CMZ+93] E.M. Clarke, K.L. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large Boolean functions with applications to technology mapping. In *Proc. 30th ACM/IEEE Design Automation Conference (DAC '93)*, pages 54–60. IEEE Computer Society Press, 1993.

[CP96] J.M. Couvreur and D. Poitrenaud. Model checking based on occurrence net graph. In R. Gotzhein and J. Bredereke, editors, *Formel Description Techniques IX: Theory, application and tools, IFIP TC6/6.1 International Conference on Formal Description Techniques IX/ Protocol Specification Testing and Verification XVI, Kaiserslautern, 8.–11. Oktober 1996*, pages 381–395, London-Glasgow-Weinheim-New York-Tokyo-Melbourne-Madras, 1996. Chapman & Hall.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[CS92] R. Cleaveland and B. Steffen. Faster model checking for the modal μ-calculus. In Kim G. Larsen and Arne Skou, editors, *Proceedings of Computer Aided Verification (CAV '91), Aalborg, Dänemark, Juli 1991*, volume 575 of *Lecture Notes in Computer Science*, pages 48–58, Berlin-Heidelberg-New York-Tokyo, 1992. Springer.

[CS93] R. Cleaveland and B. Steffen. A linear–time model–checking algorithm for the alternation–free modal μ–calculus. *Formal Methods in System Design*, 2(2):121–147, April 1993.

[CS01] E.M. Clarke and B-H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Deduction*. Elsevier, 2001.

[CT86] R.H. Carver and K. Tai. Reproducible Testing of Concurrent Programs based on Shared Variables. In *Proc. Int. Conf. on Distributed Computing Systems, Cambridge, Massachusetts*, pages 428–433. IEEE, May 1986.

[CT91] R.H. Carver and K. Tai. Replay and Testing for Concurrent Programs. *IEEE Software*, pages 67–74, March 1991.

[CT95] R.H. Carver and K.C. Tai. Test sequence generation from formal specifications of distributed programs. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95), Vancouver, Canada, 30. Mai–2. Juni 1995*, pages 360–367, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[CVWY92] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[CY95] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4):857–907, July 1995.

[CZ94] E.M. Clarke and X. Zhao. Combining symbolic computation and theorem proving: some problems of Ramanujan. In Alan Bundy, editor, *12th Int. Conf. on Automated Deduction (CADE '94)*, volume 814 of *LNCS*, pages 758–763, Nancy, France, Jun.–Jul. 1994. Springer.

[dA97] L. de Alfaro. Temporal logics for the specification of performance and reliability. In *14th Annual Symposium on Theoretical Aspects of Computer Science (STACS'97), Lübeck, 27. Februar–1. März, 1997*, volume 1200 of *Lecture Notes in Computer Science*, pages 165–176, Berlin-Heidelberg-New York-Tokyo, 1997. Springer.

[Dam94] M. Dam. CTL* and ECTL* as fragments of the modal μ–calculus. *Theoretical Computer Science*, 126:77–96, 1994.

[Dam95a] M. Dam. Compositional proof systems for model checking infinite state processes. In *Proc. 6th Int. Conf. on Concurrency Theory (CONCUR '95)*, volume 962 of *LNCS*, pages 12–26. Springer, 1995.

[Dam95b] D. Dams. *Abstract interpretation and partition refinement for model checking.* PhD thesis, Technical University Eindhoven, 1995.

[DC86] D.L. Dill and E.M. Clarke. Automatic verification of asynchronous circuits using temporal logic. *IEEE Proceedings*, 133(5), September 1986.

[DF95] J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption–commitment style reasoning and theorem proving. In Pierre Wolper, editor, *Proc. 7th Workshop on Computer Aided Verification (CAV '95)*, volume 939 of *LNCS*, pages 45–69. Springer, 1995.

[DGG94] D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems: Abstractions preserving $\forall$CTL*, $\exists$CTL* and CTL*. In Ernst-Rüdiger Olderog, editor, *Programming Concepts, Methods and Calculi (PROCOMET '94)*, IFIP Transactions, pages 561 – 581, Amsterdam, 1994. North Holland / Elsevier.

[DH84] R. DeNicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[Die90] V. Diekert. *Combinatorics on Traces*, volume 454 of *Lecture Notes in Computer Science.* Springer, Berlin-Heidelberg-New York-Tokyo, 1990.

[Dij68] E.W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.

[Dil89a] D. Dill. Timing assumptions and verification of finite–state concurrent systems. In J. Sifakis, editor, *Proc. Int. Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212, Grenoble, France, 1989. Springer.

[Dil89b] D. Dill. *Trace Theory For Automatic Hierarchical Verification Of Speed-independent Circuits.* ACM Distinguished Dissertations. MIT Press, Cambridge MA, 1989.

[dLP98] W. –P. deRoever, H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Difference*, volume 1536 of *LNCS*. Springer, 1998.

[DLW96] A. Dawar, S. Lindell, and S. Weinstein. First order logic, fixed point logic and linear order. In H. Kleine-Büning, editor, *Proc. Computer Science Logic (CSL '95)*, volume 1092 of *LNCS*, pages 161–177. Springer, 1996.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. *Lecture Notes in Computer Science*, 1066:208ff, 1996.

[DP90] A.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks. Cambridge University Press, 1990.

[Ebe95] R. Berks J.C. Ebergen. VERDECT: A verifier for asynchronous circuit design. *Newsletter of the TCCA*, October 1995.

[EC80] E.A. Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proc. 17th Int. Coll. on Automata, Languages and Programming (ICALP '80)*, volume 85 of *LNCS*, pages 169–181. EATCS, Springer, July 1980.

[EC82] E.A. Emerson and E.M. Clarke. Using branching time logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[EFT93] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking. *Distributed Computing*, 6:155–164, 1993.

[EH85] E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1):1–24, February 1985.

[EH86] E.A. Emerson and J.Y. Halpern. "sometimes" and "not never" revisited: on branching time vs. linear time. *Journal of the ACM*, 33:151–178, 1986.

[Ehr61] A. Ehrenfeucht. An application of games to the completeness problem for formalized theories. *Fund. Math.*, 49:129–141, 1961.

[EJS93] E.A. Emerson, C.S. Jutla, and A.P. Sistla. On model–checking for fragments of μ–calculus. In C. Courcoubetis, editor, *Proc. 5th Workshop on Computer Aided Verification (CAV '93)*, volume 697 of *LNCS*. Springer, June 1993.

[EL85] E.A. Emerson and C.L. Lei. Modalities for model checking: Branching time strikes back. In *Proc. 12th Symp. on Principles of Programming Languages (POPL '85)*, New Orleans, La., January 1985.

[EL86] E.A. Emerson and C.L. Lei. Efficient model checking in fragments of the propositional μ–calculus. In *Proc. 1st Symp. on Logic in Computer Science (LICS '86)*, Boston, Mass., June 1986.

[EM97] J. Esparza and S. Melzer. Model checking LTL using constraint programming. Technischer Bericht TUM-I9711, Institut für Informatik, Technische Universität München, München, 1997.

