Partial State Space Analysis
of Safety-Critical Systems

Bernd-Holger Schlingloff
Contents

0 Introduction ........................................... 7

I Modelling and Specification ........................... 11
  1 Modelling of Reactive Systems .................. 13
      1.1 Parallel Programming Paradigms ............ 14
          1.1.1 Modelling of Communication ............ 14
          1.1.2 Modelling of Execution ............... 16
      1.2 Some Basic Formalisms ..................... 17
          1.2.1 Transition Systems and Automata ........ 17
          1.2.2 ω-Regular Expressions and Process Algebras 23
          1.2.3 Nets, Programs, I/O systems .......... 29

2 Logical Specification Languages .................. 33
  2.1 Propositional and First Order Logic ........... 34
  2.2 Multimodal and Temporal Logic ................ 36
  2.3 Linear and Branching Time Logics ............. 42
  2.4 Propositionally Quantified Logics ............ 46
  2.5 Automata and Logics .......................... 53
  2.6 Relational μ-calculus ........................ 56

3 Model Transformations .............................. 63
  3.1 Models, Automata and Transition Systems ...... 63
  3.2 Safety and Liveness Properties ............... 65
  3.3 Simulation Relations .......................... 69
  3.4 Bisimulations (p-morphisms) .................. 76
  3.5 Bisimulation Minimization ..................... 79
  3.6 Conformance and Mirroring .................... 81

3
II Verification

4 Completeness and Decidability 87
   4.1 Completeness ........................................... 87
   4.1.1 Deductions in Multimodal Logic .................... 89
   4.1.2 Completeness of Multimodal Logic ................ 91
   4.1.3 Completeness of Temporal Logics ................ 92
   4.1.4 Completeness of the μ-calculus ................ 96
   4.2 Decidability ........................................... 97
   4.2.1 Modal Decision Algorithms .......................... 99
   4.2.2 Modal Tableau Rules ................................ 99
   4.2.3 Adequacy of the Modal Tableau Procedure ........ 100
   4.2.4 Global Modal Satisfiability ....................... 101
   4.2.5 Decidability for Branching Time ................... 102
   4.2.6 Tableaus for LTL ................................. 104
   4.3 Incompleteness Results ............................ 106

5 Model Checking Examples 109
   5.1 A Combinatorial Game ................................... 112
   5.2 A Sequential Circuit .................................. 114
   5.3 A Communication Protocol ............................ 116

6 Model Checking Algorithms 121
   6.1 Global Branching Time Model Checking ............... 122
   6.2 Local Linear Time Model Checking ................... 125
   6.3 Model Checking for μ-calculus ...................... 130
   6.4 Binary Decision Diagrams ............................ 135
   6.5 Symbolic Model Diagrams ............................. 146

III Real Time

7 Formalisms for Real-Time 153
   7.1 Real-Time and Hybrid Systems ....................... 153
   7.2 Timed Automata and Time Petri Nets ................ 155
   7.3 Time Net Logic ..................................... 159

8 State Space Techniques for Real-Time 163
   8.1 Model Checking for Time Nets ...................... 163
   8.2 Stubborn Analysis .................................... 166
CONTENTS

9 Verification with Timed Traces 173
  9.1 Timed Trace Theory 174
  9.2 Analysis of Time Petri Nets 181
  9.3 Experimental Results 186

10 Real-Time Conformance 189
  10.1 A Notion of Correctness 189
  10.2 Checking Safety Properties 190
  10.3 Timing Verification 193

IV Debugging and Testing 203

11 Model Checking of Program Runs 205
  11.1 Debugging by Model Checking 207
    11.1.1 State Action Nets 208
    11.1.2 A Temporal Logic for Debugging 210
    11.1.3 Model Checking Debugging Logic 212
  11.2 Safety Requirement 213
  11.3 On-the-Fly Model Checking 215

12 Testing Reactive Real-Time Systems 223
  12.1 Description of the Systems 224
    12.1.1 The ABRIXAS–PTC 224
    12.1.2 The UMTS RLC layer 227
  12.2 Testing Setup 228
  12.3 Formalization of the Requirements 229
  12.4 Results of Testing 237
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, aerospace 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 constituting 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 συστήμα “systēma”, 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 $10^{16}$ 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 cannot 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-
CHAPTER 1. MODELLING OF REACTIVE SYSTEMS

ber of subcomponents. There are two fundamental ways of composing sub-
systems: sequentially (i.e., in time) or parallel (i.e., in space). Whereas
sequential composition is well-understood in computer science and an es-
sential 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 no-
toriously error-prone and hard to understand. As a consequence, many
current programming languages do not allow or support user-defined paral-
lelism. 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 mod-
  ules; sequentialization is not the primary issue in the design;

- certain subcomponents (e.g. hardware components) actually are inde-
  pendent 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.
1.1. PARALLEL PROGRAMMING PARADIGMS

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-
16
CHAPTER 1. MODELLING OF REACTIVE SYSTEMS

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 $\Sigma \triangleq 2^P$ is the powerset of some finite basic set of propositions. An $\omega$-word over $\Sigma$ is a nonempty (finite or infinite) sequence of letters from $\Sigma$. An $\omega$-language is any set of $\omega$-words over $\Sigma$. Note
that for each nontrivial alphabet (i.e., with more than one letter) there are uncountably many \( \omega \)-words and languages.

A transition system \( T \) is a tuple \( T \triangleq (\Sigma, S, \Delta, S_0) \), where

- \( \Sigma \) 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 \( \Delta \)) is finite.

Let \( \sigma \triangleq (\sigma_1 \sigma_2 \sigma_3 \ldots) \) be an \( \omega \)-word over \( \Sigma \). We say that \( \sigma \) is generated by the transition system \( T \), if there exists a sequence \( s_0, s_1, s_2, \ldots \) 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 \( \omega \)-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 \( \sigma = (\sigma_1, \sigma_2, \sigma_3, \ldots) \) is an \( \omega \)-word, then each finite nonempty subsequence \( \sigma^{-i} \triangleq (\sigma_1, \ldots, \sigma_i) \) is called a prefix of \( \sigma \). It follows immediately from the definition that for each \( \sigma \) which is generated by the transition system \( T \) all prefixes \( \sigma^{-i} \) of \( \sigma \) are also generated by \( T \). In other words,

**Lemma 1.1** Transition systems generate prefix closed \( \omega \)-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 \( \sigma \) is weakly fair generated by the transition system \( T \), if in addition to the two requirements above the following holds:

- if \( \sigma \) is finite with last letter \( \sigma_n \), then \( s_n \) is terminal.
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 $\omega$-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 $\omega$-automaton or fair transition system for the alphabet $\Sigma$. 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 $\omega$-automaton, that is, a fair transition system where the set $S$ of states is finite. A (finite or infinite) nonempty word $\sigma \triangleq (\sigma_0, \sigma_1, \ldots)$ is accepted by an automaton $(S, \Delta, S_0, S_{acc}, S_{rec})$, if there is a function $\rho$ assigning to any letter $\sigma_i$ a state $\rho(\sigma_i) \in S$ of the automaton such that

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

- if $\sigma$ is finite with last letter $w_n$, then $(\rho(\sigma_n), \sigma_n, s) \in \Delta$ for some $s \in S_{\text{acc}}$, and

- if $\sigma$ is infinite, then $\inf(\rho) \cap S_{\text{rec}} \neq \emptyset$, where $\inf(\rho)$ is the set of states that appear infinitely often in the range of $\rho$.

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 $S_{\text{acc}} = S_{\text{rec}} = S$, and a weakly fair transition system is one where $S_{\text{rec}} = S$ and $S_{\text{acc}} = \{ s \mid \forall a, s'(s, a, s') \notin \Delta \}$. That is, in a weakly fair transition system all states are recurring, and states are accepting if they are terminal. A classical finite automaton has $S_{\text{rec}} = \emptyset$; it does not accept any infinite word. If $S_{\text{acc}} = \emptyset$, 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](image_url)

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 $T \triangleq (T_1, \ldots, T_n)$ of transition systems, such that their sets of states are pairwise disjoint ($S_i \cap S_j = \emptyset$). We also write $(T_1 \parallel \ldots \parallel T_n)$ for the parallel composition of transition systems. The global transition system $T$ associated with a parallel transition system $(T_1 \parallel \ldots \parallel T_n)$ is defined by $T \triangleq (\Sigma, S, \Delta, S_0)$, where

- $\Sigma \triangleq \bigcup \Sigma_i$
1.2. SOME BASIC FORMALISMS

- $S \triangleq S_1 \times \cdots \times S_n$
- $S_i \triangleq S_{1i} \times \cdots \times S_{ni}$, and
- $((s_1, \ldots, s_n), (s'_1, \ldots, s'_{n})) \in \Delta$ 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 $\sigma \mid_\Sigma$ of the word $\sigma \triangleq (\sigma_1, \sigma_2, \sigma_3, \ldots)$ over some alphabet onto some other alphabet $\Sigma$ is the word $(\xi_1, \xi_2, \xi_3, \ldots)$, where $\xi_k \triangleq \sigma_m$ and $m$ is the minimal index of $\sigma$ such that $\sigma^m$ contains $k$ occurrences of letters from $\Sigma$. As an example, if $\sigma \triangleq abaa\,\ldots\,aca$ and $\Sigma \triangleq \{b, c\}$, then $\sigma^k$ contains 2 occurrences of letters from $\Sigma$, and $\sigma^7$ contains 3 such occurrences, hence $\xi_3 = \sigma_7$.

An equivalent recursive definition of this operation is

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

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

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

**Proof:** Even though the statement is rather obvious, the proof is somewhat tedious. For one direction, assume that $\sigma \mid_\Sigma \triangleq (\xi_1, \xi_2, \ldots)$ is generated by $T_i$ for all $1 \leq i \leq n$. That is, for each $i$ there is a sequence $(s_i, a, s'_i)$ of states such that $(s_i, \xi_i, s_i) \in \Delta_i$. We construct a sequence $((s'_i, m))$ of states generating $\sigma$: $s'_i \triangleq s_i$, 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 $(T_1 \parallel \ldots \parallel T_n)$ generating $\sigma$.

For the other direction, assume that $\sigma \triangleq (\sigma_1, \sigma_2, \ldots)$ is generated by $(T_1 \parallel \ldots \parallel T_n)$. Then there is a sequence $((s_{j,k})) \triangleq ((s_{1,0}, \ldots, s_{n,0}), (s_{1,1}, s_{n,1}), \ldots)$ 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 $(\xi_1, \xi_2, \ldots) \triangleq \sigma_{\xi} | \Sigma_i$ is generated by $T_i$. Let $s_{\xi} \triangleq s_{\xi,0}$ and for all $0 < k \leq |\sigma_{\xi}|$, let $s_k \triangleq s_{i,m}$, where again $m$ is the smallest index for which $\sigma^\cdot \cdot^m$ contains $k$ occurrences of letters from $\Sigma$. We now show (8): If $m > 0$ is any index such that $\sigma^\cdot \cdot^m$ contains $k$ occurrences of letters from $\Sigma$, then $s_{i,m} = s_k$. If $\sigma_m \in \Sigma$, then $m$ is minimal and (8) follows from the definition of $s_k$. If $\sigma_m \notin \Sigma_i$, then $\sigma_{\cdot \cdot \cdot}^m$ also contains $k$ occurrences of letters from $\Sigma_i$, and we can inductively assume that $s_{i,m-1} = s_k$. According to the definition of $\Delta_i$, in this case $s_{i,m-1} = s_{i,k}$. Hence $s_{i,m} = s_k$, which proves (8).

Assume now that $\xi_k$ is the $m$th letter of $\sigma$, i.e., $\sigma_m \in \Sigma_i$ and $\sigma_{\cdot \cdot \cdot}^m$ contains $k$ occurrences of letters from $\Sigma_i$. From (8) 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 $\Delta$ 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 if 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 $z_1$ and $z_2$ to record whether $A_1$ and $A_2$ have been in a recurring state. That is, $z_1$ is set whenever $A_1$ enters a recurring state, and vice versa for $z_2$ 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
1.2. SOME BASIC FORMALISMS

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 \( A = (S, \Delta, S_\text{init}, S_{\text{acc}}, S_{\text{rec}}) \), define the automaton \( A' \triangleq (S', \Delta', S'_\text{init}, S'_{\text{acc}}, S'_{\text{rec}}) \) by \( S' \triangleq (S \cup S'_{\text{acc}}) \), \( S'_{\text{acc}} \triangleq \{ s' \mid s \in S_{\text{acc}} \} \), 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 \((A_1 \mid \ldots \mid A_n)\) of Büchi-automata can be defined by:

- \( \Sigma \triangleq \bigcup \Sigma_i \)
- \( S \triangleq S'_1 \times \cdots \times S'_n \times \{0, 1\}^n \)
- \( S_\text{init} \triangleq S_{1,\text{init}} \times \cdots \times S_{n,\text{init}} \times \{(0, \ldots, 0)\} \)
- \( S_{\text{acc}} \triangleq S_{1,\text{acc}} \times \cdots \times S_{n,\text{acc}} \times \{0, 1\}^n \)
- \( S_{\text{rec}} \triangleq S'_1 \times \cdots \times S'_n \times \{(1, \ldots, 1)\} \), and
- \((s_1, \ldots, s_n, (x_1, \ldots, x_n)), a, (s'_1, \ldots, s'_n, (y_1, \ldots, y_n)) \in \Delta \) iff for all \( i \)
  - if \( a \in \Sigma_i \), then \( (s_i, a, s'_i) \in \Delta'_i \),
  - if \( a \not\in \Sigma_i \), then \( s_i = s'_i \), and
  - if \( (x_1, \ldots, x_n) = (1, \ldots, 1) \) then \( (y_1, \ldots, y_n) = (0, \ldots, 0) \) else
  - if \( s'_i \in S'_{\text{acc}, i} \) or \( a \in \Sigma_i \) and \( s'_i \in S_{\text{rec}, i} \) then \( y_i = 1 \) else \( y_i = x_i \).

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

1.2.2 \( \omega \)-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 have 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.
CHAPTER 1. MODELLING OF REACTIVE SYSTEMS

\textbf{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 \textit{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:\footnote{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 "\(\cdot\)" binds stronger than "\(+\)" and omit parenthesis whenever appropriate.}

\[ \omega R ::= a \mid \varepsilon \mid (\omega R + \omega R) \mid (\omega R \cdot \omega R) \mid \omega R^+ \mid \omega R^\omega \]

In other words,

- Every letter from the alphabet is an ω-regular expression.
- \(\varepsilon\) 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 \(t^\omega\), 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). \(\varepsilon\) 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. \(t^\omega\) 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 automata we can construct such an ω-regular expression and vice versa; Büchi-automata can define all and only ω-regular languages.

\textbf{Theorem 1.3} ω-regular expressions and Büchi-automata are of equal expressive power.
1.2. SOME BASIC FORMALISMS

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 \( \omega \)-regular expression associated with any Büchi-automaton is

\[
\Sigma \{ L(s_0, s) \mid s_0 \in S_0, s \in S_{\omega} \} + \Sigma \{ L(s_0, s) ; L(s, s) \} \mid s_0 \in S_0, s \in S_{\omega} \}
\]

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. \( \square \)

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 \( \omega \)-regular expression \( (a + b)^{\omega} \). A nondeterministic automaton for this language is given in Figure 1.3. However, there is no deterministic 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 \( \sigma \). 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 \( (a; b^{m})^{\omega} \). 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 \( \mathcal{L} \). \( \square \)

Process Algebras

Parallel composition operators for regular expressions have been widely studied in the context of process algebras [Hoa85, Ros98, Sch00]. Let \( \Sigma \) be a
finite alphabet not containing the special letters ✓ and τ. Furthermore, assume that we are given a countable set Q of process variables. A process algebraic term (in a CSP-like language) is given by the following syntax:

\[
\text{PA} ::= \text{STOP} \mid \text{SKIP} \mid Q \mid (a \rightarrow \text{PA}) \mid (\text{PA} \parallel \text{PA}) \mid (\text{PA}; \text{PA}) \mid (\text{PA} ; \text{PA}) \mid \nu Q \text{ PA} \mid \text{PA}\{R\}
\]

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 → t) is a process which can perform event a and then behave like process t. External choice is denoted by (t₁ ∥ t₂): depending on the input, this process can either execute t₁ or t₂. The sequential composition of t₁ and t₂ is denoted by (t₁ ; t₂). Parallel composition of processes (t₁ ∥ t₂) 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 νq t behaves like t, where each occurrence of q is replaced by the process νq t. Finally, t{R} is the renaming of t by R ⊆ Σ × (Σ ∪ {τ}): This is a process which behaves like t but where each event a ∈ Σ in the execution is replaced by some event b such that (a, b) ∈ 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.
1.2. SOME BASIC FORMALISMS

\begin{itemize}
\item \((\text{SKIP}, \tau, \text{STOP}) \in \Delta.\)
\item If \(a \in \Sigma\), then \(((a \rightarrow t), a, t) \in \Delta.\)
\item If \(a \in \Sigma \cup \{\tau\}\) and \((t_1, a, t'_1) \in \Delta\), then
  \(((t_1 \parallel t_2), a, t'_1) \in \Delta\) and \(((t_2 \parallel t_1), a, t'_1) \in \Delta.\)
  If \((t_1, \tau, t'_1) \in \Delta, then\)
  \(((t_1 \parallel t_2), \tau, (t'_1 \parallel t_2)) \in \Delta\) and \(((t_2 \parallel t_1), \tau, (t_1 \parallel t'_1)) \in \Delta.\)
\item 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 \((t_1, \tau, t_2) \in \Delta, then\)
  \(((t_1; t_2), \tau, t_2) \in \Delta.\)
\item If \(a \in A \cup \{\tau\}\) and \((t_1, a, t'_1) \in \Delta\) and \((t_2, a, t'_2) \in \Delta\), then
  \(((t_1 \parallel t_2), a, (t'_1 \parallel t'_2)) \in \Delta.\)
  If \(a \notin A \cup \{\tau\}\) and \((t_1, a, t'_1) \in \Delta\), then
  \(((t_1 \parallel t_2), a, (t'_1 \parallel t_2)) \in \Delta\) and \(((t_2 \parallel t_1), a, (t_2 \parallel t'_1)) \in \Delta.\)
\item If \(a \in \Sigma \cup \{\tau, \text{\textit{\small{\textbullet}}}{\text{\textit{\small{\textbullet}}}}\}\) and \((t, a, t') \in \Delta.\)
  Then \(((\nu q t), a, (t'\{q := \nu q t\})) \in \Delta.\)
\item If \(a \in \Sigma, (a, b) \in R\) and \((t, a, t') \in \Delta\), then \((t\{R\}, b, t'\{R\}) \in \Delta.\)
  If \(a \in \{\tau, \text{\textit{\small{\textbullet}}}{\text{\textit{\small{\textbullet}}}}\}\) and \((t, a, t') \in \Delta, then\)
  \((t\{R\}, a, t'\{R\}) \in \Delta.\)
\end{itemize}

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 \(\Sigma \cup \{\tau, \text{\textit{\small{\textbullet}}}{\text{\textit{\small{\textbullet}}}}\}\) 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, \(a \in \Sigma \cup \{\tau, \text{\textit{\small{\textbullet}}}{\text{\textit{\small{\textbullet}}}}\}, t, t_1, t_2, t'_1, t'_2\) are terms from \(\text{PA}\), and \(t\{q := t'\}\) denotes the term \(t\) where each free occurrence of variable \(q \in Q\) 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 \(\tau\):
CHAPTER 1. MODELLING OF REACTIVE SYSTEMS

- $t \setminus A \triangleq t\{R\}$, where $(a, b) \in R$ iff $a \in A$ and $b = \sigma$ 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 $(t \setminus A, \sigma, t' \setminus A) \in \Delta$, 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 $A \cup \{\tau, \sqrt{\cdot}\}$, 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 $a \in A \cup \{\tau, \sqrt{\cdot}\}$.

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

- $(t_1 \cap t_2) \triangleq ((a \rightarrow t_1) \sqcap (a \rightarrow t_2)) \setminus \{a\}$ 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 $\sqrt{\cdot}$.

- $(t_1 || t_2) \triangleq (t_1 \parallel t_2)$

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

- $(t_1 A || B t_2) \triangleq (t_1|_A A ||_B t_2|B)$

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 $a \in ((A \cap B) \cup \{\sqrt{\cdot}\}$, 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 $\omega$-word $\sigma$ over $\Sigma$ is called a trace of a process algebraic term, if there exists a word $\sigma'$ which is generated by the associated transition system such that $\sigma = \sigma' |_\Sigma$. A failure of $t$ is a tuple $(\sigma, a)$, such that $\sigma$ is a finite trace $(\sigma_0, \ldots, \sigma_n)$ of $t$, but $\sigma$ extended by $a$, i.e., $(\sigma_0, \ldots, \sigma_n, a)$, 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 \( \sigma \) such that some prefix \( \sigma' \) of \( \sigma \) exists which is the projection of a generated word \( \sigma'' (\sigma' = \sigma'' \mid T) \), 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 \( (m_0, t_1, m_1, t_2, \ldots) \) 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 \( \Sigma \). An \( \omega \)-word \( (\sigma_1, \sigma_2, \ldots) \) is in the language of net \( N \) if there exists a firing sequence \( (m_\emptyset, t_1, m_1, t_2, \ldots) \) 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 \( P \triangleq (V, D, T, s_\emptyset) \), where

- \( V = (v_1, \ldots, v_n) \) is a set of program variables,
- \( D = (D_1, \ldots, D_n) \) is a tuple of corresponding finite domains \( D_i = \{d_{i1}, \ldots, d_{im} \} \)
- \( T \subseteq D \times D \) is a transition relation, and
- \( s_\emptyset = (d_{i1}, \ldots, d_{in}) \) is the initial state.

A state of a shared variables program is a tuple \( (d_1, \ldots, d_n) \), 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.
1.2. SOME BASIC FORMALISMS

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

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 $\Sigma$ which is partitioned into $\Sigma^{\text{in}}$, $\Sigma^{\text{out}}$, and $\Sigma^{\text{int}}$ of input, output and internal actions. Elements of $\Sigma^{\text{in}} \cup \Sigma^{\text{out}}$ are called external actions, and elements of $\Sigma^{\text{out}} \cup \Sigma^{\text{int}}$ 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 $\sigma \in \Sigma^{\text{in}}$ 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 $M \triangleq (\Sigma^{\text{in}}, \Sigma^{\text{out}}, \mathcal{T})$, where $\Sigma^{\text{in}}$ is the set of input actions, $\Sigma^{\text{out}}$ is the set of output actions ($\Sigma^{\text{in}} \cap \Sigma^{\text{out}} = \emptyset$), and $\mathcal{T}$ is a set of words (traces) over $\Sigma \triangleq \Sigma^{\text{in}} \cup \Sigma^{\text{out}}$, such that for each $\sigma \in \mathcal{T}$, $k \geq 1$ and $\sigma_i \in \Sigma^{\text{in}}$ it holds that $(\sigma^{\cdot k}; \sigma_i) \in \mathcal{T}$. 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 component's outputs the action $a$, all other components having $a$ in its input action must also perform $a$. 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 $i \neq j$

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

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 $M = \{M_1, \ldots, M_n\}$ is a (compatible) set of I/O-modules, then we define the composed action alphabet $\Sigma$ by

- $\Sigma^{\text{out}}_C \triangleq \bigcup_i \Sigma_i^{\text{out}}$,
- $\Sigma^{\text{in}}_C \triangleq \bigcup_i \Sigma_i^{\text{in}} \cup \bigcup_i \Sigma_i^{\text{out}}$,
- $\Sigma^{\text{int}}_C \triangleq \bigcup_i \Sigma_i^{\text{int}}$.

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 $\sigma$ of a composed system $M_C$ are all those traces such that the projection of $\sigma$ onto each individual alphabet $\Sigma_i$ is a trace of the system $M_i$. 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 \( \mu \)-calculus. Restrictions and extensions of these logics are introduced whenever appropriate.

### 2.1 Propositional and First Order Logic

We assume a set \( \mathcal{P} = \{ p, q, p_1, \ldots \} \) of (atomic) propositions which can be either true or false.

For example, the proposition \texttt{stack is empty} denotes the fact that “the stack is empty”. The propositional logic PL is built from \( \mathcal{P} \) with the following syntax:

\[
\text{PL} ::= \mathcal{P} \mid \bot \mid (\text{PL} \rightarrow \text{PL})
\]

That is,

- Every \( p \in \mathcal{P} \) is a well-formed formula of propositional logic,
- \( \bot \) is a well-formed formula (“the falsum”),
- if \( \varphi \) and \( \psi \) are well-formed formulae, then so is \( \varphi \rightarrow \psi \), 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: \( \neg \varphi \triangleq (\varphi \rightarrow \bot) \), \( \top \triangleq \neg \bot \), \( (\varphi \lor \psi) \triangleq \neg (\neg \varphi \land \neg \psi) \), and \( (\varphi \leftrightarrow \psi) \triangleq ((\varphi \rightarrow \psi) \land (\psi \rightarrow \varphi)) \). The precedence of these operators is fixed by \((\neg, \lor, \land, \rightarrow, \leftrightarrow)\), 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 \{true, false\} to every proposition. (For example, the proposition \texttt{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:

- \( \mathcal{I}(\bot) = \text{false} \), and
- \( \mathcal{I}(\varphi \rightarrow \psi) = \text{true} \text{ iff } \mathcal{I}(\varphi) = \text{true} \text{ implies } \mathcal{I}(\psi) = \text{true} \).

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

- \( \mathcal{I} \models p \text{ iff } \mathcal{I}(p) = \text{true} \),
- \( \mathcal{I} \not\models \bot \), and
2.1. PROPOSITIONAL AND FIRST ORDER LOGIC

- $\mathcal{I} \models (\varphi \rightarrow \psi)$ 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 $p \in \mathcal{P}$ is augmented with an additional parameter denoting time, for example, \texttt{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 $\mathcal{R} \models \{R,a,b,\ldots\}$ of \textit{accessibility relations}, and let $\mathcal{R}^+ \models \mathcal{R} \cup \{\approx, <, =\}$. Furthermore, let $\mathcal{T}$ be a set of \textit{first-order variables} $\mathcal{T} \models \{t_0, t_1, \ldots\}$ for points in time (which is assumed to be infinite unless stated otherwise).

\[
\text{FOL} \quad ::= \quad \mathcal{P}(\mathcal{T}) \mid \bot \mid (\text{FOL} \rightarrow \text{FOL}) \mid \mathcal{R}^+(\mathcal{T}, \mathcal{T}) \mid \exists \mathcal{T} \text{ FOL}
\]

When writing formulas, we often use infix notation for relational terms: $t_1 R t_2 \models R(t_1, t_2)$. 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 $(x < y \lor x = y)$, 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 $p \in \mathcal{P}$ a subset of points $\mathcal{I}(p) \subseteq U$, 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 $\mathcal{I}(=) \models \{(w, w) \mid w \in U\}$ is the \textit{equality relation}, $\mathcal{I}(\prec) \models \bigcup \{\mathcal{I}(R) \mid R \in \mathcal{R}\}$ is the \textit{transition relation}, and $\mathcal{I}(\prec)$ is the transitive closure of $\mathcal{I}(\prec)$, the \textit{reachability relation}. A \textit{variable valuation} $\mathbf{v}$ assigns to any variable $t \in \mathcal{T}$ a point $w \in U$. A first-order model $\mathcal{M} \models (U, \mathcal{I}, \mathbf{v})$ 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:

- $\mathcal{M} \models p(t)$ iff $\mathbf{v}(t) \in \mathcal{I}(p)$;
- $\mathcal{M} \not\models \bot$, and
- $\mathcal{M} \models (\varphi \rightarrow \psi)$ 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. \((\text{stack is empty}(t_0) \rightarrow \exists t_1(\text{put}(t_0, t_1) \land \neg \text{stack is empty}(t_1)))\)
   
   If \(\text{stack is empty}\), then it is possible to perform a \text{put} such that not \text{stack is empty} holds.

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

3. \(\forall t_1((t_0 \leq t_1 \land \text{req}(t_1)) \rightarrow \exists t_2((t_1 < t_2 \land \text{ack}(t_2)) \land \forall t_3((t_1 < t_3 < t_2) \rightarrow \text{req}(t_3))))\)
   
   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 \(t \in T\); propositions \(p \in P\) are nullary again. The meaning of a proposition like \text{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 \(\varphi\)”, i.e., “there exists some \(t\) accessible via \(R\) such that \(\varphi\) holds at \(t\).” Dually, \([R]\neg\varphi = \langle R \rangle \neg \varphi\) means “necessarily \(\varphi\);” “for all \(t\) accessible via \(R\), it is the case that \(\varphi\) holds at \(t\).”

\[
\text{ML} := \mathcal{P} \mid \bot \mid (\text{ML} \rightarrow \text{ML}) \mid \langle \mathcal{R} \rangle \text{ML}.
\]

Intuitively, the above example (1) could be written

\((\text{stack is empty} \rightarrow \langle \text{put} \rangle \neg \text{stack is empty})\).

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 \(p \in P\) and \(R \in \mathcal{R}\) a subset \(\mathcal{I}(p) \subseteq U\) and a relation \(\mathcal{I}(R) \subseteq U \times U\),
respectively. Instead of a valuation for free variables, a model designates a current \( w_0 \in U \):

- \( \mathcal{M} \models p \) iff \( w_0 \in I(p) \);
- \( \mathcal{M} \not\models \perp \), and
- \( \mathcal{M} \models (\varphi \rightarrow \psi) \) iff \( \mathcal{M} \models \varphi \) implies \( \mathcal{M} \models \psi \).
- \( \mathcal{M} \models (R) \varphi \) iff there exists \( w_1 \in U \) with \( (w_0, w_1) \in I(R) \) and \( (U, I, w_1) \models \varphi \).

We write \( w \models \varphi \) instead of \( (U, I, w) \models \varphi \) whenever the context \( U \) and \( I \) are given. A formula \( \varphi \) is universally valid in \( (U, I) \), 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 \( X, F^+ \) and \( F' \):

- \( w_0 \models X \varphi \) iff there exists \( w_1 \in U \) such that \( (w_0, w_1) \in I(\prec) \) and \( w_1 \models \varphi \),
- \( w_0 \models F^+ \varphi \) iff there exists \( w_1 \in U \) such that \( (w_0, w_1) \in I(<) \) and \( w_1 \models \varphi \), and
- \( w_0 \models F' \varphi \) iff there exists \( w_1 \in U \) such that \( (w_0, w_1) \in I(\leq) \) and \( w_1 \models \varphi \).

For the dual operators, we use the symbols \( \Box \varphi \overset{\Delta}{=} \neg X \neg \varphi \), and \( G^+ \varphi \overset{\Delta}{=} \neg F' \neg \varphi \), and \( G^+ \varphi \overset{\Delta}{=} \neg F' \neg \varphi \). Traditionally, \( X, F, \) and \( G \) have been used to indicate next-time, Future and Global operators\(^1\). Alternatively, \( F^+ \) and \( G^+ \) are called sometime- and always-operators. \( \Box \) is referred to as weak next-operator. [Bur74] suggested the use of a modal logic built upon \( F^+ \) and \( G^+ \) to describe program properties. [Krö78] was the first to use \( X \) and \( F^+ \) in program verification. [Pnu77] extended this framework for parallel programs. The combination of \( (R) \)- and \( F^+ \)-operators originates from dynamic logic (for an overview on dynamic logics, see [Har84, KT90]).

Intuitively, \( X \varphi \) indicates that \( \varphi \) holds at some point accessible via a single transition, \( F^+ \varphi \) specifies that \( \varphi \) must hold in some point which can

---

\(^1\)A note on notation: With the above convention, the \( X, \Box, F^+, F', G^+ \) and \( G^+ \) operators could be written as \( \langle \cdot \rangle, [\cdot], \langle \cdot \rangle, \langle \cdot \rangle, [\cdot] \) and \( [\cdot] \), respectively. In the literature, some authors use the symbols \( \odot, \odot, \odot \), and \( \Box \).
be reached by a nonempty sequence of transitions, and $F^* \varphi$ means that
$\varphi$ holds at some reachable point (possibly now). Dually, $\Box \varphi$ holds if all
successors satisfy $\varphi$, and $G^* \varphi$ and $G^+ \varphi$ determine that all reachable points
(except maybe the current point) must validate $\varphi$. With these operators,
example (2) could be written

$$G^+ (req \rightarrow F^+ ack).$$

From the definition, $w_0 \models \Box \varphi$ if $w_1 \models \varphi$ for all $w_1 \in U$ such that $(w_0, w_1) \in I(\prec)$. Similarly, $w_0 \models G^+ \varphi$ if $w_1 \models \varphi$ for all $w_1 \in U$ such that $(w_0, w_1) \in I(\prec)$. Recall that a point $w \in U$ is terminal, if $\{w' \mid (w, w') \in I(\prec)\} = \{\}$. A terminal point represents a final state of a terminating computation. Terminal points satisfy all $\Box \varphi$ and $G^+ \varphi$-formulas vacuously: If $w_0$ has no accessible successors, then $w_0 \models \Box \varphi$ and $w_0 \models G^+ \varphi$ for any formula $\varphi$.

The difference between $F^+$ and $F^*$ is that in the latter “the future includes the present”. Clearly, $(F^* \varphi \iff \varphi \lor F^+ \varphi)$. Therefore, the $F^*$-operator can be defined by $F^+$. Using the identity $(F^+ \varphi \equiv XF^\varphi)$, $F^+$ can be defined by $X$ and $F$ with linear increase in formula length. It is not possible to define the $F^+$-operator only by $F^*$: Consider two models $M_1$ and $M_2$, where $U_1 \equiv U_2 \equiv \{w\}$, $I_1(\prec) \equiv \{\}$, $I_2(\prec) \equiv \{(w, w)\}$ and $I_1(p) = I_2(p)$ for all $p \in P$. Then $M_1 \models F^* \top$ and $M_2 \models F^+ \top$. However, $w \models F^* \varphi$ if $w \models \varphi$ in both $M_1$ and $M_2$. Therefore, for all formulas $\varphi$ which involve only propositions, boolean operators and $F^*$ it holds that $M_1 \models \varphi$ iff $M_2 \models \varphi$. (Formally, the proof of this statement is by induction on the construction of such formulas). Thus $F^+ \top$ 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 $(\varphi U^+ \psi)$ meaning “$\varphi$ holds until $\psi$ holds”. This operator was introduced in [Kam68] and used to describe properties of concurrent programs in [GPSS80]. The semantics of $U^+$ is defined as follows$^2$:

- $M \models (\varphi U^+ \psi)$ 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:

\[\cdots \rightarrow \varphi \rightarrow \varphi \rightarrow \varphi \rightarrow \psi \rightarrow \cdots\]

$^2$For the relations $\sim \in \{\prec, <, =, \leq\}$, which have a fixed interpretation, we henceforth write $v \sim w$ for $(v, w) \in I(\sim)$, whenever no confusion can arise.
2.2. MULTIMODAL AND TEMPORAL LOGIC

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

\[ G^+(\text{req} \rightarrow (\text{req} U^+ \text{ack})). \]

Various other operators can be defined via \( U^+ \). Sometime-operator and nexttime operators are obtained as follows:

- \( X \varphi \leftrightarrow (\bot U^+ \varphi) \)
- \( F^+ \varphi \leftrightarrow (\top U^+ \varphi) \)

The proof of these equivalences is immediate from the definition: \( w_0 \models (\bot U^+ \psi) \) 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., \( w_0 \xrightarrow{} w_1 \). Consequently, \( w_0 \models X \varphi \). The second equivalence is obtained in a similar way.

The reflexive until-operator is defined as \( (\varphi U^+ \psi) \triangleq (\psi \lor \varphi \land (\varphi U^+ \psi)) \).