[Eme85] E.A. Emerson. Automata, tableaux, and temporal logic. In R. Parikh, editor, *Proc. Int. Conf. on Logics of Programs*, volume 193 of *LNCS*, pages 79–88. Springer, 1985.

[Eme90] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 997–1072. Elsevier, 1990.

[Eng91] J. Engelfriet. Branching processes of Petri nets. *Acta Informatica*, 28(6):575–591, 1991.

[ER98] S. Edelkamp and F. Reffel. OBDDs in heuristic search. In O. Herzog and A. Günter, editors, *Proc. KI-98: Advances in Artificial Intelligence*, volume 1504 of *LNCS/LNAI*, pages 81–92. Springer, 1998.

[ES84] E.A. Emerson and A.P. Sistla. Deciding full branching time logic. *Information and Control*, 61:175–201, 1984.

[ES93] E.A. Emerson and A.P. Sistla. Symmetry and model checking. In C. Courcoubetis, editor, *Proc. 5th Workshop on Computer Aided Verification (CAV '93)*, volume 697 of *LNCS*, Elounda, Crete, June 1993. Springer.

[Esp94] J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23(2–3):151–195, 1994. Also Universität Hildesheim, Institut für Informatik, Bericht 14/92.

[ETV97] J. Eloranta, M. Tienari, and A. Valmari. Essential transitions to bisimulation equivalences. *Theoretical Computer Science*, 179(1–2):397–419, 1 June 1997.

[Fid89] C. Fidge. Partial orders for parallel debugging. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, volume 24(1), pages 183–194, New York, January 1989. ACM SIGPLAN/SIGOPS, ACM Press.

[Fit83] M. Fitting. *Proof methods for modal and intuitionistic logics*. Reidel, Dordrecht, 1983.

[FL79] M.J. Fischer and R.E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

[FOH93] H. Fuji, G. Ootomo, and C. Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *Proc. Int. Conf. on Computer Aided Design (ICCAD '93)*. IEEE, 1993.

[For94] Formal Systems (Europe), Ltd. *Failures-Divergence-Refinement FDR, User Manual and Tutorial*, 1994.

[Fra54] R. Fraïssé. Sur quelques classifications des systèmes de relations. Séries A 1, Publications Scientifiques de l'Universit'e d'Algerie, 1954.

[Fra86] N. Francez. *Fairness*. Text and Monographs in Computer Science. Springer, 1986.

[Fre96] M. Frey. Debugging parallel programs using temporal logic specifications. In I. Jelly, I. Gorton, and P. Croll, editors, *Software Engineering for parallel and Distributed Systems*, pages 122–133, London, March 1996. IFIP, Chapman & Hall.

[FS98] M. Frey and B.-H Schlingloff. On-the-fly model checking of program runs for automated debugging. In *Proc. IEEE EuroMicro*, Västerås, Sweden, July 1998.

[FW94] M. Frey and A. Weininger. A temporal logic language for debugging parallel programs. In *Proceedings of the 20th EUROMICRO Conference, Liverpool, England*, pages 170–178. Euromicro, IEEE, September 1994.

[FYBV93] E. Felt, G. York, R. Brayton, and A.S. Vincentelli. Dynamic variable reordering for BDD minimization. In *Proc. European Design Automation Conference (EuroDAC '93)*, pages 130–135, September 1993.

[Gab89] D. Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. In B. Banieqbal, editor, *Temporal Logic in Specification*, volume 398 of *LNCS*, pages 431–448. Springer, 1989.

[GBK96] D. Gurov, S. Berezin, and B.M. Kapron. A modal μ–calculus and a proof system for value passing processes. In *Proc. Int. Workshop on Verification of Infinite State Systems (INFINITY '96)*, Electronic Notes in Theoretical Computer Science, pages 149–163, Pisa, August 1996. Report, University of Passau, MIP–9614 July 1996.

[GHP95] P. Godefroid, G.J. Holzmann, and D. Pirottin. State–space caching revisited. *Formal Methods in System Design*, 7(3):1–15, November 1995.

[GHR94] D. Gabbay, I. Hodkinson, and M. Reynolds. *Temporal Logic: Mathematical Foundations and Computational Aspects*, volume 1. Clarendon Press, Oxford, 1994.

[GKP92] U. Goltz, R. Kuiper, and W. Penczek. Propositional temporal logics and equivalences. In W.R. Cleaveland, editor, *3th International Conference on Concurrency Theory CONCUR'92, Stony Brook, NY*, volume 630 of *Lecture Notes in Computer Science*, pages 222–235, Berlin-Heidelberg-New York-Tokyo, 1992. Springer.

[GKPP95] R. Gerth, R. Kuiper, D. Peled, and W. Penczek. A partial order approach to branching time logic model checking. In *Proc. 3rd Israel Symp. on the Theory of Computing and Systems (ISTCS '95)*, pages 130–140. IEEE Computer Society Press, 1995.

[GKPW93] D. Gomm, E. Kindler, B. Paech, and R. Walter. Compositional liveness properties of en-systems. In M.A. Marsan, editor, *Applications and Theory of Petri Nets 1993, 14th Int. Conference, Chicago, Illinois, Juni 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 262–281, Berlin-Heidelberg-New York-Tokyo, 1993. Springer.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16:843–872, May 1994.

[GL96] P. Godefroid and D.E. Long. Symbolic protocol verification with queue BDDs. In *Proc. 11th Ann. IEEE Symp. on Logic in Computer Science (LICS '96)*, New Brunswick, July 1996.

[GL00] S.J. Garland and N. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 13, pages 285–312. Cambridge University Press, New York, NY, 2000.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proc. 2nd Workshop on Computer Aided Verification (CAV '90)*, volume 531 of *LNCS*, pages 176–185, Rutgers, New Brunswick, June 1990. Springer.

[GP93] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proc. 5th Workshop on Computer Aided Verification (CAV '93)*, volume 697 of *LNCS*, pages 438–449, Elounda, Crete, June 1993. Springer.

[GPSS80] D. Gabbay, Amir Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proc. 7th ACM Symp. on Principles of Programming Languages (POPL '80)*, pages 163–173, January 1980.

[GPVW95] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *Protocol Specification Testing and Verification XV, Proceedings of the Fifteenth IFIP WG 6.1 Int. Symposium, Warsaw, Poland, Juni 1995*, pages 3–18, London-Glasgow-Weinheim-New York-Tokyo-Melbourne-Madras, 1995. Chapman & Hall.

[Gra97] B. Grahlmann. The PEP tool. In O. Grumberg, editor, *Computer Aided Verification, 9th Int. Conference (CAV'97), Haifa, Israel, 22.–25. Juni 1997*, volume 1254 of *Lecture Notes in Computer Science*, pages 440–443. Springer-Verlag, 1997.

[GS86] Y. Gurevich and S. Shelah. Fixed–point extensions of first–order logic. *Annals of Pure and Applied Logic*, 32:265–280, 1986.