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

The unless or weak until-operator is defined as

\[ (\varphi W^+ \psi) \triangleq \neg(\neg \psi U^+ \neg(\varphi \lor \psi)). \]

Intuitively, it says that \( \varphi \) holds at least up to the next point where \( \psi \) holds. This can be seen as follows: Assume that \( w_0 \models \neg(\neg \psi U^+ \neg(\varphi \lor \psi)) \). By definition, it is not the case that for some \( w_1 > w_0 \) both \( w_1 \models \neg(\varphi \lor \psi) \) and \( w_2 \models \neg \psi \) for all \( w_0 < w_2 < w_1 \). Thus, for all \( w_1 > w_0 \) it holds that \( w_1 \models (\varphi \lor \psi) \), 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 \( \psi \) holds, then \( w_1 \models \varphi \). \( \square \)

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

\[ (\varphi W^+ \psi) \triangleq (((\neg \varphi U^+) \lor G^+ \varphi). \]
In models which consist of a sequence of points, these two definitions are equivalent. To prove this, we must show (i) $M \models ((\varphi W^+ \psi) \rightarrow ((\varphi U^+ \psi) \lor G^+ \varphi))$, (ii) $M \models (G^+ \varphi \rightarrow (\varphi W^+ \psi))$ and (iii) $M \models ((\varphi U^+ \psi) \rightarrow (\varphi W^+ \psi))$ for all such models $M$. For (i), assume that $w_0 \models (\varphi W^+ \psi)$ and $w_0 \not\models G^+ \varphi$. Then $w_1 \not\models \varphi$ for some $w_1 > w_0$. According to above, there is some $w_0 < w_1$ 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 $w_0 \models (\varphi U^+ \psi)$. Formula (ii) follows immediately from the definition: if $w_0 \models G^+ \varphi$, 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 $w_1 \models \neg(\varphi \lor \psi)$. This implies $w_0 \not\models (\neg \psi U^+ \neg(\varphi \lor \psi))$, i.e., $w_0 \models (\varphi W^+ \psi)$. For implication (iii), we need the property that the model is linear: If $w_0 \models (\varphi U^+ \psi)$, 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 $w > w_1$. 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 $w_0 \models (\varphi W^+ \psi)$.

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

$$(\varphi U^+ \psi) \leftrightarrow ((\varphi W^+ \psi) \land F^+ \psi).$$

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 $U^-$, with the following semantics:

- $M \models (\varphi U^- \psi)$ iff $\exists w_1 (w_1 < w_0 \land \psi(w_1) \land \forall w_2 (w_1 < w_2 < w_0 \rightarrow \varphi(w_2)))$.

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

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

We write $F^-$ $\varphi$ and $G^-$ $\varphi$ for $(\top U^- \varphi)$ and $\neg F^- \neg \varphi$, respectively. Intuitively, these operators refer to “sometime in the past” and “always in the past”. Moreover, $F^+ \varphi$ and $G^+ \varphi$ are abbreviations for $(F^- \varphi \lor \varphi \lor F^+ \varphi)$ and $\neg F^- \neg \varphi$, respectively.

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

2.2. MULTIMODAL AND TEMPORAL LOGIC

- \( \text{FOL}(p) \triangleq p(t_0) \)
- \( \text{FOL}(\bot) \triangleq (t_0 \neq t_0) \)
- \( \text{FOL}(\varphi \to \psi) \triangleq (\text{FOL}(\varphi) \to \text{FOL}(\psi)) \)
- \( \text{FOL}((\varphi U^\psi) \psi) \triangleq \exists t' t_0 < t' \land \text{FOL}(\psi) \{t_0 := t'\} \land\)
  \( \forall t'' (t_0 < t'' < t' \rightarrow \text{FOL}(\varphi) \{t_0 := t''\}) \).
- \( \text{FOL}((\varphi U^\psi) \psi) \triangleq \exists t' t_0 < t_0 \land \text{FOL}(\psi) \{t_0 := t'\} \land\)
  \( \forall t'' (t' < t'' < t_0 \rightarrow \text{FOL}(\varphi) \{t_0 := t''\}) \).

In the translation of \( (\varphi U^\psi) \psi \) and \( (\varphi U^\psi) \psi \), the symbols \( t' \) and \( t'' \) denote arbitrary variables which do not occur in \( \text{FOL}(\varphi) \) or \( \text{FOL}(\psi) \). The formula \( \text{FOL}(\psi) \{t_0 := t'\} \) denotes the formula \( \text{FOL}(\psi) \), 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.

\[
\text{FOL}(\neg \text{ack} U^\text{req} U^\text{ack})
= \exists t_1 (t_0 < t_1 \land \text{ack}(t_1) \land \forall t_2 (t_0 < t_2 < t_1 \rightarrow \text{FOL}(\neg \text{ack} U^\text{req})\{t_0 := t_2\}))
= \exists t_1 (t_0 < t_1 \land \text{ack}(t_1) \land \forall t_2 (t_0 < t_2 < t_1 \rightarrow \exists t_3 (t_3 < t_2 \land \text{req}(t_3) \land \forall t_4 (t_3 < t_4 < t_2 \rightarrow \neg \text{ack}(t_4))))).
\]

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

In other words, \( \text{FOL} \) is at least as expressive as \( \text{LTL} \). 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, \( \text{FOL}(\neg \text{ack} U^\text{req} U^\text{ack}) \) is equivalent to

\[
\exists t_1 (t_0 < t_1 \land \text{ack}(t_1) \land \forall t_2 (t_0 < t_2 < t_1 \rightarrow \exists t_3 (t_3 < t_2 \land \text{req}(t_3) \land \forall t_4 (t_3 < t_4 < t_2 \rightarrow \neg \text{ack}(t_4))))).
\]

As an immediate consequence, \( \text{LTL} \) 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.

\[
\exists t_1, t_2, t_3 (t_0 < t_1 \land t_0 < t_2 \land t_0 < t_3 \land t_1 < t_2 \land t_1 < t_3 \land t_2 < t_3)
\]

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

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

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

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

which in turn can be expressed temporally as $F^+ F^+ F^+ T$.

Therefore, attention is restricted to certain classes of structures, like complete linear orders, or finitely-branching trees, etc. A natural model $\mathcal{M} \models (U, I, w_0)$ 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 $<$ interpreted as the usual successor relation.

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

**Theorem 2.2 (Kamp, Gabbay)** Temporal logic is expressively 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 modeled with an idle loop. In this book, we use $\omega$-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 programs can involve assertions about all maximal paths in a tree. A path in a model is a (finite or infinite) nonempty sequence of points \( \sigma = (w_0, w_1, \ldots) \), 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 finite, or its final point \( w_n \) is terminal (there is no \( w \) such that \( w_n < w \)). Computation tree logic (CTL) [CE81] has the following syntax:

\[
\text{CTL} := \mathcal{P} \mid \bot \mid (\text{CTL} \to \text{CTL}) \mid E(\text{CTL} U^+ \text{CTL}) \mid A(\text{CTL} U^+ \text{CTL}).
\]

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 "\(<\)" then denotes the usual tree-order: \((w_1, w_2) \in \mathcal{I}(\prec)\) if \( w_1 \) is on the (unique) path from the root \( w_0 \) up to \( w_2 \).

- \( w_0 \models E(\varphi \ U^+ \psi) \) iff there exists \( w_1 > w_0 \) such that \( w_1 \models \psi \), and for all \( w_2 \in \mathcal{U}, w_0 < w_2 < w_1 \) then \( w_2 \models \varphi \).
- \( w_0 \models A(\varphi \ U^+ \psi) \) 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 \( E \ U^+ \)-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.

\[
\begin{align*}
E X \psi & \triangleq E(\bot \ U^+ \psi), & A X \psi & \triangleq A(\bot \ U^+ \psi), \\
E \neg \psi & \triangleq \neg A X \neg \psi, & A \neg \psi & \triangleq \neg E X \neg \psi, \\
E F^+ \psi & \triangleq E(\top \ U^+ \psi), & A F^+ \psi & \triangleq A(\top \ U^+ \psi), \\
E G^+ \psi & \triangleq \neg A F^+ \neg \psi, & A G^+ \psi & \triangleq \neg E F^+ \neg \psi, \\
E(\varphi \ U^+ \psi) & \triangleq (\psi \lor \varphi \land E(\varphi \ U^+ \psi)), & A(\varphi \ U^+ \psi) & \triangleq (\psi \lor \varphi \land A(\varphi \ U^+ \psi)), \\
E F^+ \psi & \triangleq (\psi \lor E F^+ \psi), & A F^+ \psi & \triangleq (\psi \lor A F^+ \psi),
\end{align*}
\]
\[
E G^+ \psi \triangleq (\psi \land E G^+ \psi), \quad A G^+ \psi \triangleq (\psi \land A G^+ \psi)
\]
\[
E(\varphi W^+ \psi) \triangleq \neg A(\neg \psi U^+ \neg(\varphi \lor \psi)), \quad A(\varphi W^+ \psi) \triangleq \neg E(\neg \psi U^+ \neg(\varphi \lor \psi))
\]

Informally, \(E X \psi\) means that some successor node satisfies \(\psi\), and \(A \triangleleft \psi\) holds if all successors are \(\psi\). In a terminal point, \(A \triangleleft \perp\) is true, but \(A X \perp\) 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 \(\sigma\) there is no \(w_1 > w_0\), therefore each formula \(A(\varphi U^+ \psi)\) and \(E(\varphi W^+ \psi)\) must be false. As a special case, in such a point \(E X \top\) is false, but \(A \triangleleft \top\) and \(E \triangleleft \perp\) are true. In a nonterminal point, \((E X \psi \leftrightarrow E \triangleleft \varphi)\) and \((A X \psi \leftrightarrow A \triangleleft \varphi)\). Thus, if we restrict attention to models without terminal points, these operators coincide. The operators \(A X\) and \(E \triangleleft\) can be expressed by \(E X\) and \(A \triangleleft\) (with linear increase of formula length) via \((A X \psi \leftrightarrow A \triangleleft \varphi \land E X \top)\) and \((E \triangleleft \varphi \leftrightarrow E X \varphi \lor A \triangleleft \perp) \leftrightarrow (E X \top \rightarrow E X \varphi)\). Thus, all CTL nexttime-operators can be expressed in terms of \(E X\).

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

\[
E(\varphi U^+ \psi) \quad A(\varphi U^+ \psi) \quad E X \psi \quad A X \psi
\]

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

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

\[
A(\varphi U^+ \psi) \leftrightarrow (A(\varphi W^+ \psi) \land A F^+ \psi) = (\neg E(\neg \psi U^+ \neg(\varphi \lor \psi)) \land A F^+ \psi).
\]

Therefore, it is sufficient to consider only the two basic operators \(E U^+\) and \(A F^+\) in formal proofs and algorithms. Similarly, the formula \(E(\varphi W^+ \psi)\) can be replaced by \((E(\varphi W^+ \psi) \lor E G^+ \varphi)\). However, there is no negation-free “dual” characterization of \(A W^+\) and \(E U^+\).
2.3. LINEAR AND BRANCHING TIME LOGICS

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

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

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

— $A G^+ A F^+ \text{stack_is_empty}$: The proposition $\text{stack_is_empty}$ holds infinitely often on every computation path

— $A G^+ E F^+ \text{restart}$: From any state it is possible to get to a $\text{restart}$ state.

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

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

Therefore, one considers $\text{LTL}$ and $\text{CTL}$ on (nonlinear, nontree) Kripke-models $(U, I, w_0)$. In contrast to natural or tree models, Kripke-models can contain reflexive points, loops or even dense relations. We call an $\text{LTL}$-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 $\text{CTL}$-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 $\text{LTL}$ and $\text{CTL}$ are incomparable. For example, the $\text{LTL}$-formula $\varphi \triangleleft F^+ G^+ p$ is not expressible in $\text{CTL}$ (it is not the same property as $A F^+ A G^+ p$). That is, there is no $\text{CTL}$-formula $\psi$ such that $\psi$ is tree-valid in exactly the same Kripke-models in which $\varphi$ is sequence-valid. Similarly, $A G^+ E F^+ p$ is not expressible in $\text{LTL}$ (it is not the same as $G^+ F^+ p$). For more information on the expressiveness of linear versus branching time see [EL85, EH86, CD88, Eme90].

On Kripke-models, the logic $\text{CTL}^p$ (see [EL85, EH86]) subsumes $\text{CTL}$ and $\text{LTL}$ by separating path quantification ($E$) from temporal quantification ($U^+$). Thus it is possible to write e.g. $E G^+ F^+ p$. The logic $\text{CTL}^p$ is strictly
more expressive than both CTL and LTL. On binary trees, the expressiveness of 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 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 p holds on every second point of an execution sequence, without constraining the values of p in intermediate points. Formally, for a natural model where $U = (w_0, w_1, ...)$, define the new operator $G^{2n}$ by

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

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

$$G^{2n}_{\text{CTL}} \varphi \triangleq \varphi \land G^* (\varphi \rightarrow \Box \Box \Box \varphi)$$

$$(G^{2n}_{\text{FOL}} \varphi)(t_0) \triangleq \varphi(t_0) \land \forall t \geq t_0 (\varphi(t) \rightarrow \forall t_1, t_2 (t R_1 R_2 \rightarrow \varphi(t_2)))$$

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

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

Thus the $G^{2n}$ 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 $G^{2n} \varphi$ holds, it suffices to provide a “new” proposition $q$ (not occurring in $\varphi$) such that $G^{2n}_{\text{CTL}} q$ holds, and that $\varphi$ is true wherever $q$ is true. This puts an additional constraint on the “auxiliary variable” $q$, which can be considered as an “implementation detail” in the context of $\varphi$. If we neglect the value of $q$, then the models satisfying $(G^{2n}_{\text{CTL}} q \land G^* (q \rightarrow \varphi))$ are exactly those satisfying $G^{2n} \varphi$. That is, for any model $\mathcal{M}$ such that $\mathcal{M} \models (G^{2n}_{\text{CTL}} q \land G^* (q \rightarrow \varphi))$ it holds that $\mathcal{M} \models G^{2n} \varphi$, and for every model
2.4. PROPOSITIONALLY QUANTIFIED LOGICS

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

\[
G^{2n} \varphi \leftrightarrow \exists q(G_{\text{TL}}^{2n} q \land G^* (q \rightarrow \varphi))
\]

\[
(G^{2n} \varphi)(t_0) \leftrightarrow \exists q((G_{\text{FOTL}}^{2n} q)(t_0) \land \forall t \geq t_0(q(t) \rightarrow \varphi(t)))
\]

The language used in the first item is called quantified temporal logic \( q\text{TL} \) [Sis83], the language of the second item is monadic second order logic \( \text{MSOL} \).

\[
\begin{align*}
q\text{TL} & : = \mathcal{P} \mid \mathcal{Q} \mid \bot \mid (q\text{TL} \rightarrow q\text{TL}) \\
& \quad \mid (q\text{TL} \uparrow q\text{TL}) \mid (q\text{TL} \downarrow q\text{TL}) \mid \exists Q q\text{TL}.
\end{align*}
\]

\[
\begin{align*}
\text{MSOL} & : = \mathcal{P}(\mathcal{T}) \mid \mathcal{Q}(\mathcal{T}) \mid \bot \mid (\text{MSOL} \rightarrow \text{MSOL}) \\
& \quad \mid R^+(\mathcal{T}, \mathcal{T}) \mid \exists \mathcal{T} \text{ MSOL} \mid \exists Q \text{ MSOL}
\end{align*}
\]

To define this syntax, we used another syntactic category \( Q = \{q, q_0, \ldots\} \) 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, \( q\text{TL} \) has the same expressiveness as \( \text{MSOL} \).

**Lemma 2.5** On natural models, the \( \mathcal{U}^+ \)-operator in \( q\text{TL} \) is definable by \( G^\pm \) and \( X \):

\[
(\varphi \mathcal{U}^+ \psi) \leftrightarrow \forall q(G^\pm (X (\psi \lor \varphi \land q) \rightarrow q) \rightarrow q).
\]

**Proof:** Since this lemma is used several times in subsequent chapters, we give a detailed proof. For one direction, assume that \( (\varphi \mathcal{U}^+ \psi) \) is true in \( \mathcal{M} \triangleq (U, \mathcal{T}, w_0) \). To prove that \( \mathcal{M} \models \forall q(G^\pm (X (\psi \lor \varphi \land q) \rightarrow q) \rightarrow q) \), let \( \mathcal{T}'(q) \) be an arbitrary set of points, and show that \( (U, \mathcal{T}', w_0) \models (G^\pm (X (\psi \lor \varphi \land q) \rightarrow q) \rightarrow q) \). In other words, from the assumption \( w_0 \models G^\pm (X (\psi \lor \varphi \land q) \rightarrow q) \) we have to show that \( w_0 \models q \). In any natural model satisfying \( w_0 \models (\varphi \mathcal{U}^+ \psi) \), there are \( w_1, \ldots, w_n \in U \) 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 \( w_0 \models G^\pm (X (\psi \lor \varphi \land q) \rightarrow q) \), then \( w_i \models (X (\psi \lor \varphi \land q) \rightarrow q) \) for all
i \geq 0. Hence, \( w_i \models (X \psi \rightarrow q) \) and \( w_i \models (X (\varphi \land q) \rightarrow q) \) for all \( i \geq 0 \).

From \( w_n \models \psi \) it follows that \( w_{n-1} \models X \psi \). Since \( w_{n-1} \models (X \psi \rightarrow q) \), we have \( w_{n-1} \models q \). Therefore \( w_{n-1} \models (\varphi \land q) \), and \( w_{n-2} \models X (\varphi \land q) \). Since \( w_{n-2} \models (X (\varphi \land q) \rightarrow q) \), 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 \( w_0 \models \forall q (G^\xi (X (\psi \lor \varphi \land q) \rightarrow q) \rightarrow q) \) and show that \( w_0 \models (\varphi U^+ \psi) \). First, we show that there must be some \( w \succ w_0 \) satisfying \( w \models \psi \). Assume for contradiction that this is not the case. Choose \( I(q) \triangleq \{ w \mid not \ w \geq w_0 \} \). 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) \( w \not\models q \) for all \( w \succeq w_0 \). We show that (iii): \( w \models (X (\psi \lor \varphi \land q ) \rightarrow q) \) for all \( w \in U \). According to the contradiction assumption, \( w \not\models \psi \) for all \( w > w_0 \). With (iii), it follows that \( w \not\models X (\psi \lor \varphi \land q) \) for all \( w \geq w_0 \). Hence, \( w \not\models X (\psi \lor \varphi \land q) \) for all \( w \geq w_0 \). As a consequence, (iii) holds for all \( w \succeq w_0 \). If not \( w \succeq w_0 \), then (iii) is an immediate consequence of (i). From (iii), we infer that \( w_0 \models G^\xi (X (\psi \lor \varphi \land q) \rightarrow q) \). Therefore, \( w_0 \models q \), which is a contradiction to (ii).

Let \( w_1, \ldots, w_n \) be a set of points such that \( w_i < w_{i+1} \) for all \( 0 \leq i < n \), and \( w_n \) is the smallest point satisfying \( \psi \) (i.e., \( w_n \models \psi \) and \( w_i \models \neg \psi \) for all \( w_i < w < w_n \)). If \( n = 1 \), we are done: In this case \( w_0 \models X \psi \), which implies that \( w_0 \models (\varphi U^+ \psi) \). If \( n > 1 \), to prove \( w_0 \models (\varphi U^+ \psi) \) 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: \( w_0 \models \forall q (q \rightarrow F^\xi (q \land X (\psi \lor \varphi \land q))) \). Choose \( I(q) \triangleq \{ w \mid w_0 \leq w < w_i \} \). It follows that \( w_0 \models F^\xi (q \land X (\psi \lor \varphi \land q))) \). That is, there is some \( w \in U \) such that \( w \models (q \land X (\psi \lor \varphi \land q))) \). Since \( n \) is minimal, there is no \( w \in I(q) \) which satisfies \( w \models X \psi \). Therefore, it follows that there is a \( w \succeq w_0 \) such that \( w \models (q \land X (\varphi \land \neg q))) \). Since \( w_{i-1} \) is the only point with \( w_{i-1} \models (q \land X \neg q) \), we can conclude that \( w_{i-1} \models X \varphi \), i.e., \( w_i \models \varphi \). \( \square \)

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

\[
(\varphi W^+ \psi) \leftrightarrow \forall q (G^\xi (X (\neg (\varphi \lor \psi) \lor (\neg \psi \land q)) \rightarrow q) \rightarrow q) \\
\rightarrow \exists q (\neg q \land G^\xi (X ((\neg \psi \land \neg \varphi) \lor (\neg \psi \land q)) \rightarrow q)) \\
\rightarrow \exists q (\neg q \land G^\xi (\neg q \rightarrow \neg X ((\neg \psi \land (\neg \varphi \lor q)))) \\
\rightarrow \exists q (\neg q \land G^\xi (\neg q \rightarrow \neg \neg \psi (\psi \lor (\varphi \land q)))) \\
\rightarrow \exists q (q \land G^\xi (q \rightarrow \neg \neg \psi (\psi \lor (\varphi \land q))))
\]
This is an instance of the dual scheme \( \exists q (q \land G^\perp (q \rightarrow \xi)) \) with \( \xi \triangleq \neg \phi \).

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 \( \mu \)-calculus \( \mu TL \) in [EC80, Pra81, Koz83, KP83]:

\[
\mu TL := \mathcal{P} \mid \mathcal{Q} \mid \perp \mid (\mu TL \rightarrow \mu TL) \mid \langle R \rangle \mu TL \mid \nu Q \mu TL.
\]

The semantics of \( \mu TL \) can be defined by a translation into \( \text{MSOL} \):

- \( \text{MSOL}(\phi) \triangleq \text{FOL}(\phi) \), if \( \phi \in \text{ML} \)
- \( \text{MSOL}(q) \triangleq q(t_0) \), if \( q \in \mathcal{Q} \)
- \( \text{MSOL}(\nu q \phi) \triangleq \exists q (q(t_0) \land \forall t (q(t) \rightarrow \text{MSOL}(\phi\{t_0 := t\})) \)

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

- \( \text{MSOL}(\nu q \phi) := \exists q (\neg q(t_0) \land \forall t (\neg q(t) \rightarrow \neg \text{MSOL}(\phi\{t_0 := t\})) \)
- \( = \forall q (q(t_0) \lor \forall t (\neg q(t) \rightarrow \neg \text{MSOL}(\phi\{t_0 := t\})) \)
- \( = \forall q (\forall t (\text{MSOL}(\phi\{t_0 := t\}) \rightarrow q(t)) \rightarrow q(t_0)) \)

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

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

\( \mathcal{M} \models (\phi \ U^+ \psi) \iff \mathcal{M} \models \nu q \ X (\psi \lor \phi \land q) \)

**Proof:** The equivalence follows almost immediately from the definitions.

\[
\text{MSOL}(\nu q X (\psi \lor \phi \land q))
\]  
\[
= \forall q (\forall t (\text{MSOL}(X (\psi \lor \phi \land q)))\{t_0 := t\} \rightarrow q(t)) \rightarrow q(t_0))
\]  
\[
= \text{MSOL}(\forall q (G^\perp (X (\psi \lor \phi \land q) \rightarrow q)) \rightarrow q))
\]  
\[
\iff \text{FOL}((\nu q (U^+ \psi)) \quad \text{(acc. to Lemma 2.5)} \quad \Box
\]

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

\[
\mathcal{M} \models \F^+ \psi \iff \mathcal{M} \models \nu q \ X (\psi \lor q)
\]  
\[
\mathcal{M} \models (\phi \ W^+ \psi) \iff \mathcal{M} \models \nu q \ \Box (\psi \lor \phi \land q)
\]  
\[
\mathcal{M} \models \G^+ \psi \iff \mathcal{M} \models \nu q (\psi \land \Box q)
\]  
\[
\mathcal{M} \models (\phi \ U^+ \psi) \iff \mathcal{M} \models \nu q (\psi \lor \phi \land X q)
\]
For certain formulas, an alternative semantical description of the \( \nu \) and \( \mu \) quantifiers in terms of greatest and least fixed points can be given. A function \( f : 2^U \rightarrow 2^U \) 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 \( f : 2^U \rightarrow 2^U \) 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, \( f(\bigcup\{Q \mid Q \subseteq f(Q)\}) \subseteq f(\bigcup\{Q \mid Q \subseteq f(Q)\}) \).

Thus, \( f(\bigcup\{Q \mid Q \subseteq f(Q)\}) \subseteq \bigcup\{Q \mid Q \subseteq f(Q)\} \), which means \( \bigcup\{Q \mid Q \subseteq f(Q)\} \subseteq \bigcup\{Q \mid Q \subseteq f(Q)\} \).

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

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

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

Call an occurrence of a proposition variable \( q \) in a formula \( \varphi \) 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 \( (\varphi \rightarrow \psi) \) is positive, if it is a negative occurrence in \( \varphi \) or a positive occurrence in \( \psi \), and negative, if it is a positive occurrence in \( \varphi \) or a negative occurrence in \( \psi \). An occurrence of \( q \) in \( (R) \varphi \) and \( vq' \varphi \) is positive or negative, if it is positive or negative in \( \varphi \), respectively. A formula \( \varphi \) is called syntactically monotonic in \( q \), if every free occurrence of \( q \) in \( \varphi \) is positive. It is syntactically monotonic, if each sub-formula \( vq \psi \) is syntactically monotonic in \( q \).

**Lemma 2.9** If \( \varphi \) is syntactically monotonic in \( q \), then \( \varphi^M_q \) is a monotonic predicate transformer.

**Proof:** This statement can be proved by induction on the structure of \( \varphi \). 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 \( (\varphi \rightarrow \psi) \) is positive in \( q \), then \( \psi \) must be positive and \( \varphi \) must be negative in \( q \). Therefore, \( \neg \varphi \) is positive in \( q \). The induction hypothesis is that \( \psi^M_q(P) \subseteq \psi^M_q(Q) \) and \( \neg \psi^M_q(P) \subseteq \neg \psi^M_q(Q) \). From this we can infer that \( \psi^M_q(Q) \subseteq \psi^M_q(P) \). Therefore, if \( \psi^M_q(P) \subseteq \psi^M_q(Q) \) then \( \varphi^M_q(Q) \subseteq \psi^M_q(Q) \). This follows from \( \varphi^M_q(Q) \subseteq \psi^M_q(P) \subseteq \psi^M_q(Q) \). In other words, \( (\varphi \rightarrow \psi)^M_q(P) \subseteq (\varphi \rightarrow \psi)^M_q(Q) \).

For the case \( (R) \varphi \), the induction hypothesis is that \( \varphi^M_q(P) \subseteq \varphi^M_q(Q) \). Then, \( \{ w \mid \exists w' \in \top(R) \land w' \in \varphi^M_q(P) \} \subseteq \{ w \mid \exists w' \in \top(R) \land w' \in \varphi^M_q(Q) \} \). In other words, \( (R) \varphi^M_q(P) \subseteq (R) \varphi^M_q(Q) \).

Similarly, for formulas \( vq' \varphi \), where \( q \) and \( q' \) are different variables, the induction hypothesis is that \( \varphi^M_{q,q'}(P,X) \subseteq \varphi^M_{q,q'}(Q,X) \) for all \( X \). Therefore, \( X \subseteq (\varphi^M_{q,q'}(P,X) \land \varphi^M_{q,q'}(Q,X) \) for all \( X \). Consequently, \( \{ w \mid \text{for some } X, w \in X \land X \subseteq (\varphi^M_{q,q'}(P,X) \} \subseteq \{ w \mid \text{for some } X, w \in X \land X \subseteq (\varphi^M_{q,q'}(Q,X) \} \). According to the definition, this is the semantics of \( (vq' \varphi)^M_q(P) \subseteq (vq' \varphi)^M_q(Q) \). The last case is \( vq \varphi \). Since this formula has no free occurrence of variable \( q \), its denotation \( (vq \varphi)^M_q \) is a constant function. Trivially, constant functions are monotonic. \( \square \)

**Corollary 2.10** If \( \varphi \) is syntactically monotonic, then

- \( \models (vq \varphi \leftrightarrow \varphi[q := vq \varphi]) \) and \( \models (\mu q \varphi \leftrightarrow \varphi[q := \mu q \varphi]) \).

- If \( (U,I) \models (\chi \leftrightarrow \varphi[q := \chi]) \) then both \( (U,I) \models (\chi \rightarrow vq \varphi) \) and \( (U,I) \models (\chi \\rightarrow (vq \varphi \rightarrow \chi)) \).
\( (U, I) \models (\chi \rightarrow \varphi[q := \chi]) \) implies \( (U, I) \models (\chi \rightarrow \nu q \varphi) \), and
\( (U, I) \models (\varphi[q := \chi] \rightarrow \chi) \) implies \( (U, I) \models (\mu q \varphi \rightarrow \chi) \)

**Proof:** If \( \varphi \) is syntactically monotonic, then \( \varphi^M \) is monotonic according to Lemma 2.9. Theorem 2.7 asserts that \( \text{gfp}(\varphi^M) = \varphi^M(\text{gfp}(\varphi^M)) \). In the notation of Lemma 2.8, this means \( (\nu q \varphi)^M = \varphi^M((\nu q \varphi)^M) \). Moreover, \( \varphi^M((\nu q \varphi)^M) = (\varphi_q[q := \nu q \varphi])^M \). Therefore, \( M \models (\nu q \varphi \rightarrow \varphi_q[q := \nu q \varphi]) \). The other statements are shown similarly. \( \Box \)

In particular, according to Corollary 2.6, \( (\varphi \ U^+ \psi) \) and \( (\varphi \ W^+ \psi) \) in natural models are least and greatest fixed points of \( X(\psi \lor \varphi \land q) \) and \( \#(\psi \lor \varphi \land q) \), respectively. Therefore, the following recursion and induction axioms hold:

- \( \models (\varphi \ U^+ \psi) \leftrightarrow X(\psi \lor \varphi \land (\varphi \ U^+ \psi)) \) and
  \( \models (\varphi \ W^+ \psi) \leftrightarrow \#(\psi \lor \varphi \land (\varphi \ W^+ \psi)) \).
- \( \models F^+ \psi \leftrightarrow (\psi \lor X F^+ \psi) \) and \( \models G^+ \varphi \leftrightarrow (\varphi \land \# G^+ \psi) \).
- \( (U, I) \models (X(\psi \lor \varphi \land \chi) \rightarrow \chi) \) implies \( (U, I) \models ((\varphi \ U^+ \psi) \rightarrow \chi) \), and
  \( (U, I) \models (\chi \rightarrow \#(\psi \lor \varphi \land \chi)) \) implies \( (U, I) \models (\chi \rightarrow (\varphi \ W^+ \psi)) \).

As we have shown, monotonic \( \mu TL \) formulas designate 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 \( (\neg q)^M \). However, the \( MSOL \) semantics of \( \nu q \neg q \) is \( \exists q(q(t_0) \land \forall t(q(t) \rightarrow \neg q(t))) \), which is equivalent to the well-defined value \( \bot \). On general Kripke models, monotonic \( \mu TL \) is strictly weaker in expressiveness than unrestricted \( \mu TL \). Even unrestricted \( \mu TL \) 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 \( \mu TL \) formula equivalent to \( \nu q(\langle R \rangle \neg q) \).

(b) There is no \( \mu TL \) formula which is equivalent to \( \forall t(p) \)

**Proof:** For (a), consider \( \varphi \triangleq \nu q(\langle R \rangle \neg q) \). Then \( MSOL(\varphi) = \exists q(q(t_0) \land \forall t(q(t) \rightarrow \exists t'(t R t' \land \neg q(t'))) \). Choosing \( I(q) \triangleq \{I(t_0)\} \) shows that \( w \models \varphi \) iff \( w \models \exists t(t_0 R t \land t \neq t) \). There is no monotonic formula which can express this property: Consider the model \( M \triangleq (U, I) \), where \( U \triangleq \{w_0, w_1\} \), \( I(R) \triangleq \{(w_0, w_0), (w_0, w_1), (w_1, w_1)\} \) and \( I(p) = I(q) = \emptyset \) for all \( p \in P \) and \( q \in Q \). Then \( w_0 \models \varphi \) and \( w_1 \not\models \varphi \). For each monotonic formula \( \psi \), however, it holds that \( w_0 \models \psi \) iff \( w_1 \not\models \psi \). To prove this, we show by induction on the
structure of $\psi$ that $\psi^\mathcal{M} = \emptyset$ or $\psi^\mathcal{M} = U$. For propositional formulas, this is immediate; the case $(R) \psi$ 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 $\psi$ 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 $\varphi \triangleq \forall p(t)$ can be invalidated by adding an isolated point $w$ with $w \not\models p$. Formally, consider the models $\mathcal{M}_0 \triangleq (U_0, \mathcal{I}, w_0)$ and $\mathcal{M}_1 \triangleq (U_1, \mathcal{I}, w_1)$, where $U_0 \triangleq \{w_0\}, U_1 \triangleq \{w_1, w\}, \mathcal{I}(R) \triangleq \emptyset$ and $\mathcal{I}(p) \triangleq \mathcal{I}(q) \triangleq \{w_0\}$. Then $\mathcal{M}_0 \models \varphi$ and $\mathcal{M}_1 \not\models \varphi$, whereas for each $\mu \mathbf{TL}$ formula $\psi$ it holds that $\mathcal{M}_0 \models \psi$ iff $\mathcal{M}_1 \models \psi$. □

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

\[
\mathcal{M} \models \nu q \varphi \quad \text{iff} \quad \mathcal{M} \models \exists q (q \land G^\varphi (q \rightarrow \varphi)), \\
\mathcal{M} \models \mu q \varphi \quad \text{iff} \quad \mathcal{M} \models \forall q (G^\varphi (\varphi \rightarrow q) \rightarrow q).
\]

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

## 2.5 Automata and Logics

Let $(U, \mathcal{I})$ be universe and interpretation of a natural model. Formula $\varphi$ 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 $\varphi$ is universally valid iff $G^\varphi \varphi$ is initially valid, and $\varphi$ is initially valid iff $(G^\varphi \bot \rightarrow \varphi)$ 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 labeling function \( \mathcal{L} : U \to 2^P \) by

\[
p \in \mathcal{L}(w) \quad \text{iff} \quad w \in I(p)
\]

That is, \( \mathcal{L}(w) \triangleq \{ p \mid w \in I(p) \} \) is the label of point \( w \in U \). If \( U = (w_0, w_1, w_2, \ldots) \), then the sequence \( \sigma = (\mathcal{L}(w_0), \mathcal{L}(w_1), \mathcal{L}(w_2), \ldots) \) is called the \( \omega \)-word of \( \mathcal{M} \) over the alphabet \( \Sigma \triangleq 2^P \). A set of \( \omega \)-words is called an \( \omega \)-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 \( \omega \)-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 \( \varphi \) there exists an LTL or qTL future formula (without \( U^\bot \)-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 \( \exists q(\varphi \lor \psi) \leftrightarrow (\exists q \varphi \lor \exists q \psi) \). Moreover, if \( \varphi_1, \ldots, \varphi_n \) are pure past, and \( \psi_1, \ldots, \psi_m \) are pure present or future, then \( \exists q(\varphi_1 \land \ldots \land \varphi_n \land \psi_1 \land \ldots \land \psi_m) \) is equivalent to \( \exists q(\varphi_1 \land \ldots \land \varphi_n) \land \exists q(\psi_1 \land \ldots \land \psi_m) \). If the past-formulas are true in a model \((U, I, w_0)\) where \( I(q) \triangleq Q_1 \) and the present- and future-formulas are true if \( I(q) \triangleq Q_2 \), then the conjunction of past, present and future part is true if \( I(q) \triangleq \{ w \mid w < w_0 \} \cap Q_1 \cup \{ w \mid w \geq w_0 \} \cap Q_2 \). Thus, using disjunctive normal form, propositionally quantified formulas can be separated. Given a separated formula \( \varphi \), let \( \varphi^+ \) be the formula \( \varphi \) where every sub-formula \((\varphi \lor \psi)\) is replaced by \( \bot \). Then \( \varphi \) is initially valid in any natural model \( \mathcal{M} \) iff \( \psi^+ \) is initially valid in \( \mathcal{M} \). Thus \( \varphi \) and \( \varphi^+ \) define the same language.

Languages can also be defined by (\( \omega \)-)regular expressions and by finite (\( \omega \)-)automata.