[GS90a] S. Graf and B. Steffen. Compositional minimization of finite state processes. In R. Kurshan and E.M. Clarke, editors, *Proc. 2nd Workshop on Computer Aided Verification (CAV '90)*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1990. Also in Springer LNCS 531.

[GS90b] C.A. Gunter and D.S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 633–674. Elsevier, 1990.

[GS92] S.M. German and A.P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39:675–735, 1992.

[GW91] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science (LICS '91), Amsterdam, 15.–18. Juli 1991*, pages 406–415, Los Alamitos, CA, July 1991. IEEE Computer Society Press.

[GW92a] V.K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science, 12th Conference, New Delhi, Indien, Dezember 1992*, volume 652 of *Lecture Notes in Computer Science*, pages 253–264, Berlin-Heidelberg-New York-Tokyo, 1992. Springer.

[GW92b] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In Kim G. Larsen and Arne Skou, editors, *Proceedings of Computer Aided Verification (CAV '91), Aalborg, Dänemark, Juli 1991*, volume 575 of *Lecture Notes in Computer Science*, pages 332–342, Berlin-Heidelberg-New York-Tokyo, 1992. Springer.

[GW94] V.K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed systems. *IEEE Trans. on Parallel and Distributed Systems*, 5(3):229–307, March 1994.

[GW96] V.K. Garg and B. Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1323–1333, December 1996.

[GYK90] G.S. Goldszmidt, S. Yemini, and S. Katz. High-level language debugging for concurrent programs. *ACM Transactions on Computer Systems*, 8(4):311–336, November 1990.

[Har84] D. Harel. Dynamic logic. In F. Guenthner D. Gabbay, editor, *Handbook of Philosophical Logic*, chapter II. 10, pages 488–604. Reidel, 1984.

[Hax88] A.E. Haxthausen. Mutually recursive algebraic domain equations. In R. Bloomfield; L. Marshall; R. Jones, editor, *Proceedings of the 2nd VDM-Europe Symposium*, volume 328 of *LNCS*, pages 299–317, Berlin, September 1988. Springer.

[Hax97] A.E. Haxthausen. Order-sorted algebraic specifications with higher-order functions. *Theoretical Computer Science*, 183(2):157–185, September 1997. also in LNCS 936, pp. 133ff, 1995.

[HC77] G.E. Hughes and M.J. Cresswell. *An Introduction to Modal Logic*. Methuen, 1977.

[HC84] G.E. Hughes and M.J. Cresswell. *A Companion to Modal Logic*. Methuen, London-New York, 1984.

[Hen80] M. Hennessy. A proof system for the first-order relational calculus. *Journal of Computer and System Sciences*, 20(1):96–110, February 1980.

[Hen88] M. Hennessy. *Algebraic Theory of Processes*. The MIT Press, Cambridge, Mass., 1988.

[HHK85] P.K. Harter, D.M. Heimbigner, and R. King. Idd: An Interactive Distributed Debugger. In *5th International Conference on Distributed Computing Systems, Denver, Colorado*, pages 498–506, 1985.

[HJ89] H. Hansson and B. Jonsson. A framework for reasoning about time and reliability. In *Proc. 10th IEEE Real-Time Systems Symp.*, pages 102–111, 1989.

[HK90] Z. Har'El and R.P. Kurshan. Software for analytical development of communications protocols. *AT&T Tech. J.*, 69(1):45–59, Jan.-Feb. 1990.

[HLP90] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit Clock Temporal Logic. In *5th annual IEEE Symposium on Logic in Computer Science, Philadelphia, PA, Juni 1990*, pages 402–413, Los Alamitos, CA, 1990. IEEE Computer Society.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, January 1985.

[HMP92] T. Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. Technical Report TR 92–1263, Dept. of CS, Cornell Univ., January 1992.

[HMPS96] G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian analysis of large finite state machines. *IEEE Transactions on Computer–Aided Design of Integrated Circuits and Systems*, 15(12):1479–1493, December 1996.

[Hoa78] C.A.R. Hoare. An axiomatic basis of computer programming. In C.V. Ramamoorthy and Raymond T. Veh, editors, *Tutorial: Software Methodology*, pages 260–265. IEEE Computer Society Press, 1978. Nachdruck von *Communications of the ACM*, 12(10):576–580, Oktober 1969.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hol91] G. Holzmann. *Design and Validation of Computer Protocols*. Software Series. Prentice Hall, 1991.

[Hol95] G. Holzmann. An analysis of bitstate hashing. In *Proc. 15th Int. Conf. on Protocol Specification, Testing and Verification*, pages 301–314, Warsaw, June 1995. INWG/IFIP, Chapman and Hall.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[How86] W.E. Howden. A functional approach to program testing and analysis. *IEEE Transactions on Software Engineering*, 12(10):997–1005, October 1986.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, volume F13 of *NATO ASI Series*, pages 477–498. Springer, 1985.

[HP95] U. Hamer and J. Peleska. Z applied to the A330/340 CICS cabin communication system. In M.G. Hinchey and J.P. Bowen, editors, *Applications of Formal Methods*, pages 253–284. Prentice Hall International Series in Computer Science, 1995.

[HP99] A.E. Haxthausen and J. Peleska. Formal development and verification of a distributed railway control system. In Jeanette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99—Formal Methods, Volume II; Proceedings of the 1st FMERail Workshop*, volume 1709 of *Lecture Notes in Computer Science*, pages 1546–1563, Utrecht, The Netherlands, June 1999. Springer.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[HV00] J. Helovuo and A. Valmari. Checking for CFFD-preorder with tester processes. In *Proc. Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, volume 1785 of *LNCS*, pages 283–298, Berlin, March 2000. Springer.

[ID93] C.W. Ip and D. Dill. Better verification through symmetry. In L. Claesen, editor, *Proc. 11th Int. Symp. on Computer Hardware Description Languages and their Applications*. North–Holland, April 1993.

[IEE94] IEEE. *IEEE Standard for the Futurebus+ Logical Protocol Specification*. IEEE Computer Society, 1994. IEEE Standard 896. 1, 1994 Edition.

[Imm86] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.

[Jan90] P. Jančar. Decidability of a temporal logic problem for Petri nets. *Theoretical Computer Science*, 74(1):71–93, July 1990.

[JM87] R. Jahanian and A. Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEETC: IEEE Transactions on Computers*, 36, 1987.

[Jos93] B. Josko. Modular specification and verification of reactive systems. Habilitationsschrift, University of Oldenburg, April 1993.

[JW96] D. Janin and I. Walukiewicz. On the expressive completeness of the propositional μ–calculus with respect to monadic second order logic. In *Proc. 7th Int. Conf. on Concurrency Theory (CONCUR '96)*, volume 1119 of *LNCS*, Pisa, Italy, 1996. Springer.

[Kam68] H.W. Kamp. *Tense Logic and the Theory of Linear Order*. PhD thesis, Univ. of Calif. , Los Angeles, 1968.