Trivially, \( \omega \)-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.
2.5. AUTOMATA AND LOGICS

Proof: According to Lemma 2.12 it suffices to give a translation for formulas without $U^-$. Automata for propositions are trivial two-state machines (with $2^{|P|} - 1$ transitions between these two states). An automaton for $\perp$ is one which never accepts. From an automaton for $\varphi$, an automaton for $X \varphi$ and $F^+ \varphi$ can be built by an appropriate prefixing with a single step or loop on the initial states. According to Lemma 2.5, $U^+$ can be expressed with $X$, $F^+$ and second order quantification. Implications $(\varphi \rightarrow \psi)$ can be written as $(\neg \varphi \lor \psi)$ 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 describe a tableaux decision procedure, which can be seen as an efficient algorithm to construct a Büchi-automaton from a formula.

$\omega$-regular expressions are at most as expressive as $\mu$TL:

Lemma 2.14 For every $\omega$-regular expression there exists a $\mu$TL-formula describing the same language.

Proof: The proof associates with every $\omega$-regular expression $\varphi$ a $\mu$TL-formula $\mu$TL($\varphi$) with at most one free proposition variable $q$ indicating the end of the sequence.

- $\mu$TL($P$) $\triangleq$ ($\bigwedge_{p \in P} p \land \bigwedge_{\neg p \in P} \neg p \land q$), if $P \in 2^P$
- $\mu$TL($\epsilon$) $\triangleq$ $\perp$
- $\mu$TL($\varphi + \psi$) $\triangleq$ ($\mu$TL($\varphi$) $\lor$ $\mu$TL($\psi$))
- $\mu$TL($\varphi; \psi$) $\triangleq$ $\mu$TL($\varphi$) $\{q := X \mu$TL($\psi$) $\}$
- $\mu$TL($\varphi^+$) $\triangleq$ $\nu q_1$ ($\mu$TL($\varphi$) $\{q := q \lor X q_1\}$)
- $\mu$TL($\varphi^\omega$) $\triangleq$ $\nu q_1$ ($\mu$TL($\varphi$) $\{q := X q_1\}$)

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

As an example, consider the expression $(-p1)^\omega + (T^+; p2)^\omega$.

$\mu$TL($(-p1)^\omega + (T^+; p2)^\omega$)  
$= \nu q_1$ ($\mu$TL($-p1$) $\{q := X q_1\}$) $\lor$ $\nu q_2$ ($\mu$TL($T^+; p2$) $\{q := X q_2\}$)
\[ \nu q_1 (\neg p1 \land q) \{ q := X \, q_1 \} \lor \\
\nu q_2 (\mu TL (\top^+) \{ q := X \mu TL (p2) \}) \{ q := X \, q_2 \} \]
\[ = \nu q_1 (\neg p1 \land X \, q_1) \lor \\
\nu q_2 (\mu q_2 (\top \land q) \{ q := q \lor X \, q_3 \}) \{ q := X \, (p2 \land q) \} \{ q := X \, q_2 \}) \]
\[ = \nu q_1 (\neg p1 \land X \, q_1) \lor \nu q_2 (\mu q_3 (X \land q_2) \lor X \, q_3) \lor \\
\nu q_1 (\neg p1 \land \top \land q) \lor \nu q_2 (\mu q_3 (X \land q_2) \lor X \, q_3) \]
\[ = G' (\neg p1 \land X \land q) \lor G' F' p2 \]

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

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

\[ \mu TL = q TL = MSOL = \text{Büchi-automata} = \omega \text{-regular expressions} \]

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

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 (\( \mu TL / q TL \) on tree models) and tree automata (\( \Delta \subseteq S \times 2^P \times (\mathcal{R} \times S)^n \)) (see [Niw88, Tho90, Sch92b]).

### 2.6 Relational \( \mu \)-calculus

In this section we introduce yet another, richer logical language which is closer to functional and logic programming paradigm. This is the relational \( \mu \)-calculus defined in [Par74]. Informally, this \( \mu \)-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.
2.6. RELATIONAL μ-CALCULUS

A signature $\Sigma = (D, \mathcal{F}, \mathcal{R})$ consists of a finite set $D$ 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 $\tau$, 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 $I$ for a signature $\Sigma$ on a structure $S$ is a mapping $I : \Sigma \rightarrow S$ assigning a nonempty domain $D^S \triangleq I(D)$ for each domain name $D$, and a function $f^S \triangleq I(f)$ and relation $R^S \triangleq I(R)$ of appropriate type for each function and relation symbol, respectively. That is, if $\tau(f) = (D_0, ..., D_n)$, then $f^S \in (D_0^S \times \cdots \times D_n^S \rightarrow D^S)$, 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 $\Sigma_1 \triangleq (\mathbb{N}, \{\}, \{\lt, \lt, =\})$ is decidable, whereas the set of valid formulas in $\Sigma_2 \triangleq (\mathbb{N}, \{+, \star\}, \{=\})$ is not recursively enumerable.

Given a signature $\Sigma$, 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 $\mu$-calculus, there are two more syntactic categories: well-formed formulas and relation terms of type $\tau$. Assuming that the symbols $(, )$, $\perp$, $\rightarrow$, $\equiv$, $\exists$, $\mu$, and $\lambda$ are not in the signature, a well formed formula $\phi$ is built according to the following syntax:

- $\perp$, $(\phi \rightarrow \psi)$, where $\phi$ and $\psi$ 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 \( \varphi \) is a well formed formula, and \( x \) is an individual variable, or

• \( \rho t_1 \ldots t_n \), where \( \rho \) is a relation term of type \((D_1, \ldots, D_n)\) (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 \( \mu \)-calculus, more complex relations can be specified via \( \lambda \)-abstraction and \( \mu \)-recursion. In this calculus, a relation term \( \rho \) of type \((D_1, \ldots, D_n)\) is

• a relation symbol \( R \) or relation variable \( X \) of type \((D_1, \ldots, D_n)\),

• \( \lambda x_1 \ldots x_n(\varphi) \), where \( \varphi \) 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 \((D_1, \ldots, D_n)\), and \( \rho \) a relation term of the same type which is syntactically monotone in \( X \).

As above, in this definition \( \rho \) 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 \( \rho \) is monotone in the lattice of values for \( X \) and thus the least fixpoint of this functional exists.

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

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

• \( x^\mathcal{M} \triangleq \nu(x) \), if \( x \in \mathcal{V} \) is an individual variable, and

• \( (ft_1 \ldots t_n)^\mathcal{M} \triangleq I(f)(t_1^\mathcal{M} \ldots t_n^\mathcal{M}) \), i.e., the value of function \( f^S \) at \( (t_1^\mathcal{M} \ldots t_n^\mathcal{M}) \).

• \( \bot^\mathcal{M} \triangleq \text{false} \),
2.6. RELATIONAL $\mu$-CALCULUS

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

For closed terms $t$ (not containing any free variables), the denotation $t^M$ is independent of the variable valuation $v$. Assume that the object term $t$ of type $D$ contains free individual variables $x_1, \ldots, x_n$ of type $D_1, \ldots, D_n$. For any model $M$, the denotation $t^M$ is an object in the domain $\mathcal{I}(D)$. For any frame $F \equiv (S,I)$, however, we define $t^F$ to be a function $t^F : \mathcal{I}(D_1) \times \cdots \times \mathcal{I}(D_n) \to \mathcal{I}(D)$. The function value is given by $t^F(d_1, \ldots, d_n) = t^F(S,I,v')$, where $v'$ 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 $(D_1, \ldots, D_n)$, then $t^F$ 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 $\lambda$ and $\mu$ are similar to the operators used in $\lambda$-calculus and in denotational semantics. In fact, we could define well formed formulas to be object terms of a special type $\text{boolean}$. Relation terms could then be defined as function terms with boolean result, and the $\lambda$ abstraction builds such a function term from a boolean object term.

The relational $\mu$-calculus extends first order logic in a similar way as the propositional $\mu$ 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 $\mu$ calculus. In addition, the relational $\mu$ 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 $\mu$-calculus there is no $\lambda$-abstraction on function or relation variables. This would result in a second-order calculus. In contrast to second order logic, there is no $\mu$-calculus formula expressing that domain $D$ is finite [Par74]. On the other hand, the minimization operator can be expressed in second order logic:

$$\mu X(\rho)\exists \leftrightarrow \forall X(\forall \rho \exists (\rho \exists \rightarrow X \exists) \rightarrow X \exists)$$

Since the induction axiom for arithmetic can be formulated as a least fixpoint formula, the natural numbers have a categorical theory in the relational $\mu$-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 $\mu$-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:

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

In the relational $\mu$-calculus, this definition can be written as

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

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 $\rho$ is the relation term

$$\lambda xyz((x = 0 \land y = z) \lor \exists uv(u \prec x \land v \prec z \land Xuv)).$$

The calculation is as follows.

\[
\begin{align*}
Add & \ 123 \\
\leftrightarrow & \ \rho\{X := Add\} \ 123 \\
\leftrightarrow & \ \lambda xyz((x = 0 \land y = z) \lor \exists uv(u \prec x \land v \prec z \land Add \ uv))123 \\
\leftrightarrow & \ ((1 = 0 \land 2 = 3) \lor \exists uv(u \prec 1 \land v \prec 3 \land Add \ uv)) \\
\leftrightarrow & \ Add \ 022 \\
\leftrightarrow & \ \lambda xyz((x = 0 \land y = z) \lor \exists uv(u \prec x \land v \prec z \land Add \ uv))022 \\
\leftrightarrow & \ ((0 = 0 \land 2 = 2) \lor \exists uv(u \prec 0 \land v \prec 2 \land Add \ uv)) \\
\leftrightarrow & \ T.
\end{align*}
\]

As another example, consider the following Prolog program.
2.6. RELATIONAL $\mu$-CALCULUS

\[
\begin{align*}
\text{parent}(\text{john}, \text{mary}). \\
\text{parent}(\text{jane}, \text{john}). \\
\text{ancestor}(X, Y) & : \neg \text{parent}(X, Y). \\
\text{ancestor}(X, Y) & : \neg \text{parent}(X, Z), \text{ancestor}(Z, Y). \\
\neg & \text{ ancestor}(\text{jane}, \text{mary}).
\end{align*}
\]

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

\[
\begin{align*}
\text{parent} & \equiv \lambda xy \left( x = \text{john} \land y = \text{mary} \lor x = \text{jane} \land y = \text{john} \right) \\
\text{ancestor} & \equiv \mu X \left( \lambda xy \left( \text{parent} \ x y \lor \exists z \left( \text{parent} \ x z \land X \ y z \right) \right) \right).
\end{align*}
\]

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

\[
\begin{align*}
\text{ancestor} & \ \text{jane} \ \text{mary} \\
& \leftrightarrow \mu X \left( \lambda xy \left( \text{parent} \ x y \lor \exists z \left( \text{parent} \ x z \land X \ y z \right) \right) \right) \ \text{jane} \ \text{mary} \\
& \leftrightarrow \lambda xy \left( \text{parent} \ x y \lor \exists z \left( \text{parent} \ x z \land \text{ancestor} \ x y \right) \right) \ \text{jane} \ \text{mary} \\
& \leftrightarrow \left( \text{parent} \ \text{jane} \ \text{mary} \lor \exists z \left( \text{parent} \ \text{jane} \ z \land \text{ancestor} \ z \ \text{mary} \right) \right) \ \text{jane} \ \text{mary} \\
& \leftrightarrow \left( \left( \text{jane} = \text{john} \land \text{mary} = \text{mary} \lor \text{jane} \land \text{mary} = \text{john} \right) \lor \right. \\
& \left. \exists z \left( \left( \text{jane} = \text{john} \land z = \text{mary} \lor \text{jane} = \text{jane} \lor z = \text{john} \right) \land \right. \\
& \left. \text{ancestor} \ z \ \text{mary} \right) \ \text{jane} \ \text{mary} \\
& \leftrightarrow \text{ancestor} \ \text{jane} \ \text{mary} \\
& \leftrightarrow \left( \text{parent} \ \text{jane} \ \text{mary} \lor \exists z \left( \text{parent} \ \text{jane} \ z \land \text{ancestor} \ z \ \text{mary} \right) \right) \ \text{jane} \ \text{mary} \\
& \leftrightarrow \top.
\end{align*}
\]

Similarly, the term $\lambda x (\text{ancestor} \ x \ \text{jane})$ evaluates to $\{\text{john}, \text{mary}\}$. More complicated properties like $\forall x \exists y \ (x \neq y \land \neg \text{ancestor} \ x y)$ can be formulated in the relational $\mu$-calculus. In general, such formulas need more powerful proof principles than $\lambda$-substitution and $\mu$-unfolding. In the above example, all domains are finite. Therefore, the result to this and similar queries can be calculated completely automatically 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 $\omega$-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 $\omega$-automata and linear temporal formulas via the $\omega$-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 $\mathcal{M} = (U, I, w_0)$ 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 $\sigma = (\sigma_0, \sigma_1, \sigma_2, \ldots)$ be an $\omega$-word, where $\sigma_i = (P_i, R_i)$. We say that $\sigma$ is generated by $\mathcal{M}$ if there exists a mapping $\rho$ from letters of $\sigma$ 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 $(w, w') \in I(R_i)$, and
4. if $\sigma$ is finite with last letter $\sigma_n$, and $\rho(\sigma_n) = w$, then $w$ is terminal (i.e., there is no $w'$ such that $w \prec w'$).

(Recall that $\mathcal{L}(w) \triangleq \{ p \mid p \in I(w) \}$ 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 $\omega$-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 $\mathcal{M} = (U, I, w_0)$ 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 $S \triangleq U \cup \{ \text{stop} \}$, where $\text{stop}$ 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 $(w, s) \in I(R)$ or $w$ is terminal and $s = \text{stop}$. Then, $\mathcal{M}_\mathcal{A}$ accepts exactly the set of all normal 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 $\varphi$ be an LTL-formula. Then $\varphi$ is sequence-valid in $\mathcal{M}$ iff the language generated by $\mathcal{M}$ (i.e., the language accepted by the weakly fair transition system $\mathcal{M}_A$ for $\mathcal{M}$) is a subset of the language accepted by the Büchi-automaton $\mathcal{M}_\varphi$ for $\varphi$. That is,\[
\mathcal{M} \models \varphi \quad \text{iff} \quad L(\mathcal{M}_A) \subseteq L(\mathcal{M}_\varphi).\]
The latter condition is equivalent to $L(\mathcal{M}_A) \cap \overline{L(\mathcal{M}_\varphi)} = \emptyset$, or $L(\mathcal{M}_A \times \mathcal{M}_{\neg \varphi}) = \emptyset$. Therefore, model checking of LTL sequence-validity in finite models reduces to the nonemptiness problem of Büchi-automata: a feasible way to check whether $\mathcal{M} \models \varphi$ is to construct the Büchi-automata $\mathcal{M}_A$ for the model and $\mathcal{M}_{\neg \varphi}$ for $\neg \varphi$, and to check whether the language of the product automaton $\mathcal{M}_A \times \mathcal{M}_{\neg \varphi}$ is empty.

If both system $\mathcal{M}$ and property $\varphi$ are given as automata, then "specification" $\varphi$ can be regarded as a "more abstract version" of the "implementation" $\mathcal{M}$. We write $\mathcal{M}_1 \models \mathcal{M}_2$ if $L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$, i.e., if (the language of) $\mathcal{M}_1$ is a subset of (the language of) $\mathcal{M}_2$. A property $\varphi$ is defined to be just any $\omega$-language $\varphi \subseteq \Sigma^\omega$, where $\Sigma = 2^P \times \mathcal{R}$.

**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 $\varphi$, if $\mathcal{M}_2 \models \varphi$ then $\mathcal{M}_1 \models \varphi$.

2. $\mathcal{M}_1 \models \mathcal{M}_2$ iff for all $\omega$-regular $\varphi$, 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 $\varphi$ 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 $\omega$-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 $\mathcal{M}'$.

- $\varphi$ is a safety property, iff for every natural model $\mathcal{M}$,\[
\mathcal{M} \models \varphi \quad \text{if} \quad \forall i \exists \mathcal{M}' : \mathcal{M}^{[i]} \circ \mathcal{M}' \models \varphi
\]
CHAPTER 3. MODEL TRANSFORMATIONS

This definition is from [AS85]. An $\omega$-language $\varphi$ is a safety property if for every model not satisfying $\varphi$ there is a finite prefix $M^{[i]}$ which can not be completed by any continuation $M'$ such that $M^{[i]} \circ M' \models \varphi$. In other words, for every model dissatisfying $\varphi$ 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].

- $\varphi$ is a liveness property, iff for every natural model $M$,

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

A liveness property $\varphi$, 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 $\varphi$, hence $\varphi$ 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 $\varphi$ there exists a safety property $\varphi_S$ and a liveness property $\varphi_L$ 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 $\varphi$ is the topological closure of $\varphi$, that is, the least safety property containing $\varphi$. As an example, on natural models the LTL-formula $(p \neq q)$ is equivalent to $((p \neq q) \land \mathbf{F}^+ q)$, where the language defined by $(p \neq q)$ is a safety property and the language defined by $\mathbf{F}^+ q$ 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 $\bot, \top, \land, \lor$ and $W^+$ defines a safety property. In the proof, the only interesting case
is \((\varphi \mathbf{W}^+ \psi)\). Assume that any model \(\mathcal{M}\) falsifying both \(\varphi\) and \(\psi\) 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 \(w_j \models (\neg \varphi \land \neg \psi)\), and \(w_k \models \neg \psi\) for \(w_0 < w_k < w_j\). Therefore, in any model \(\mathcal{M}^{[i-]} \circ \mathcal{M}'\), the formula \((\varphi \mathbf{W}^+ \psi)\) must be false. \(\square\)

Another syntactical characterization of safety in linear temporal logic is with past-operators. Any LTL formula \(G^+ \psi\), where \(\psi\) 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 \((\Sigma, S, \Delta, S_0)\) finitary, if \(S_0\) is finite and \(\Delta\) 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 \(\sigma\), if \(\forall i \exists \sigma' : \sigma^{[i]} \cdot \sigma' \in L\) then \(\sigma \in L\). In other words, assume that any finite prefix of \(\sigma\) can be extended to a string in \(L\) and show \(\sigma \in L\). If \(\sigma\) is finite, then it is a finite prefix of itself; thus there exists some \(\sigma'\) such that \(\sigma \cdot \sigma' \in L\). Since every state of a transition system is accepting, it follows that \(\sigma \in L\). If \(\sigma\) is infinite, consider the following computation tree: each node is marked by \((s, \sigma^{[i-1]}\), where \(s\) is a state of the transition system and \(\sigma^{[i]}\) is a finite prefix of \(\sigma\). 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 \((s_0, \sigma_0)\), where \(\sigma_0 = \sigma^{[0]}\) is the first letter of \(\sigma\). Given a node marked \((s, \sigma^{[i-1]}\) (where \(i > 0\)), for any \(s'\) such that \((s_0, \sigma_0, s') \in \Delta\) there is a child node in the tree marked \((s', \sigma_{i-1})\). Thus there exists a node marked \((s, \sigma^{[i-1]}\) iff there is a path from some initial state to state \(s\) which is labelled by \((\sigma_0, \ldots, \sigma_{i-1})\). Since \(S_0\) is finite and \(\Delta\) is image finite, the computation tree is finitely branching. Since every prefix of \(\sigma\) can be extended to a string which is accepted by the transition system, the tree contains infinitely many nodes. Thus, by König’s lemma, it must contain an infinite branch. Therefore, there is a path in the transition system labelled by \(\sigma\). Since all states in a transition system are recurring, it accepts \(\sigma\). \(\square\)

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 \((F^i X \bot)\), which is not a safety
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 \( \omega \)-regular safety property. A reverse statement also holds:

**Lemma 3.4** For every \( \omega \)-regular safety property there is a finite transition system defining this property.

**Proof:** Assume that a Büchi-automaton defining a certain safety property \( \varphi \) 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 \( \varphi \) is a safety property, for any accepted path \( \rho \) 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 \( \rho_1 \) leads into the nonaccepting SCC. Consider the (nonaccepted) path \( \rho_1 \circ \sigma^n \) 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.
3.3. SIMULATION RELATIONS

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

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

Given a finite Kripke model \( \mathcal{M} \) and an \( \omega \)-regular safety property \( \varphi \), checking whether \( \mathcal{M} \) sequence-validates \( \varphi \) 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 \( \varphi \). As above, \( \mathcal{M} \models \varphi \) if \( L(\mathcal{M}_\mathcal{A}) \subseteq L(\mathcal{M}_\varphi) \). 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 \( \omega \)-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 \( \varphi \) 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 \( \varphi \), 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 \( \sigma \) accepted by \( \mathcal{M}_1 \) such that \( \sigma \not\in \varphi \). 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 \( \mathcal{M}_1 \models \mathcal{M}_2 \). \( \Box \)

3.3 Simulation Relations

The above characterization results concentrate on containment between the \( \omega \)-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 $M_1$ and $M_2$ of Figure 3.2 over $\mathcal{P} = \{\}$ and $\mathcal{R} = \{a, b, c\}$.

![Figure 3.2: Two sequence-equivalent but branching-inequivalent Kripke-models](image)

Clearly, $L(M_1) = L(M_2)$, and therefore $M_1 \models M_2$. That is, if we observe sequences of transitions, then every possible behavior of $M_1$ is also a possible behavior of $M_2$. However, if we observe not only transitions which are taken, but also transitions which could be taken, then the behavior of $M_1$ and $M_2$ differs: If “possible continuations” are indicated by small light bulbs, then in the first system after performing 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 $\varphi$ it holds that $\varphi$ is sequence-valid in $M_1$ iff $\varphi$ is sequence-valid in $M_2$. For $\varphi \triangleq [a] (\perp \lor [c] \perp)$, it holds that $M_2 \models \varphi$, but $M_1 \not\models \varphi$.

Given two models $M_1 = (U_1, I_1, w_1)$ and $M_2 = (U_2, I_2, w_2)$, we say that $M_1$ is a submodel of $M_2$ (denoted by $M_1 \subseteq M_2$), if $U_1 \subseteq U_2$, $I_1 = I_2 \upharpoonright U_1$, 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 $M_1 = (U_1, I_1, w_1)$ and $M_2 = (U_2, I_2, w_2)$, a relation $H \subseteq U_1 \times U_2$ is called a simulation relation between $M_1$ and $M_2$ if

- $(w_1, w_2) \in H$,
- For all $p \in \mathcal{P}$, $u \in U_1$, and $v \in U_2$, if $(u, v) \in H$ then $u \in I_1(p)$ iff
3.3. SIMULATION RELATIONS

\[ v \in I_2(p). \]

- For all \( u \) and \( v \) such that \((u, v) \in H\) and all \( R \) and \( u' \) such that \((u, u') \in I_1(R)\) there is a \( v' \) with the property that \((v, v') \in I_2(R)\) and \((u', v') \in H\).

Figure 3.3 illustrates the third condition.

![Simulation condition for u and v](image)

We say that \( M_1 \) is simulated by \( M_2 \), or \( M_2 \) simulates \( M_1 \) (denoted by \( M_1 \lesssim M_2 \)), if there exists a simulation relation \( H \) between \( M_1 \) and \( M_2 \). Simulation relates a model \( M_1 \) to an abstraction \( M_2 \) of the model \( M_1 \). 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. \( \square \)

If \( M_1 \subseteq M_2 \), then \( M_1 \lesssim M_2 \). Moreover, if \( M_1 \lesssim M_2 \), then \( M_1 \models M_2 \): If \( M_2 \) can simulate \( M_1 \), then for every maximal run \( \sigma \) generated by \( M_1 \) there exists a corresponding \( \sigma' \in M_2 \). 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 \((w, w') \in \mathcal{T}(R)\). For deterministic \( M_2 \) also the converse holds: \( M_1 \models M_2 \) if \( M_1 \lesssim M_2 \). 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 $\mathcal{M}_1 = (U_1, I_1, w_1)$ and $\mathcal{M}_2 = (U_2, I_2, w_2)$, and $(w'_1, w'_2) \in H$. Then $(U_1, I_1, w'_1) \preceq (U_2, I_2, w'_2)$.

**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 $\varphi$ and $\psi$ are modal box formulas, then $(\varphi \land \psi)$, $(\varphi \lor \psi)$ 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 $\mathcal{M}_1 = (U_1, I_1, w_1)$ and $\mathcal{M}_2 = (U_2, I_2, w_2)$ be Kripke-models. Then $\mathcal{M}_1 \preceq \mathcal{M}_2$ implies that for all modal box properties $\varphi$, if $\mathcal{M}_2 \models \varphi$ then $\mathcal{M}_1 \models \varphi$.

**Proof:** The proof is by induction on $\varphi$. The base cases $\bot, \top$ are trivial. For $p \in P$, the assumption $(w_1, w_2) \in H$ implies $w_1 \in I_1(p)$ iff $w_2 \in I_2(p)$. For boolean operators $\land, \lor$, 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 $(w_1, w'_1) \in I_1(R)$ and $w'_1 \not\models \varphi$. Since $\mathcal{M}_1 \preceq \mathcal{M}_2$, there is a $w'_2 \in U_2$ such that $(w_2, w'_2) \in I_2(R)$ and $(w'_1, w'_2) \in H$. Lemma 3.7 asserts that $(U_1, I_1, w'_1) \preceq (U_2, I_2, w'_2)$. 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 $\text{ACTL}$ [Lon93, CGL93, CLM89, DGG94] is “CTL without E quantifier”. That is, literals and $\top, \bot$ are $\text{ACTL}$ formulas, and if $\varphi$ and $\psi$ are $\text{ACTL}$ formulas, then $(\varphi \land \psi), (\varphi \lor \psi), A(\varphi \ U \psi)$ and $A(\varphi \ W^+ \psi)$ are $\text{ACTL}$ formulas, where $A(\varphi \ W^+ \psi) \equiv \neg A(\neg \psi \ U^+ \neg(\varphi \lor \psi))$.

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

**Proof:** Intuitively, this theorem is true because formulas in $\text{ACTL}$ describe properties that are true of all paths of a model. They cannot express...
the existence of a specific path in the model. If $M_1 \preceq M_2$, then every behavior of $M_1$ is a behavior of $M_2$. Thus every formula of $\text{ACTL}$ that is true in $M_2$ must also be true in $M_1$. Formally, if $w_1 \not\models A(\varphi U^m \psi)$, then there is in $M_1$ 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 $w_1^{(n)} \not\models (\varphi \lor \psi)$, 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 $w_2^{(n)} \not\models (\varphi \lor \psi)$, or $w_2^{(i)} \not\models \psi$ for all $i > 0$. Thus $w_2 \not\models A(\varphi U^m \psi)$. \hfill \Box

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. $M_1$ has an additional branch of infinite length. These two models cannot be distinguished by any modal formula:

**Lemma 3.10** For any $\varphi \in \text{ML}$ it holds that $M_1 \models \varphi$ iff $M_2 \models \varphi$. 

**Proof:** The statement is proved by induction on $\varphi$. The only interesting case is $\varphi = (R) \psi$, $M_1 \models \varphi$, and the successor $w_1'$ of $w_1$ for which $w_1' \models \psi$
is on the additional infinite branch of $M_1$. Choose any branch of $M_2$ of
length at least $n$, where $n$ is the number of modal operators in $\varphi$. Denote the $i$-th point on the infinite branch of $M_1$ and on the chosen branch of $M_2$
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 $\xi_i$ of $\varphi$ 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 $\varphi$ has $n$ modal operators, $w_1^{(n)} \models \varphi$ iff $w_2^{(n)} \models \varphi$. \hfill \Box

In particular, Lemma 3.10 implies that for every modal box formula $\varphi$, if $M_2 \models \varphi$ then $M_1 \models \varphi$. Yet, $M_2$ does not simulate $M_1$: Assume a simulation relation $H$ mapping the first node $w$ of the infinite path of $M_1$ to any node $w'$ of any finite path in $M_2$. 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 no nodes $u \in M_1$ and $v \in M_2$ such that $(u,v) \in H$, and $u$ has a successor in $M_1$, but $v$ has no successor in $M_2$.

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 $M_1$ and $M_2$ be image finite Kripke-models. Then $M_1 \preceq M_2$ iff for all modal box formulas $\varphi$, if $M_2 \models \varphi$ then $M_1 \models \varphi$.

**Proof:** Assume that all modal box formulas holding in $M_1$ are also valid for $M_1$, and construct a simulation between $M_1$ and $M_2$. Define $H$ by $(u,v) \in H$ iff for all modal box formulas $\varphi$, if $v \models \varphi$ then $u \models \varphi$. Then $(w_1,w_2) \in H$ by definition, and $(u,v) \in H$ implies $E(u) = E(v)$ since literals are modal box formulas. Assume $(u,v) \in H$ and $(u,v') \in I_2(R)$. We have to show that there is a $v'$ such that $(v,v') \in I_2(R)$ and for all modal box formulas $\varphi$, if $v' \not\models \varphi$ then $v' \not\models \varphi$. Assume for contradiction that for each $v'$ with $(v,v') \in I_2(R)$ there is a $\varphi_{v'}$ such that $u' \not\models \varphi_{v'}$ and $v' \models \varphi_{v'}$. Since $M_2$ 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 $u' \models [R] \bigvee \varphi_{v'}$, a contradiction to the assumption that $u' \not\models \varphi_{v'}$ for some $\varphi_{v'}$. \hfill \Box

We already mentioned that the above theorems can be used to reduce the complexity of model checking. To prove that $M_1 \models \varphi$, it can help to find an appropriate abstraction $M_2$, and to prove $M_1 \preceq M_2$ and $M_2 \models \varphi$. 

3.3. SIMULATION RELATIONS

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 $\mathcal{M}_1 = (U_1, I_1, w_1)$ and $\mathcal{M}_2 = (U_2, I_2, w_2)$, to check whether $\mathcal{M}_1 \preceq \mathcal{M}_2$ we define a sequence of relations $H^n, H^1, \ldots$ on $U_1 \times U_2$ as follows:

1. $(u, v) \in H^n$ iff for all $p \in \mathcal{P}$ it holds that $u \in I_1(p)$ iff $v \in I_2(p)$

2. $(u, v) \in H^{n+1}$ iff $(u, v) \in H^n$ and for all $R$ and $u' \in U_1$ such that $(u, u') \in I_1(R)$ there is a $v'$ with the property that $(v, v') \in I_2(R)$ 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, $\mathcal{M}_1 \preceq \mathcal{M}_2$ 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 $\{u' \mid (u, u') \in I(R)\}$,

```
|start:| $H^{\text{new}} := \{(u,v) \mid L_1(u) = L_2(v)\}$ |
|repeat:| $H^{\text{old}} := H^{\text{new}}$; $H^{\text{new}} := \emptyset$ |
|for all $(u,v) \in H^{\text{old}}$ do |
|add $:= \top$; for all $R \in \mathcal{R}$ do |
|if not $R(u) \subseteq ((R(u) \times R(v)) \cap H^{\text{old}})_1$ then add $:= \bot$ |
|if add then $H^{\text{new}} := H^{\text{new}} \cup \{(u,v)\}$ |
|until $H^{\text{new}} = H^{\text{old}}$ |
```

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 $\mathcal{M}_1 \cong \mathcal{M}_2$ 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 $\subseteq$ is isomorphism. For $\mathcal{M}_1 \preceq \mathcal{M}_2$, 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 \((U_1, I_1, w_1)\) and \((U_2, I_2, w_2)\) such that

1. \( w_1 \sim w_2 \),
2. If \( u \sim v \), then \( u \in I_1(p) \) iff \( v \in I_2(p) \)
3. If \( u \sim v \) and \((u, u') \in I_1(R)\), then there exists \( u' \) such that \((v, u') \in I_2(R)\) and \( u' \sim v' \).
4. If \( u \sim v \) and \((v, v') \in I_2(R)\), then there exists \( u' \) such that \((u, u') \in I_1(R)\) and \( u' \sim v' \).

Two Kripke-models \( M_1 \) and \( M_2 \) are bisimilar (denoted by \( M_1 \sim M_2 \)), if there exists a bisimulation between them. Figure 3.6 shows some examples for bisimilar models.

![Bisimilar models](image)

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.
3.4. BISIMULATIONS (P-MORPHISMS)

If $M_1 \sim M_2$, then $M_1 \preceq M_2$ and $M_2 \preceq M_1$; 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](image)

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 $M_1 \sim M_2$ and all multimodal formulas $\varphi$, we have $M_1 \models \varphi$ iff $M_2 \models \varphi$.

The proof is by induction on the structure of $\varphi$, 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 $\mu TL$. Given any formula $\varphi$ which is syntactically monotonic in $q$, and a natural number $n$, we define $\nu^n q \varphi \triangleq \top$, 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 $M \triangleq (U, I, w)$ be a finite model, where $|U| = n$. Then $M \models \nu q \varphi$ iff $M \models \nu^n q \varphi$

**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 $\varphi$ 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 \(\varphi\) is monoton 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, \((\nu^n q \varphi)^\mathcal{M} = \top^\mathcal{M} = U \supseteq (\nu^1 q \varphi)^\mathcal{M}\). Since \(\varphi^\mathcal{M}\) is monotonic (cf. Fact 2.9), \((\nu^1 q \varphi)^\mathcal{M} = \varphi^\mathcal{M}(\nu^1 q \varphi)^\mathcal{M}) \supseteq \varphi^\mathcal{M}((\nu^1 q \varphi)^\mathcal{M} = (\nu^2 q \varphi)^\mathcal{M}\). 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 \((U, I)\). 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, \((U, I, w) \not\models \nu q \varphi\) means that for all \(Q \subseteq U\) such that \(w \in Q\) there exists a \(v \in Q\) such that \((U, I', v) \not\models \varphi\), where \(I'(q) = Q\). Let \(Q = (\nu^n q \varphi)^\mathcal{M}\). If \(w \not\in Q\), then \((U, I, w) \not\models \nu^n q \varphi\) and we are done. If \(w \in Q\), then for some \(v\) it holds that \((U, I, v) \models \nu^n q \varphi\), and \((U, I', v) \not\models \varphi\), where \(I'(q) = (\nu^n q \varphi)^\mathcal{M}\). In other words, \((U, I', v) \not\models \varphi[q := \nu^n q \varphi]\), which means that \((U, I, v) \not\models \nu^{n+1} q \varphi\). Since \((U, I, v) \models (\nu^n q \varphi \rightarrow \nu^{n+1} q \varphi)\), this is a contradiction. \(\square\)

This lemma is important for model checking of \(\mu\text{TL}\) on finite Kripke models. Moreover, it allows to prove the following result.

**Theorem 3.15** Finite models are monotonic \(\mu\text{TL}-equivalent iff they are bisimilar.

**Proof:** Finite and monotonic \(\mu\text{TL}-equivalent models are also image finite and modal equivalent, since \(\mu\text{TL}-equivalence implies modal equivalence.

Hence as an immediate consequence of Theorem 3.13 such models are bisimilar. For the other direction, assume that \(\mathcal{M}_1 \models \varphi \land \mathcal{M}_2 \not\models \varphi\), where \(\varphi\) is a monotonic \(\mu\text{TL}\)-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 \(\varphi^n\) is \(\varphi\) where every sub-formula \(\nu q \psi\) is replaced by \(\nu^n q \psi\). Since \(\varphi^n\) is a multimodal formula, \(\mathcal{M}_1\) and \(\mathcal{M}_2\) are modally inequivalent and therefore not bisimilar. \(\square\)

**Corollary 3.16** Any two finite Kripke-models which can be distinguished by a monotonic \(\mu\text{TL}\)-formula can also be distinguished by a multimodal formula.

[BaCoG88] proved that if two finite models can be distinguished by a formula of the logic \(\text{CTL}^\ast\), then they can be distinguished by a \(\text{CTL}\) formula. Since every \(\text{CTL}^\ast\) formula has a monotonic \(\mu\text{TL}\) 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 \( \mathcal{M} \triangleq (U,I,w_0) \), and any equivalence relation \( \equiv \) on \( U \). Define the quotient of \( \mathcal{M} \) with respect to \( \equiv \) to be the model \( \mathcal{M}^{\equiv} \triangleq (U^{\equiv},I^{\equiv},w_0^{\equiv}) \), where \( U^{\equiv} \) is the set of equivalence classes of \( U \) with respect to \( \equiv \), \( w_0^{\equiv} \) is the equivalence class of \( w_0 \), \( w^\equiv \in I^{\equiv}(p) \) if \( w \in I(p) \), and \( (w^\equiv_1,w^\equiv_2) \in I^{\equiv}(R) \) if \( (w_1,w_2) \in I(R) \).