[Kim95] S. Kimura. Residue BDD and its application to the verification of arithmetic circuits. In *Proc. 32nd Int. Design Automation Conference (DAC '95)*, 1995.

[KL86] R. Kannan and R.J. Lipton. Polynomial–time algorithm for the orbit problem. *Journal of the ACM*, 33(4):808–821, 1986.

[KLM+97] R.P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun. Verifying hardware in its software context. In *Proc. Int. Conf. on Computer Aided Design (ICCAD '97)*, San Jose, CA, USA, November 1997. IEEE.

[KM89] R.P. Kurshan and K.L. McMillan. A structural induction theorem for processes. In *Proc. 8th Ann. ACM Symp. on Principles of Distributed Computing*. ACM Press, August 1989.

[Kni95] P. Knirsch. *Kaskadenrekursion in verteilten Systemen*. PhD thesis, Institut für Informatik, Technische Universität München, München, May 1995.

[Koz83] D. Kozen. Results on the propositional μ–calculus. *Theoretical Computer Science*, 27:333–354, December 1983.

[KP83] D. Kozen and R. Parikh. A decision procedure for the propositional μ–calculus. In *Proc. Int. Symp. Logic of Programs*, 1983.

[KP88] S. Katz and D. Peled. An efficient verification method for parallel and distributed programs. In deBakker et al., editor, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*. Springer, 1988.

[KP89] S. Katz and D. Peled. Interleaving set temporal logic. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proceedings of the Conference on Temporal Logic in Specification, Altrincham, UK, 8.–10. April 1987*, volume 398 of *Lecture Notes in Computer Science*, pages 21–43, Berlin-Heidelberg-New York-Tokyo, April 1989. Springer.

[KP90] S. Katz and D. Peled. Interleaving Set Temporal Logic. *Theoretical Computer Science*, 75:263–287, 1990.

[KP95] Y. Kesten and A. Pnueli. A complete axiomatization of QPTL. In *Proc. 10th Ann. IEEE Symp. on Logic in Computer Science (LICS '95)*, pages 2–12, 1995.

[KPP94] M. Kwiatkowska, D. Peled, and W. Penczek. A hierarchy of partial order temporal properties. In D.M. Gabbay and H.J. Ohlbach, editors, *Temporal Logic, Proceedings of the 1st International Conference (ICTL '94), Bonn, Juli 1994*, volume 827 of *Lecture Notes in Computer Science (LNAI)*, pages 398–414, Berlin-Heidelberg-New York-Tokyo, 1994. Springer.

[Kri63] S.A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[Kri75] S.A. Kripke. Outline of a theory of truth. *Journal of Philosophy*, 72:690–716, 1975.

[Krö75] F. Kröger. Formalization of algorithmic reasoning. In Jirí Becvár, editor, *Mathematical Foundations of Computer Science 1975, 4th Symposium*, volume 32 of *lncs*, pages 287–293, Mariánské Lázne, Czechoslovakia, 1–5 September 1975. Springer.

[Krö76] F. Kröger. Logical rules of natural reasoning about programs. In S. Michaelson and R. Milner, editors, *Automata, Languages and Programming, 3rd Int. Colloqium (ICALP'76), Edinburgh*, pages 87–97, Edinburgh, 1976. Edinburgh Press.

[Krö78] F. Kröger. A uniform logical basis for the description, specification and verification of programs. In *Proc. IFIP Work. Conf. Formal Desription of Programming Concepts*, pages 441–457, St. Andrews, Canada, 1978. North Holland.

[Krö80] F. Kröger. Infinite proof rules for loops. *Acta Informatica*, 14(4):371–389, October 1980.

[Krö87] F. Kröger. *Temporal logic of Programs*. Number 8 in EATCS monographs on TCS. Springer, 1987.

[KT90] D. Kozen and J. Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 791–839. Elsevier, 1990.

[Kur89] R.P. Kurshan. Analysis of discrete event coordination. In J.W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proc. REX Workshop on Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness*, volume 430 of *LNCS*. Springer, May 1989.

[Kur94] R.P. Kurshan. *Computer–Aided Verification of Coordinating Processes: The Automata–Theoretic Approach*. Princeton University Press, Princeton, New Jersey, 1994.

[KV96] O. Kupferman and Moshe Y. Vardi. Verification of fair transition systems. In Rajeev Alur and T. Henzinger, editors, *Proc. 8th Workshop on Computer Aided Verification (CAV '96)*, volume 1102 of *LNCS*, pages 372–382. Springer, Jul./Aug. 1996.

[KV97] E. Kindler and T. Vesper. A temporal logic for events and states in Petri nets. In B. Farwer, D. Moldt, and M.-O. Stehr, editors, *Petri-Nets in Systems-Engineering (PNSE'97) Modelling, Verification, and Validation*, Fachberichte FBI-HH-B-205/07. Universität Hamburg, 1997.

[KV00] L.M. Kristensen and A. Valmari. Improved question-guided stubborn set methods for state properties. In *Application and Theory of Petri Nets 2000, 21st International Conference, Aarhus, Denmark, Proceedings, Lecture Notes in Computer Science 1825, Springer-Verlag, pp. 282-302*, pages 397–419, June 2000.

[KW96] D. Kindred and J.M. Wing. Fast, automatic checking of security protocols. In *USENIX 2nd Workshop on Electronic Commerce*, 1996.

[Lam78] L. Lamport. Time, clocks and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–564, 1978.

[Lam80] L. Lamport. "sometimes" is sometimes "not never". In *Proc. 7th Ann. ACM Symp. on Principles of Programming Languages (POPL '80)*, pages 174–185, Las Vegas, January 1980. ACM.

[Lam83] L. Lamport. What good is temporal logic? In *Proc. IFIP*, pages 657–668, 1983.

[Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3), May 1994.

[Lar88] K.G. Larsen. Proof systems for Hennessy-Milner logic with recursion. In M. Dauchet and M. Nivat, editors, *Proceedings 13th Coll. on Trees in Algebra and Programming, CAAP'88, Nancy, France, 21.–24. März 1988*, volume 299 of *Lecture Notes in Computer Science*, pages 215–230, Berlin-Heidelberg-New York-Tokyo, 1988. Springer-Verlag.

[LBC+94] D.E. Long, A. Browne, E.M. Clarke, S. Jha, and W.R. Marrero. An improved algorithm for the evaluation of fixpoint expressions. In *Proc. 6th Workshop on Computer Aided Verification (CAV '94)*, LNCS, pages 338–350. Springer, 1994.

[Lew12] C.I. Lewis. Implication and the algebra of logic. *Mind*, 21:522–531, 1912.

[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*, 6(1):11–44, 1995. also in CAV '92, LNCS 663.

[LMC87] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[Lon93] D.E. Long. *Model Checking, Abstraction and Compositional Verification*. PhD thesis, CMU School of Computer Science, CMU–CS–93–178, July 1993.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th Ann. ACM Symp. on Principles of Programming Languages (POPL '85)*, New Orleans, La., January 1985. ACM press.