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. page 76) 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 \( L(u) = L(v) \). Thus if \( u \sim v^{\equiv} \) then \( u \in I(p) \) iff \( v^{\equiv} \in I^{\equiv}(p) \). Furthermore, if \( (u_1,u_2) \in I(R) \) and \( u_1 \sim v_1^{\equiv} \), then by definition \( (u_1^{\equiv},u_2^{\equiv}) \in I^{\equiv}(R) \) and \( u_1 \equiv v_1 \). Therefore, \( u_1^{\equiv} = v_1^{\equiv} \), i.e., \( (v_1^{\equiv},u_2^{\equiv}) \in I^{\equiv}(R) \). For the last condition, assume that \( (v_1^{\equiv},v_2^{\equiv}) \in I^{\equiv}(R) \) 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 \( (w_1,w_2) \in I(R) \). 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 \( w_2 \equiv w_2 \) such that \( (u_1,w_2) \in I(R) \). From \( u_2 \equiv w_2 \) and \( w_2 \equiv w_2 \) 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 \( \langle R \rangle P \triangleq \{ w \mid \exists w' \in P, (w, w') \in I(R) \} \). Given any partition of \( U \) into equivalence classes, call a component \( w \) **uniform**, if for all \( p \in P \) it holds that \( w \subseteq I(p) \) or \( w \cap I(p) = \emptyset \). That is, \( w \) is uniform if \( \mathcal{L}(w_1) = \mathcal{L}(w_2) \) for all \( w_1, w_2 \in w \). A component \( w \) is called **stable with respect to** \( P \), if for all \( R \) either \( w \subseteq \langle R \rangle P \) or \( w \cap \langle R \rangle P = \emptyset \). 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 automorphism.

**Proof:** First, we show that any stable partition is an automorphism. Trivially, \( w_0 \equiv w_0 \). Since \( u \equiv v \) implies \( \mathcal{L}(u) = \mathcal{L}(v) \). If \( (u, u') \in I(R) \), then \( u \subseteq \langle R \rangle u' \equiv v \equiv v' \), because \( u \) is stable with respect to \( v' \).

In other words, \( u \subseteq \{ v \mid \exists u' \equiv v' \in I(R) \} \). Therefore, if \( u \equiv v \), then there is a \( v' \equiv u' \) such that \( (v, v') \in I(R) \). The symmetric condition is proved symmetrically. Vice versa, every automorphism defines a stable partition: To show that \( u \equiv v \) is stable with respect to \( v' \), assume that \( u_1 \equiv u_2 \in u \). Since \( \equiv \) is a bisimulation, for every \( (u_1, u'_1) \in I(R) \) and \( u'_2 \in v' \), there must be a \( u'_2 \equiv u'_1 \in v' \) such that \( (u_2, u'_2) \in I(R) \). Therefore, \( u \equiv \langle R \rangle v' \) or \( u \cap \langle R \rangle v' = \emptyset \). If \( \equiv \) is the coarsest stable partition, then for any automorphism \( \sim \) 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 \( \equiv \). \( \square \)

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 \( w \) and a proposition \( p \in P \);
  - Split \( w \cap I(p) \) and \( w \backslash I(p) \)
  - Choose components \( w_1 \) and \( w_2 \), and a relation \( R \in \mathcal{R} \);
  - Split \( w_1 \cap R \) and \( w_2 \backslash \langle R \rangle w_1 \)
- 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 $p \in \mathcal{P}$ and $R \in \mathcal{R}$. Then, a $w^= \in C$ is split into $w_1^= \in F$ and $w_2^= \triangleq w^= \setminus w_1^=$. Any $w^= \in F$ is split into four parts. First, it is split with respect to $\langle R \rangle w_1$, and then again with respect to $\langle R \rangle w_2$.

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

Figure 3.8: Page-Tarjan algorithm for bisimulation minimization

In the last split, since $w^= \setminus w_1^=$ is stable with respect to $C$, either $w^= \subseteq \langle R \rangle w$ or $w^= \cap \langle R \rangle w = \emptyset$ 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 $w^=$ 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 \( M = (\Sigma^\text{in}, \Sigma^\text{out}, T) \) is an admissible environment for \( M_C \), if \( \Sigma_C \)
and \( \Sigma_E \) are compatible, \( \Sigma^\text{out}_E \supseteq \Sigma^\text{in}_C \), and \( \Sigma^\text{in}_E \subseteq \Sigma^\text{out}_C \). 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 \( M = (\Sigma^\text{in}, \Sigma^\text{out}, T) \), then \( M \) without \( w \) \((M \setminus w)\) is the I/O-module
\((\Sigma^\text{in}, \Sigma^\text{out}, T)\), where \( \Sigma^\text{in} = \Sigma^\text{in} - \{w\} \), \( \Sigma^\text{out} = \Sigma^\text{out} - \{w\} \) and \( T = T \setminus \{w\} \). 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 \( y \models x \). Furthermore, for \( \mathcal{M} = \{M_1, \ldots, M_n\} \), we say that \( \mathcal{M} \models x \) if \( M_k \models x \) for all \( k \leq n \).

Assume a system consisting of a set \( \mathcal{M}_C \triangleq \{M_1, \ldots, M_n\} \) of
I/O-modules and a specification given as a single I/O-module \( M_S \triangleq
(\Sigma^\text{in}_S, \Sigma^\text{out}_S, T_S) \) such that \( \Sigma^\text{in}_S = \bigcup \Sigma^\text{in}_k - \bigcup \Sigma^\text{out}_k \) and \( \Sigma^\text{out}_S \subseteq \bigcup \Sigma^\text{out}_k \). 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
\( M_E \triangleq (\Sigma^\text{out}_S, \Sigma^\text{in}_S, T_E) \), whenever \( \{M_S, M_E\} \) is failure-free, also \( \mathcal{M}_C \cup \{M_E\} \)
is failure-free.

In other words, the system \( M_C \) 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 implementa-
tion 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 \( \varphi \) holds for a given model or theory?

Answer: encode the model or theory as a set of assumptions \( \Phi \) and check whether the formula in question follows from \( \Phi \).

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:

- \( \Phi \models \varphi \) if from \( \Phi \) follows \( \varphi \),
  i.e. if any model in which all formulas from \( \Phi \) are valid also validates \( \varphi \),

- \( \Phi \vdash \varphi \) if \( \Phi \) proves \( \varphi \),
  i.e. if there is a proof of \( \varphi \) which uses only assumptions from \( \Phi \), and

- \( \Phi \rightarrow \varphi \) if \( \Phi \) implies \( \varphi \),
  i.e. this is a statement of the object language which is only defined if \( \Phi \) is a single formula. To be liberal, we can identify a finite set of formulas \( \{ \varphi_1, \ldots, \varphi_n \} \) with the conjunction \( (\varphi_1 \land \ldots \land \varphi_n) \).
Note that \( \Phi \models \varphi \) is different from \( \mathcal{M} \models \varphi \). The notations \( \models \varphi \) and \( \vdash \varphi \) are short for \( \{\} \models \varphi \) 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: \textit{local validity} \( (U, I, w_0) \models \varphi \), where the evaluation point is given, and \textit{universal validity} \( (U, I) \models \varphi \). 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 \( \Phi \) and \( \varphi \),

- if \( \Phi \vdash \varphi \), then \( \Phi \models \varphi \) (Correctness), and
- if \( \Phi \models \varphi \), 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. COMPLETENESS

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) \( p, (p \rightarrow q) \vdash q \)
(N) \( q \vdash [a] q \)
(K) \( \vdash [a] p \land [a] (p \rightarrow q) \rightarrow [a] q \)
(U) \( \vdash \langle a \rangle q \rightarrow [a] q \)
(L) \( \vdash \langle a \rangle q \rightarrow \langle b \rangle \neg q \) for \( a \neq b \)

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

As an example, let us assume \( (p \rightarrow q) \) and derive some consequences:

1. \( p \rightarrow q \) (ass)
2. \([a] (p \rightarrow q) \) (1, N)
3. \([a] (p \rightarrow q) \rightarrow ([a] p \rightarrow [a] q) \) (K, taut)
4. \([a] p \rightarrow [a] q \) (2,3,MP)
5. \( \neg q \rightarrow \neg p \) (1)
6. \([a] \neg q \rightarrow [a] \neg p \) (5)
7. \( \langle a \rangle p \rightarrow \langle a \rangle q \) (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) \( p \rightarrow q \vdash \varphi(p) \rightarrow \varphi(q) \).

(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, \( [a] q, \langle a \rangle q, q \land [a] (q \lor \langle a \rangle q) \) are all positive in \( q \).

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

1. \( \langle a \rangle q \rightarrow [b] \neg q \) (L)
2. \( (a) \models \neg \perp \rightarrow \neg [b] \rightarrow \neg \perp \) \hspace{1cm} (1)
3. \([a] \perp \lor [b] \perp \) \hspace{1cm} (2, repl)

And the other direction:

4. \( \perp \rightarrow \neg q \) \hspace{1cm} (t aut)
5. \([a] \perp \rightarrow \neg [a] \neg q \) \hspace{1cm} (4, mon)
6. \( (a) q \rightarrow \neg [a] \perp \) \hspace{1cm} (5)
7. \( \neg [a] \perp \rightarrow \neg [b] \perp \) \hspace{1cm} (3)
8. \([b] \perp \rightarrow \neg [b] \neg q \) \hspace{1cm} (4, mon)
9. \( (a) q \rightarrow \neg [b] \neg q \) \hspace{1cm} (6,7,8)

As a more practical example, let us derive from the assumptions

\[
\text{set} \rightarrow \langle V \rangle \neg \text{set} \quad \text{and} \\
\neg \text{set} \rightarrow \langle P \rangle \text{set} \lor \langle V \rangle \neg \text{set} \quad \text{the property} \\
[P][P] \perp.
\]

The assumptions can be seen as describing the actions of a semaphore with two states, \( \text{set} \) and \( \neg \text{set} \), 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. \( \perp \rightarrow [P] \perp \) \hspace{1cm} (t aut)
2. \([P] \perp \rightarrow [P][P] \perp \) \hspace{1cm} (1,mon)
3. \( \langle V \rangle \top \rightarrow [P] \perp \) \hspace{1cm} (L)
4. \( \langle V \rangle \neg \text{set} \rightarrow \langle V \rangle \top \) \hspace{1cm} (mon)
5. \( \langle V \rangle \neg \text{set} \rightarrow [P] \perp \) \hspace{1cm} (3,4)
6. \( \langle V \rangle \neg \text{set} \rightarrow [P][P] \perp \) \hspace{1cm} (2,5)
7. \( \langle P \rangle \langle V \rangle \neg \text{set} \rightarrow \langle P \rangle [P] \perp \) \hspace{1cm} (5,mon)
8. \( \text{set} \rightarrow \langle V \rangle \neg \text{set} \) \hspace{1cm} (ass)
9. \( \text{set} \rightarrow [P][P] \perp \) \hspace{1cm} (6,8)
10. \( \langle P \rangle \text{set} \rightarrow \langle P \rangle \langle V \rangle \neg \text{set} \) \hspace{1cm} (8,mon)
11. \( \langle P \rangle \text{set} \rightarrow \langle P \rangle [P] \perp \) \hspace{1cm} (7,10)
12. \( \langle P \rangle [P] \perp \rightarrow [P][P] \perp \) \hspace{1cm} (U)
13. \( \langle P \rangle \text{set} \rightarrow [P][P] \perp \) \hspace{1cm} (11,12)
14. \( \langle P \rangle \text{set} \lor \langle V \rangle \neg \text{set} \rightarrow [P][P] \perp \) \hspace{1cm} (6,8)
15. \( \neg \text{set} \rightarrow \langle P \rangle \text{set} \lor \langle V \rangle \neg \text{set} \) \hspace{1cm} (ass)
16. \( \neg \text{set} \rightarrow [P][P] \perp \) \hspace{1cm} (14,15)
17. \( \langle \text{set} \lor \neg \text{set} \rangle \rightarrow [P][P] \perp \) \hspace{1cm} (9,16)
18. \([P][P] \perp \) \hspace{1cm} (17)
4.1. Completeness

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 ($p \rightarrow q$) 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 $\Psi$ of formulas is inconsistent with $\Phi$, if there is a finite subset $\{\psi_1, ..., \psi_n\} \subseteq \Psi$ such that $\Phi \vdash (\neg \psi_1 \lor \ldots \lor \neg \psi_n)$. To prove completeness, we have to show

(*) Every formula consistent with $\Phi$ is satisfiable in a model validating $\Phi$.

For, if $\Phi \not\models \varphi$, then no model validating $\Phi$ satisfies $\{\neg \varphi\}$; therefore with (*) it follows that $\{\neg \varphi\}$ is inconsistent with $\Phi$, hence $\Phi \vdash \varphi$. (Without loss of generality, we can assume here $\Phi$ to be consistent with itself, or else $\Phi \not\models \varphi$ 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 $\varphi$ which is consistent with $\Phi$ 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 $\psi$ according to a fixed enumeration add either $\psi$ or $\neg \psi$ to $w$, whichever is consistent with the set constructed so far.

The canonical model for $\Phi$ is $(U, I, w)$, where
• $U$ is the set of maximal consistent sets which include $\Phi$,

• $I(a) \triangleq \{(w_\emptyset, w_1) \mid q \in w_1 \rightarrow \langle a \rangle q \in w_\emptyset\}$, and

• $I(p) \triangleq \{w_\emptyset \mid p \in w_\emptyset\}$, and

• $w$ is any element from $U$ such that $\varphi \in w$.

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

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

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

In the inductive step for this lemma, we have to show that $(a) \varphi \in w_\emptyset$ iff $(U, I, w_\emptyset) \models (a) \varphi$. The ‘if’ direction being a direct consequence of definition and induction hypothesis, assume that $(a) \varphi \in w_\emptyset$. We have to find a maximal consistent set $w_1$ such that $(w_\emptyset, w_1) \in I(a)$ and $\varphi \in w_1$. Since $\models (\langle a \rangle \varphi \land \langle a \rangle \psi) \rightarrow (\langle a \rangle (\varphi \land \psi))$, 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 $(a) \psi$ must be in $w_\emptyset$ (otherwise, a contradiction could be derived). Therefore $(w_\emptyset, w_1) \in I(a)$. Since $\varphi \in w_1$, the induction hypothesis gives $(U, I, w_1) \models \varphi$. Together with $(w_\emptyset, w_1) \in I(a)$ we have $(U, I, w_\emptyset) \models (a) \varphi$.

Since for the canonical model $(U, I, w)$ it holds that $\Phi \subseteq w$ and $\varphi \in w$, we proved that $(U, I, w) \models \Phi$ and $(U, I, w) \models \varphi$. 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 $R$ for the transitive closure; the extensions for until- and since operators being almost straightforward extensions of the basic ideas.

The relation between $X$ and $(a)$ is fixed by the following axiom:
4.1. COMPLETENESS

\[(\text{nec}) \quad \vdash (a) \, q \rightarrow X \, q \text{ for all } a \in \mathcal{R}\]

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

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

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 $\Phi$ it holds that $\Phi \models \varphi$ implies $\Phi \vdash \varphi$.

In first order logic, the deduction theorem allows to discard any finite set of assumptions: $\psi \models \varphi$ if $\models \forall \psi \rightarrow \varphi$, where $\forall \psi$ is the universal closure of $\psi$. In temporal logic, similar deduction theorem holds:

\[\psi \models \varphi \iff \models \psi \land \Box \psi \rightarrow \varphi\]

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

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

\[(\text{Rec}) \quad \vdash X \,(q \lor F^+ \, p) \rightarrow F^+ \, q\]
\[(\text{Ind}) \quad X \,(p \lor q) \rightarrow q \vdash F^+ \, p \rightarrow q\]

Dually, this can be written as

\[(\text{Rec}) \quad \vdash \Box \, q \rightarrow \neg X \, \neg(q \land \Box q)\]
\[(\text{Ind}) \quad q \rightarrow \neg X \, \neg(p \land q) \vdash q \rightarrow \Box p\]

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 $a \in \mathcal{R}$. \textbf{(Rec)} is the recursion axiom which can be used to unfold a Box-operator:

\[\Box \varphi \rightarrow \neg X \, \neg(\varphi \land \neg X \, \neg(\varphi \land \neg X \, \neg(\varphi \land \ldots \))).\]

\textbf{(Ind)} is the induction axiom which can be used to derive a property $\Box \varphi$ from an invariant $\psi$, i.e. from a formula $\psi$ 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 $\varphi$ for which we have to construct a model. Define the notion of extended sub-formula of $\varphi$ (sometimes also called Fisher-Ladner closure) as follows:

- $\varphi$ is an extended sub-formula of $\varphi$,
- $\neg \varphi$ is an extended sub-formula of $\varphi$, if $\varphi$ is not of form $\neg \varphi$,
- $\varphi_1$ and $\varphi_2$ are extended sub-formulas of $(\varphi_1 \rightarrow \varphi_2)$, (thus $\varphi$ is an extended sub-formula of $\neg \varphi$)
- $\varphi$ is an extended sub-formula of $X \varphi$,
- $X \varphi$ is an extended sub-formula of $F^+ \varphi$, and
- $XF^+ \varphi$ is an extended sub-formula of $F^+ \varphi$

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 $\psi$ of $\varphi$, either $\psi$ or $\neg \psi$ is in the finitely maximal consistent set.

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

$$F^+ \varphi \in w_0 \iff (U, I, w_0) \models F^+ \varphi.$$ 

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 $\neg X \Box \varphi$ to be in $w_0$, and hence $\Box \varphi$ is in every $w_1$ with $(w_0, w_1) \in I(a)$.

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

- $\vdash w_0 \rightarrow \Psi$, since $w_0$ is one of the $\Psi_i$ of which $\Psi$ is composed. Furthermore,
- $\vdash \Psi \rightarrow \neg \varphi$, since $\neg \varphi$ was assumed to hold in the whole component. Finally,
4.1. **Completeness**

- \( \vdash \Psi \rightarrow \neg X \neg \Psi \), since \( \Psi \) consists of all finitely maximal consistent sets in this component.