[LPS81] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, justice, and fairness: The ethics of concurrent termination. In *Proc. Int. Conf. on Automata, Languages, and Programming (ICALP '81)*, volume 115 of *LNCS*. Springer, 1981.

[LPY97] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1(1/2), October 1997.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Proc. Int. Conf. Logics of Programs*, volume 193 of *LNCS*, pages 196–218. Springer, 1985.

[LRT89] K. Lodaya, R. Ramanujam, and P.S. Thiagarajan. A logic for distributed transition systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Nordwijkerhout, Niederlande, 30. Mai–3. Juni 1988*, volume 354 of *Lecture Notes in Computer Science*, pages 508–522, Berlin-Heidelberg-New York-Tokyo, 1989. Springer.

[LRT92] K. Lodaya, R. Ramanujam, and P.S. Thiagarajan. Temporal logics for communicating sequential agents: I. *International Journal of Foundations of Computer Science*, 3(2):117–159, 1992.

[LRT93] K. Lodaya, R. Ramanujam, and P.S. Thiagarajan. Decidability of a partial order based temporal logic. In A. Lingas, R. Karlsson, and S. Carlsson, editors, *Automata, Languages and Programming, 20th Int. Colloquium, ICALP'93, Lund, Schweden, 5.–9. Juli 1993*, volume 700 of *Lecture Notes in Computer Science*, pages 582–592, Berlin-Heidelberg-New York-Tokyo, 1993. Springer.

[LT87] K. Lodaya and P.S. Thiagarajan. A modal logic for a subclass of event structures. In Th. Ottmann, editor, *Proceedings of the 14th International Colloquium on Automata, Languages, and Programming (ICALP'87), Karlsruhe*, volume 267 of *Lecture Notes in Computer Science*, pages 290–303, Berlin-Heidelberg-New York-Tokyo, 1987. Springer.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[Lyn88] N.A. Lynch. I/O automata: A model for discrete event systems. In *Proc. of 22nd Conf. on Information Sciences and Systems*, pages 29–38, Princeton, NJ, USA, March 1988.

[Mad92] A. Mader. Tableau recycling. In *Proc. 4th Workshop on Computer Aided Verification (CAV '92)*, LNCS. Springer, 1992.

[Maz77] A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus, 1977.

[Maz87] A. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II, Bad Honnef*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324, Berlin-Heidelberg-New York-Tokyo, 1987. Springer.

[MB59] D. Muller and W. Bartke. A theory of asynchronous circuits. *Theory of Switching*, 1959.

[MC85] B. Mishra and E.M. Clarke. Hierarchical verification of asynchonous circuits using temporal logic. *Theoretical computer science*, 38:269–291, 1985.

[McM92] K. McMillan. Using unfoldings to avoid the state space explosion problem in the verification of asynchronous circuits. In *Proc. 4th Workshop on Computer Aided Verification*, volume 663 of *LNCS*, pages 164–177, Montreal, Canada, June 1992. Springer.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[McM95a] K. McMillan. A technique of a state space search based on unfolding. *Formal Methods in System Design*, 6(1), 1995.

[McM95b] K.L. McMillan. Trace theoretic verification of asynchronous circuits using unfoldings. *Lecture Notes in Computer Science*, 939:180ff, 1995.

[Mea94] C. Meadows. The NRL protocol analyzer: An overview. In *Proc. 2nd Int. Conf. on the Practical Applications of Prolog*, 1994.

[Mer97] S. Merz. Abstraction as a proof rule. Technical report, Universität München, 1997.

[MF76] P. Merlin and D.J. Faber. Recoverability of communication protocols. *IEEE Transactions on Communication*, 24(9):1036–1043, September 1976.

[MG91] R. Marelly and O. Grumberg. Gormel — grammar oriented model checker. Technical Report 697, The Technion, 1991.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[Mil85] R. Milner. Lectures on a calculus for communicating systems. In A.W. Roscoe S.D. Brookes and G. Winskel, editors, *Seminar on Concurrency, Proceedings, Pittsburgh, PA, Juli 1984*, volume 197 of *Lecture Notes in Computer Science*, pages 197–220, Berlin-Heidelberg-New York-Tokyo, 1985. Springer.

[Min99] M. Minea. Partial order reduction for model checking of timed automata. In *Proc. Concur 99*, LNCS. Springer, 1999.

[MMS97] J.C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using MURφ. In *Proc. 1997 IEEE Symp. on Security and Privacy*. IEEE Computer Society Press, 1997.

[Moo94] J.S. Moore. Introduction to OBDD algorithm for the ATP community. *Journal of automated reasoning*, 12:33–45, 1994.

[MP81] Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R.S. Boyer and J.S. Moore, editors, *The Correctness Problem in Computer Science*, Int. Lecture Series in Computer Science, pages 215–273. Academic Press, London, 1981.

[MP82] Z. Manna and A. Pnueli. Verification of concurrent programs: Temporal proof principles. In Dexter Kozen, editor, *Proc. Workshop on Logics of Programs*, volume 131 of *LNCS*, pages 200–252. Springer, 1982.

[MP87] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Proc. 6th Ann. ACM Symp. on Principles of Distributed Computing*, Stanford, CA 94305, October 1987. Stanford University Press.

[MP89] Z. Manna and A. Pnueli. The anchored version of the temporal framework. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 201–284. Springer, 1989.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems – Specification*. Springer, 1992.

[MP93] Z. Manna and A. Pnueli. Models for reactivity. *Acta Informatica*, 30(7):609–678, 1993.

[MP95] Z. Manna and A. Pnueli. *Temporal Verifications of Reactive Systems – Safety*. Springer, 1995.

[MS92] K.L. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In N. Suzuki, editor, *Proc. Int. Symp. on Shared Memory Multiprocessing*, pages 111–134. MIT Press, 1992.

[Mye79] G.J. Myers. *The Art of Software Testing*. Wiley, New York, 1979.

[NA90] J.R. Nowicki and L.J. Adam. *Digital Circuits*. Edward Arnold, 1990.

[Niw88] D. Niwinsky. Fixed points vs. infinite generation. In *Proc. 3rd Ann. IEEE Symp. on Logic in Computer Science (LICS '88)*, pages 402–409, 1988.

[NPW81] M. Nielsen, G. Plotkin, and G. Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13(1):85–108, 1981.

[NRT92] M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. Elementary transition systems. *Theoretical Computer Science*, 96(1):3–33, 1992.

[NRT95] M. Nielsen, G. Rozenberg, and P.S. Thiagarajan. Transition systems, event structures, and unfoldings. *Information and Computation*, 118(2):191–207, 1995.

[Par74] D.M. Park. Finiteness is mu–ineffable. Theory of Computation Report 3, University of Warwick, 1974.

[Par81] D.M. Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science: 5th GI–Conference, Karlsruhe*, volume 104 of *LNCS*, pages 167–183. Springer, March 1981.

[Pel91] J. Peleska. Design and verification of fault tolerant systems with CSP. *Distributed Computing*, 5(2):95–106, 1991.

[Pel93] D. Peled. All from one, one for all: on model checking using representatives. In C. Courcoubetis, editor, *Proc. 5th Workshop Computer Aided Verification (CAV '93)*, volume 697, Elounda, Crete, June 1993. Springer.

[Pel94] D. Peled. Combining partial order reductions with on-the-fly model-checking. In D.L. Dill, editor, *6th International Conference on Computer Aided Verification (CAV'94), Stanford, CA, 21.–23. Juni 1994*, volume 818 of *Lecture Notes in Computer Science*, page 377ff, Berlin-Heidelberg-New York-Tokyo, 1994. Springer-Verlag.

[Pel96a] J. Peleska. Formal methods and the development of dependable systems. Technical Report Report No. 9612, Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik, 1996.

[Pel96b] J. Peleska. Test automation for safety-critical systems: Industrial application and future developments. *Lecture Notes in Computer Science*, 1051:39–59, 1996.

[Pen90] W. Penczek. A concurrent branching time temporal logic. In E. Börger, H. Kleine Bünning, and M.M. Richter, editors, *Proceedings of the 3rd Workshop on Computer Science Logic (CSL'89), Kaiserslautern*, volume 440 of *Lecture Notes in Computer Science*, pages 337–354, Berlin-Heidelberg-New York-Tokyo, 1990. Springer.

[Pen93] W. Penczek. Temporal logics for trace systems: On automated verification. *International Journal of Foundations of Computer Science*, 4(1):31–67, 1993.

[Pet62] C.A. Petri. Kommunikation mit Automaten. Schriften des IIM Nr. 2, Institut für Instrumentelle Mathematik, Bonn, 1962.

[Pet83] G.L. Peterson. A new solution to Lamport's concurrent programming problem using small shared variables. *ACM Trans. Program. Lang. and Syst.*, 5(1):56–65, 1983.

[PHPd98] C. Petersohn, C. Huizing, J. Peleska, and Willem-Paul de Roever. Formal semantics for Ward and Mellor's transformation schemas and its application to fault tolerant systems. *International Journal of Computer Systems Science and Engineering*, 13(2):131–136, March 1998.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th Ann. IEEE Symp. on Found. of Comp. Science (FOCS '77)*, pages 46–57, 1977.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[Pnu84] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI series*. Springer, 1984.

[Pnu86] A. Pnueli. Application of temporal logic to the specification of reactive systems: A survey of current trends. In J.W. Baker, W.-P. de Roever, and G. Rozenberg, editors, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584, Berlin-Heidelberg-New York-Tokyo, 1986. Springer.

[PP90] D. Peled and A. Pnueli. Proving partial order liveness properties. In M.S. Paterson, editor, *Automata, Languages and Programming, 17th Int. Colloquium (ICALP '90), Warwick, England, 16.–20. Juli 1990*, volume 443 of *Lecture Notes in Computer Science*, pages 553–571, Berlin-Heidelberg-New York-Tokyo, 1990. Springer-Verlag.

[Pra76] V. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proc. 17th IEEE Symp. on Foundations of Comp. Sci. (FOCS'76)*, pages 109–121, 1976.

[Pra81] V. Pratt. A decidable μ–calculus. In *Proc. Ann. ACM Symp. on Foundations of Computer Science (FOCS '81)*, 1981.

[Pri57] A.N. Prior. *Time and Modality*. Oxford University Press, 1957.

[Pri67] A. Prior. *Past, Present and Future*. Clarendon Press, Oxford, 1967.

[PS96] J. Peleska and M. Siegel. From testing theory to test driver implementation. *Lecture Notes in Computer Science*, 1051:538ff, 1996.

[PS97] J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Jounal*, 19:53–77, February 1997.

[PT87] R. Paige and R. Tarjan. Three efficient algorithms based on partition refinement. *SIAM journal on computing*, 16(6), December 1987.

[PW84] S.S. Pinter and P. Wolper. A temporal logic for reasoning about partially ordered computations. In *Proceedings of the 3rd annual ACM Symposium on Principles of Distributed Systems (PODC '84), Vancouver, British Columbia*, pages 28–37, New York, 1984. ACM.

[PW97] D. Peled and T. Wilke. Stutter–invariant temporal properties are expressible without the nexttime operator. *Information Processing Letters*, 1997.

[PZ99] J. Peleska and C. Zahlten. Test automation for avionic systems and space technology (extended abstract). In *Softwaretechnik-Trends, Proceedings of the GI Working Group Test, Analysis and Verification of Software*, February 1999.

[QS81] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int. Symp. on Programming*, 1981.

[QS82a] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium in Programming, 5th Colloquium, Turin, Italien, 6.–8. April 1982*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, Berlin-Heidelberg-New York-Tokyo, 1982. Springer-Verlag.

[QS82b] J.P. Quielle and J. Sifakis. Fairness and related properties in transition systems. Technical Report 292, IMAG, March 1982.

[Rab69] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Trans. AMS*, 141:1–35, 1969.

[Ram96] R. Ramanujam. Locally linear time temporal logic. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science (LICS'96), New Brunswick, New Jersey, 27.–30. Juli 1996*, pages 118–127, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[RB86] J.-L Roux and B. Berthomieue. Verification of a local area network protocol with TINA, a software package for time Petri nets. In *Proc. 7th European Workshop on Application and Theory of Petri Nets*, LNCS, pages 183–205. Springer, 1986.

[RCP95] O. Roig, J. Cortadella, and E. Pastor. Verification of asynchronous circuits by BDD-based model checking of Petri nets. *Lecture Notes in Computer Science*, 935:374ff, 1995.

[Rei86] W. Reisig. *Petrinetze*. Springer, Berlin-Heidelberg-New York-Tokyo, 1986.

[Rei88] W. Reisig. Temporal logic and causality in concurrent systems. In F.H. Vogt, editor, *Proceedings of the International Conference on Concurrency (Concurrency 88), Hamburg, 18.–19. Oktober 1988*, volume 335 of *Lecture Notes in Computer Science*, pages 121–139, Berlin-Heidelberg-New York-Tokyo, 1988. Springer.

[Rei89] W. Reisig. Towards a temporal logic of causality and choice in distributed systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of the School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Noordwijkerhout, Niederlande, 30. Mai–3. Juni 1988*, volume 354 of *Lecture Notes in Computer Science*, pages 603–527, Berlin-Heidelberg-New York-Tokyo, 1989. Springer.

[Rei91] W. Reisig. Concurrent temporal logic. SFB-Bericht 324/7/91 B, Institut für Informatik, Technische Universität München, München, 1991.

[Rei98] W. Reisig. *Elements of Distributed Algorithms*. Springer, 1998.

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency*. International Series in Computer Science. Prentice Hall, 1998.

[RU71] N. Rescher and A. Urquhart. *Temporal Logic*. Springer, 1971.

[Rud93] R.L. Rudell. Dynamic variable reordering for ordered binary decision diagrams. In *Proc. IEEE/ACM Int. Conf. on Computer Aided Design (ICCAD '93)*, pages 42–47, November 1993.

[Saf88] S. Safra. On the complexity of ω–automata. In *Proc. 29th IEEE Symp. on Foundations of Computer Science (FOCS '88)*, White Plains, October 1988.

[Sal70] A. Salwicki. Formalized algorithmic languages. *Bull. Acad. Polon. Sci. , Ser. Sci. Math. Astron. Phys.*, 18:227–232, 1970.

[SB97] R.H. Sloan and U. Buy. Stubborn sets for real–time Petri nets. *Formal Methods in System Design*, 11(1):23–40, July 1997.

[SC86] A.P. Sistla and E.M. Clarke. Complexity of propositional temporal logics. *Journal of the ACM*, 32(3):733–749, July 1986.

[Sch90a] B.-H. Schlingloff. Modal definability of ω-tree languages. In *Proceedings of the ESPRIT BRA ASMICS Workshop on Logics and Recognizable Sets*, Dersau, 1990.

[Sch90b] B.-H. Schlingloff. *Zur temporalen Logik von Bäumen*. Bericht TUM-I9012, Institut für Informatik, Technische Universität München, München, March 1990.

[Sch92a] B.-H. Schlingloff. Expressive completeness of temporal logic of trees. *Journal of Applied Non–Classical Logics*, 2. 2:157–180, 1992.

[Sch92b] B.-H. Schlingloff. On the expressive power of modal logic on trees. In A. Nerode and M. Taitslin, editors, *Proc. 2nd Int. Symp. Logical Foundations of Computer Science ("Logic at Tver")*, volume 620 of *LNCS*, pages 441–451. Springer, July 1992.

[Sch97] B.-H. Schlingloff. Verification of finite–state systems with temporal logic model checking. *South African Computer Journal*, 19:27–52, February 1997.

[Sch98] B.-H Schlingloff. Modelling message buffers with binary decision diagrams. In Ali Jaoua, Peter Kempf, and Gunther Schmidt, editors, *Using Relational Methods in Computer Science*, Technical Report Nr. 1998-03, pages 59–70. Fakultät für Informatik, Universität der Bundeswehr München, July 1998.

[Sch00] S. Schneider. *Concurrent and Real-Time Systems: The CSP Approach*. Worldwide Series in Computer Science. John Wiley and Sons, LTD., 2000.

[Seg68] K. Segerberg. Decidability of S4.1. *Theoria*, 34:7–20, 1968.

[Seg71] K. Segerberg. An essay in classical modal logic. Technical Report Filosofiska Studier 13, Department of Philosophy, University of Uppsala, 1971.

[Seg82] K. Segerberg. A completeness theorem in the modal logic of programs. *Universal Algebra and Applications*, Banach Center Publications, Vol. 9, 1982.

[SH97] B.-H. Schlingloff and W. Heinle. Relational algebra and modal logics. In C. Brink, W. Kahl, and G. Schmidt, editors, *Relational Methods in Computer Science*, Advances in Computing Science, chapter 5, pages 70–89. Springer, 1997.

[Sis83] A.P. Sistla. *Theoretical Issues in the Design and Verification of Distributed Systems*. PhD thesis, CMU Dept. of Computer Science, CMU–CS–83–146, August 1983.

[SMH99] B.-H Schlingloff, O. Meyer, and Th. Hülsing. Correctness analysis of an embedded controller. In *Proc. Int. Conf. on Data Systems in Aerospace (Dasia '99)*, Lissabon, 1999.

[SS90] G. Stålmarck and M. Säflund. Modelling and verifying systems and software in propositional logic. In B.K. Daniels, editor, *Proc. Int. Conf. on Safety of Computer Control Systems (SAFECOMP '90)*, pages 31–36. Pergamon Press, 1990.

[Stå89] G. Stålmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. Swedish Patent No. 467076 (1992), US Patent No. 5 276 897 (1994), European Patent No. 0404 454 (1995), 1989.

[Sta90] P.H. Starke. *Analyse von Petri-Netz-Modellen*. B.G. Teubner, Stuttgart, 1990.

[Sta91] P.H. Starke. Reachability analysis of Petri nets using symmetries. *Syst. Anal. Model. Simul.*, 8(4/5):293–303, 1991.

[Sti87] C. Stirling. Modal logics for communicating systems. *Theoretical Computer Science*, 49:311–348, July 1987.

[Sti91] C. Stirling. Modal and temporal logics. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, 1991.

[Str81] R.S. Streett. Propositional dynamic logic of looping and converse. In *Conference Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computation, Milwaukee, WI, 11.–13. Mai 1981*, pages 375–383, New York, 1981. ACM.

[Str82] R.S. Streett. Propositional dynamic logic of looping and converse is elementarily decidable. *Information and Control*, 54(1/2):121–141, July/August 1982.

[SVW87] A.P. Sistla, Moshe Y. Vardi, and Pierre Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[SW91] C. Stirling and D.J. Walker. Local model checking in the modal μ–calculus. *Theoretical Computer Science*, 89(1):161–177, October 1991. also in Proc. TAPSOFT '89, Springer LNCS 351, 369–386, 1989.

[TA87] K.C. Tai and S. Ahuja. Reproducible testing of communication software. In *Computer Software & Applications Conference (CompSAC'87), Tokio, Japan*, pages 317–339. IEEE, October 1987.

[Tai85] K.C. Tai. On testing concurrent programs. In *Computer Software & Applications Conference (COMPSAC'85), Chicago, Illinois*, pages 310–317. IEEE, October 1985.

[Tar55] A. Tarski. A lattice–theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.

[Tar72] R.E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal of Computing*, 1:146–160, 1972.

[TBK91] H.J. Touati, R.K. Brayton, and R.P. Kurshan. Testing language containment for ω–automata using BDDs. In *Proc. 1991 Int. Workshop on Formal Methods in VLSI Design*, January 1991.

[TC96] K.C. Tai and R.H. Carver. Testing of distributed programs. In A.Y. Zomaya, editor, *Parallel and Distributed Computing Handbook*, pages 955–978. McGraw-Hill, New York, u.a., 1996.

[Thi94] P.S. Thiagarajan. A trace based extension of linear time temporal logic. In *Proceedingsof the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS'94), Paris, Frankreich, 4.–7. July 1994*, pages 438–447, Los Alamitos, CA, 1994. IEEE Computer Society Press.

[Thi95] P S. Thiagarajan. A trace consistent subset of PTL. In I. Lee and S.A. Smolka, editors, *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95), Philadelphia, PA, August 1995*, volume 962 of *Lecture Notes in Computer Science*, pages 438–452, Berlin-Heidelberg-New York-Tokyo, 1995. Springer.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. Elsevier, 1990.

[Tho99] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Language Theory*, volume III, pages 389–455. Springer, 1999.

[TK86] R.N. Taylor and C.D. Kelly. Structural testing of concurrent programs. In *Workshop on Software Testing, Banff, Canada*, pages 164–170. IEEE, July 1986.

[TKI+97] A. Takamura, M. Kuwako, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, and T. Nanya. TITAC-2: An asynchronous 32-bit microprocessor based on scalable-delay-insensitive model and its implementation. *ICCD*, pages 288–294, October 1997.

[TO86] K.C. Tai and E.E. Obaid. Reproducible testing of ADA tasking programs. In *Conference on* ADA *Applications and Environments, Miami Beach, Florida*, pages 69–79. IEEE, April 1986.

[TSS98] L. Twele, B.-H. Schlingloff, and H. Szczerbicka. Performability analysis of an avionics-interface. In *IEEE Symp. on Man, Machine and Cybernetics; San Diego*, pages 499–504, 1998.

[TW97] P.S. Thiagarajan and I. Walukiewicz. An expressively complete linear time temporal logic for Mazurkiewicz traces. In *Proc. 12th Ann. IEEE Symp. on Logic in Computer Science (LICS '97)*, pages 183–194, 1997.

[UKP98] G. Urban, H.-J. Kolinowitz, and J. Peleska. A survivable avionics system for space applications. In *Proc. FTCS-28, 28th Annual Symposium on Fault-Tolerant Computing*, pages 372–381, Munich, June 1998.

[Val90] A. Valmari. A stubborn attack on state explosion. In E.M. Clarke and R.P. Kurshan, editors, *Proc. 2nd Workshop on Computer Aided Verification (CAV '90)*, volume 531 of *LNCS*, pages 156–165, Rutgers, New Brunswick, June 1990. Springer.

[Val91] A. Valmari. Stubborn sets for reduced state space generation. In G. Rozenberg, editor, *Advances in Petri Nets 1990, 10th Int. Conf on Applications and Theory of Petri Nets, Bonn, 28.–30. Juni 1989*, volume 483 of *Lecture Notes in Computer Science*, pages 491–517, Berlin-Heidelberg-New York-Tokyo, 1991. Springer.

[Var82] M.Y. Vardi. The complexity of relational query languages. In *Proc. 14th Int. ACM Symp. on the Theory of Computing*, pages 137–146, 1982.

[Var88] M.Y. Vardi. A temporal fixpoint calculus. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 15th Annual ACM Symposium on Principels of Programming Languages (POPL '88), San Diego, CA, USA, Januar 1988*, pages 250–259, New York, 1988. ACM Press.

[Var95] M. Vardi. Alternating automata and program verification. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*. Springer, 1995.

[vB83] J. van Benthem. *Modal Logic and Classical Logic*. Bibliopolis, Naples, 1983.

[vB84] J. van Benthem. Correspondence theory. In F. Guenthner D. Gabbay, editor, *Handbook of Philosophical Logic*, chapter II. 4, pages 167–249. Reidel, 1984.

[vB91] J. van Benthem. *The Logic of Time*. Kluwer, Dordrecht, 2nd edition, 1991.

[VW86] M.Y. Vardi and P. Wolper. An automata–theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science (LICS '86)*, Boston, Mass., June 1986.

[Wal95] I. Walukiewicz. Completeness of Kozen's axiomatisation of the propositional μ–calculus. In *Proc. 10th Ann. IEEE Symp. on Logic in Computer Science (LICS '95)*, pages 14–24, 1995.

[WG93] P. Wolper and P. Godefroid. Partial-order methods for temporal verification. In E. Best, editor, *CONCUR'93, 4th International Conference on Concurrency Theory, Hildesheim, 23.–26. August 1993*, volume 715, pages 233–246, Berlin-Heidelberg-New York-Tokyo, 1993. Springer-Verlag.

[Win91] G. Winskel. A note on model checking the modal ν–calculus. *Theoretical Computer Science*, 83:157–167, 1991. also in Proc. ICALP '89, Ausiello et. al. (ed.), LNCS 372, 761–772.

[Win94] J. Winkowski. Algebras of Processes of Timed Petri Nets. In B. Jonsson and J. Farrow, editors, *CONCUR'94: Concurrency Theory, 5th Int. Conf., Uppsala, Sweden*, volume 836 of *Lecture Notes in Computer Science*, pages 194–209, Berlin-Heidelberg-New York-Tokyo, August 1994. Springer.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *Proc. Int. Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*. Springer, 1989.

[WL93] T.Y. C. Woo and S.S. Lam. A semantic model for authentication protocols. In *Proc. IEEE Symp. on Research in Security and Privacy*, 1993.

[Wol82] P. Wolper. Specification and synthesis of communicating processes using an extended temporal logic. In *Proc. 9th Int. Symp. on Principles of Programming Languages (POPL '82)*, pages 20–33, Albuquerque, January 1982.

[Wol83] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1–2):72–99, 1983.

[Wol85] P. Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, 110–111:119–136, 1985.

[Wol86] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. on Principles of Programming Languages (POPL '86)*, January 1986.

[Wol89] P. Wolper. On the relation of programs and computations to models of temporal logic. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification, Altrincham, UK, 8.–10. April 1987*, pages 75–123. Springer-Verlag, 1989.

[YNT89] T. Yoneda, K. Nakade, and Y. Tohma. A fast timing verification method based on the independence of units. In *Proc. of 19th Int. Symp. on Fault-tolerant Computing*, pages 134–141, 1989.

[Yon99] T. Yoneda. Abstracted instruction cache of TITAC2 - as a benchmark circuit for timed asynchronous circuit verification. Technical report, Tokyo Institute of Technology, June 1999.

[Yov97] S. Yovine. KRONOS: A verification tool for real–time systems. *Software Tools for Technology Transfer*, 1(1/2), October 1997.

[Yov98] S. Yovine. Model–checking timed automata. In G. Rozenberg and F. Vandraager, editors, *Embedded Systems*, LNCS. Springer, 1998.

[YR99] T. Yoneda and H. Ryu. Timed trace theoretic verification using partial order reduction. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 108–121, April 1999.

[YS92] T. Yoneda and B.-H. Schlingloff. On model checking for Petri nets and a linear-time temporal logic. Technical report, IEICE Technical Report, Vol. FTS92 No.1, Tokyo, 1992.

[YS97] T. Yoneda and B.-H. Schlingloff. Efficient verification of parallel real–time systems. *Formal Methods in System Design*, 11(2):187–215, August 1997.

[YZS99] T. Yoneda, B. Zhou, and B.-H. Schlingloff. Verification of bounded delay asynchronous circuits with timed traces. *Lecture Notes in Computer Science*, 1548:59–73, January 1999.

[Zha97] H. Zhang. SATO: An efficient propositional prover. In *Proc. Int. Conf. on Automated Deduction (CADE '97)*, volume 1249 of *LNCS/LNAI*, pages 272–275. Springer, 1997.

[ZYS01] B. Zhou, T. Yoneda, and B.-H. Schlingloff. Conformance and mirroring for timed asynchronous circuits. In *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC 2001)*, January 2001.