Putting these parts together, we have a contradiction, since the induction axiom gives \( \vdash w_0 \rightarrow \square \neg \varphi \), but \( F \varphi \in w_0 \).

**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, Kro87], where the setting in [LPZ85] is closest to ours. Let \( X^{<} \varphi \equiv (\perp U^{<} \varphi) \) and \( \neg X^{<} \varphi \equiv \neg X^{<} \neg \varphi \) be strong and weak previous operators. The following axioms are adequate:

\[
\begin{align*}
(N) & \quad q \vdash (X q \land \neg X q) \\
(K) & \quad \vdash (X p \rightarrow q) \rightarrow (X p \rightarrow X q) \\
& \quad \vdash (X^\neg (p \rightarrow q) \rightarrow (X^\neg p \rightarrow X^\neg q)) \\
(B') & \quad \vdash (p \rightarrow (XX^\neg p \land X^\neg X p)) \\
(\text{Init}) & \quad \vdash (X \perp \rightarrow F X^\neg \perp) \\
(U) & \quad \vdash (X p \rightarrow X p) \\
& \quad \vdash (X^\neg p \rightarrow X^\neg p) \\
(\text{Rec} U^{<}) & \quad \vdash (X (q_1 \lor q_1 \land (q_1 U^{<} q_2)) \rightarrow (q_1 U^{<} q_2)) \\
(\text{Rec} U^\neg) & \quad \vdash (X^\neg (q_1 \lor q_1 \land (q_1 U^\neg q_2)) \rightarrow (q_1 U^\neg q_2)) \\
(\text{Ind} U^{<}) & \quad \vdash (X (q_1 \lor q_1 \land p) \rightarrow p) \vdash ((q_1 U^{<} q_2) \rightarrow p) \\
(\text{Ind} U^\neg) & \quad \vdash (X^\neg (q_1 \lor q_1 \land p) \rightarrow p) \vdash ((q_1 U^\neg q_2) \rightarrow p)
\end{align*}
\]

Axiom \((B')\) relates future and past operators. It corresponds to the fact that the relation described by \( X^{<} \) is the converse of that described by \( X \). Axiom \((\text{Init})\) can be translated as \( \forall x \exists y \leq x \exists z (x \prec y) \); 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 \( I(\prec) \) in the finite canonical model (where points are finitely maximal consistent sets). Assume that \( p \in ESF(\varphi) \), but \( X p \notin ESF(\varphi) \). For any finitely maximal consistent set \( w \) such that neither \( \vdash (\dot{w} \rightarrow X p) \) nor \( \vdash (\dot{w} \rightarrow X \neg p) \), there will be \( w_1 \) and \( w_2 \) with \( w \prec w_1 \), \( w \prec w_2 \), and \( p \in w_1 \), \( p \notin w_2 \). 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 \((\varphi_1 U^+ \varphi_2) \in w \) 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 \((\text{RecU}^+)\): Similar to above, if \( w \in W \) and \((\varphi_1 U^+ \varphi_2) \in w \) there exists a \( w' > w \) such that \( \varphi_2 \in w' \). Since \( W \) is terminal, it follows that \( w' \notin W \). A path \( \sigma \) through the finite canonical model for \( \varphi \) is called accepting, if

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

Recall that \( \inf(\sigma) \) is the set of points occurring infinitely often in \( \sigma \). Any accepting path \( \sigma \) constitutes a natural model for \( \varphi \). If \( \sigma \) infinitely often traverses all points in a nontrivial terminal SCC, then for all \( w_i \in \sigma \) such that \((\varphi_1 U^+ \varphi_2) \in w_i \) there will be some \( w_{i+n} \in \sigma \) such that \( \varphi_2 \in w_{i+n} \). A similar statement holds, if \( \sigma \) is finite. Thus, we have to show that such a path exists. We use the finite canonical model for \( \varphi' \equiv \varphi \land (\neg X^- \psi \lor F \neg \neg X^- \psi) \). In this model, there exists a point \( w_{\text{init}} \leq w_0 \) such that \( \neg X^- \psi \in w_{\text{init}} \). Since \( w_{\text{init}} \) is consistent, there is no \( X^- \psi \) in \( w_{\text{init}} \). From axiom (B') we can prove that there is no \( w \) such that \( w \prec w_{\text{init}} \). Therefore, \( w_{\text{init}} \) is an initial point. For any \((\varphi_1 U^+ \varphi_2) \in w_0 \) 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 \( w_{\text{term}} \) be a point in any terminal SCC reachable from \( w_0 \). For all \((\varphi_1 U^+ \varphi_2) \in w_0 \) we can prove: either there exists a point \( w_0 < w \leq w_{\text{term}} \) such that \( \varphi_2 \in w \), or \((\varphi_1 U^+ \varphi_2) \in w_{\text{term}} \). Since \( W \) is self-fulfilling, any path from \( w_{\text{init}} \) 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 \( \mu \)-calculus

We just briefly indicate how the above axioms can be extended for \( \mu \text{TL} \).

\[
(\text{Rec}\nu) \quad \vdash \nu q \varphi(q) \rightarrow \varphi(\nu q \varphi(q))
\]

\[
(\text{Ind}\nu) \quad q \rightarrow \varphi(q) \vdash q \rightarrow \nu p \varphi(p)
\]
The recursion and induction axiom can be obtained as special cases of these very general axioms by defining $\Box p \triangleq \nu q \land [a_i] (p \land q)$.

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

The problem of completeness of these axioms for all $\mu TL$ 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 $\Phi$ of assumptions, and a formula $\varphi$. We want to decide whether $\Phi \models \varphi$, which by completeness is the same as $\Phi \models \varphi$. Now $\Phi \models \varphi$ 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 $(a)$-operators) and of temporal logic (with operators $X$ and $F^\ast$). 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 ($\Phi \models \varphi \Rightarrow \Phi \models \varphi$ for finite $\Phi$), 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:

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

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

For any finite $\Phi$, 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 $\Phi$, 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

\[ \bot \notin w, \quad \text{and} \]

\[ (\psi_1 \to \psi_2) \in ESF(\varphi) \text{ then } (\psi_1 \to \psi_2) \in w \text{ iff } \psi_1 \in w \text{ implies } \psi_2 \in w. \]

That is, if $(\psi_1 \to \psi_2) \in w$ then $\neg \psi_1 \notin w$ 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 $(\psi_1 \land \psi_2) \in ESF(\varphi)$,
  
  \[ (\psi_1 \land \psi_2) \in w \text{ iff both } \psi_1 \in w \text{ and } \psi_2 \in w, \]

• for any $(\psi_1 \lor \psi_2) \in ESF(\varphi)$,
  
  \[ (\psi_1 \lor \psi_2) \in w \text{ iff } \psi_1 \in w \text{ or } \psi_2 \in w. \]

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 $I(p) \triangleq \text{true}$ iff $p \in w$.

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

The first, ‘local’ algorithm, is tableau-based. Start with the set $w_1, \ldots, w_n$ of all propositionally maximal consistent sets which include $\Phi$ and try to systematically extend one of these to a model. Given a propositionally maximal consistent set $w$, 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
4.2. DECIDABILITY

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
$w_i^1 \triangleq \{\psi \mid \langle a \rangle \psi \in w_i\} \cup \{\neg \psi \mid \neg \langle a \rangle \psi \in w_i\}$ and $w_i^2 \triangleq w_i^1 \cup \Phi$ (We are
considering linear accessibility relations and universal consequence!). There
are finitely many propositionally maximal consistent extensions $w_{i,1}, \ldots, w_{i,n}$
of $w_i^n$.

If $w_i^n$ 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 an 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 $\Phi$ be the set of formulas whose satisfiability we have to check. $\Gamma$ is
any set of formulas:

\[
\begin{align*}
(\rightarrow) & \quad \frac{\Gamma, (\psi_1 \rightarrow \psi_2)}{\Gamma, \neg \psi_1, \psi_2} \\
(\bot_1) & \quad \frac{\Gamma, \psi, \neg \psi}{*} \\
(\bot_2) & \quad \frac{\Gamma, \bot}{*} \\
(\neg) & \quad \frac{\Gamma, \neg \psi}{\Gamma, \psi}
\end{align*}
\]

\[
\begin{align*}
(\langle \rangle) & \quad \frac{\Gamma, \langle \varphi_1, \ldots, \varphi_n, \neg \langle a \rangle \psi_1, \ldots, \neg \langle a \rangle \psi_m \rangle}{\Phi, \varphi_1, \ldots, \varphi_n, \neg \psi_1, \ldots, \neg \psi_m}
\end{align*}
\]

Derived rules:

\[
\begin{align*}
(\lor) & \quad \frac{\Gamma, (\psi_1 \lor \psi_2)}{\Gamma, \psi_1, \Gamma, \psi_2} \\
(\land) & \quad \frac{\Gamma, (\psi_1 \land \psi_2)}{\Gamma, \psi_1, \psi_2} \\
(\neg \langle a \rangle) & \quad \frac{\Gamma, \neg \langle a \rangle \psi}{\Gamma, [a] \neg \psi} \\
(\text{U}) & \quad \frac{\Gamma, \langle a \rangle \psi_1, [a] \psi_2}{\Gamma, \langle a \rangle \psi_1}
\end{align*}
\]
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 \( \psi_2 \neq \perp \).
- Rules \( \langle \psi \rangle \) and \( \langle \langle \rangle \rangle \) can only applied if no other rule is applicable.
- Rule \( \langle \psi \rangle \) can only be applied if no other \( \langle a \rangle \varphi \) or \( \neg \langle a \rangle \psi \) is in \( \Gamma \).
- Rule \( \langle \langle \rangle \rangle \) can only be applied if no \( \langle a \rangle \varphi \) is in \( \Gamma \).

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

- The root of the tableau is \( \Phi \), and
- The successors of each node are constructed according to some tableau rule.

A leaf is called closed, if it consists of the symbol \( \ast \). 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 \( \langle \langle \rangle \rangle \) regenerates the root \( \Phi \), the new leaf is open. Also, any empty node constructed by rule \( \langle \langle \rangle \rangle \) 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 \( \Phi \) has any successful tableau, then every completed tableau for \( \Phi \) is successful.

### 4.2.3 Adequacy of the Modal Tableau Procedure

\( \Phi \) is satisfiable iff \( \Phi \) has a successful tableau.

The proof of this statement is more or less straightforward: Assume \( \Phi \) is satisfiable in a natural model \( \mathcal{M} \models ((w_0, w_1, w_2, \ldots), I, w_0) \), and show that there is a tableau for \( \Phi \) with an open leaf. Equivalently, assume that any tableau for \( \Phi \) 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 \( w(n_0) \models w_0 \). 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 \( w(n_{i+1}) \models w_j \), 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 $w(n_{i+1}) \equiv w_j$. If $n_i$ has a successor obtained by rule $(\lnot)$, then $w(n_{i+1}) \equiv w_{j+1}$. 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 tableau contains open leafs, then the unfolding contains infinite paths. In the unfolding, call any node whose successor is constructed by rule $(\lnot)$ or $(\lnot)$ 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 $[P][P] \perp$ from the assumptions

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

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

$$\Phi, \langle P \rangle \top \\
\neg s, \langle P \rangle s, \langle P \rangle \top \\
\Phi, s, \langle P \rangle \top \\
s, \langle V \rangle \neg s, \langle P \rangle \top \\
*$$

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 $[P][P] \perp$ follows from the assumptions $\Phi$.

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 $(w_0, w_1) \in I(a)$ 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 $(w_0, w_1) \in I(a)$ with $\psi \in w_1$. The given formula set $\Phi$ 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 $\langle a \rangle (p \land \langle b \rangle p)$.

![Diagram](image)

(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 $w_0, \ldots, w_n$ of all propositionally maximal consistent sets which include
\[ 4.2. \textit{Decidability} \quad 103 \]

\( \Phi \). 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 \( \psi_1, \ldots, \psi_n \), 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 \( (\Psi_1, \ldots, \Psi_n) \) of propositionally maximal consistent sets satisfying

If \( \psi_j = \langle a \rangle \psi \), then \( \Psi_j \) 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 \( (\Psi_1, \ldots, \Psi_n) \in S(w_i) \), all \( \Psi_i \) 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:

\[
\frac{
\rightarrow_1}{\Gamma \vdash (\psi_1 \rightarrow \psi_2)} \quad \frac{
\rightarrow_2}{\Gamma \vdash (\psi_1 \rightarrow \psi_2)}
\]

\[
\frac{
(\Theta \vdash \psi) \quad \Phi \vdash \psi_i \quad \ldots \quad \Phi \vdash \psi_n
}{\Gamma, \Phi \vdash \Psi}
\]

where \( \Psi = \{ \langle a_1 \rangle \psi_1, \ldots, \langle a_n \rangle \psi_n \} \), no \( \langle a \rangle \psi \) is in \( \Gamma \), 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 stabilization is reached. Here, bad arcs are pairs \( (w_0, w_1) \in I(a) \) 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 \( (w_0, w_1) \in I(a) \) 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 $X\varphi$ should be regarded as extended subformula of the formula $F^+\varphi$. Furthermore, we define also $XF^+\varphi$ to be an extended subformula of $F^+\varphi$. We have to be a bit careful to avoid a nonterminating recursion in the algorithmic reformulation of the recursive definition:

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

Now, this definition of extended subformula guarantees that for any $F^+\psi$ in $w_i$, all possible successors will contain $F^+\psi$ or $\neg F^+\psi$ 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 $F^+\psi$, but no formula $X\psi'$, we can discard it, because the eventuality $F^+\psi$ is not fulfilled. Also, if some candidate successor $w'_{i}$ of node $w_i$ with $F^+\psi \in w_i$ contains $\neg F^+\psi$ and $\neg \psi$, we can discard $w'_{i}$. But, if $w'_{i}$ contains $F^+\psi$ again, this unsatisfied eventualuity could be propagated, resulting in a cycle where the fulfilment of $F^+\varphi$ is infinitely delayed. The solution is to require that a backward loop only can be regarded as open, if for any $F^+\psi$ 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:
4.2. DECI-DABILITY

\[
\frac{\Gamma, \exists \alpha \, \psi}{\Gamma, \exists \alpha \, \psi} \\
\frac{\Gamma, \forall \alpha \, \psi \quad \Gamma, \forall \alpha \, \psi}{\Gamma, \exists \alpha \, \psi}
\]

(\neg X)

\[
\frac{\Gamma, \neg X \psi}{\Gamma, \neg X \psi} \\
\frac{\Gamma, \neg X \psi}{\Gamma, \neg X \psi}
\]

(\neg F^+)

\[
\frac{\Gamma, F^+ \psi}{\Gamma, X (\psi \lor F^+ \psi)} \\
\frac{\Gamma, \neg F^+ \psi}{\Gamma, X (\psi \lor F^+ \psi)}
\]

These rules are based on the unfolding of the X- and F^+-operators:

- X \psi \leftrightarrow \bigvee_{\alpha} \langle \alpha \rangle \psi
- F^+ \psi \leftrightarrow X (\psi \lor F^+ \psi)

Additionally we have to require that a leaf is only called open, if for all formulas F^+ \psi occurring in it, the formula \psi 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 F^+ \bot is unsatisfiable:

\[
\frac{F^+ \bot}{X (\bot \lor F^+ \bot)} \\
\frac{\langle \alpha \rangle (\bot \lor F^+ \bot)}{\bot \lor F^+ \bot} \\
\frac{\bot \lor F^+ \bot \quad \ldots \quad \bot \lor F^+ \bot}{* \quad * \quad *}
\]

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

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

- (\psi_2 \ U^+ \psi_1) \leftrightarrow X (\psi_1 \lor \psi_2 \land (\psi_2 \ U^+ \psi_1))

There is a close connection between the tableau decision procedure and \omega-automata: The pre-states in the tableau can be seen as states of a generalized Büchi-automaton. The set of open leaves forms the acceptance condition, and the recurrence condition is given as follows: For every subformula \langle \psi_2 \ U^+ \psi_1 \rangle, either it is infinitely often not contained in the accepting run, or \psi_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 $\text{qTL}$ and $\mu\text{TL}$ on natural models. The relational $\mu$-calculus, however, can categorically define addition and multiplication of natural numbers; therefore, 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 $\{\text{hi, lo, er, put, get, eq, } \langle X \rangle, \langle F \rangle, \langle M \rangle\}$. 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 $X$:

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

Using these relations, we fix the propositions such that
- the number of $M^t$-successors in any world labelled $\text{eq}$ is the value of the counter,
4.3. INCOMPLETENESS RESULTS

— 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 exe-
cuted next.

The relevant formulas are:

- put increases the length of the counter by one:
  \((\text{put} \land [M] \text{false} \rightarrow \langle X \rangle [M] [M] \text{false})\)

- get decreases the length of the counter by one:
  \(\langle [M] (\text{get} \land [M] \text{false}) \rightarrow \langle X \rangle [M] \text{false} \rangle\)

- Every world is exactly one of \{put, get\} and \{hi, lo, er\}:
  \((\text{put} \oplus \text{get}) \land (hi \oplus lo \oplus er) \quad (\ominus \text{denoting exclusive disjunction})\)

- All worlds reachable by \(M^*\) have the same marking:
  \((P \rightarrow [M] P) \text{ for } P \in \{\text{put, get, hi, lo, er}\}\)

- eq propagates only in one dimension:
  \((\text{eq} \rightarrow [X] \text{eq}), \quad [M] \neg \text{eq}\)

- Transitions:
  \((lo \land \text{put} \rightarrow [X] \text{hi}), \quad (hi \land \text{get} \rightarrow [X] \text{lo}),\)
  \((lo \land \text{get} \rightarrow [X] \text{er}), \quad (hi \land \text{put} \rightarrow [X] \text{er}),\)
  \((er \land \text{get} \rightarrow [X] \text{er}), \quad (er \land \text{put} \rightarrow [X] \text{false})\)

For a conditional transition like “from er go to lo if counter is zero” we
could use the sentence \((er \land \text{eq} \land [M] \text{false} \rightarrow [X] \text{lo})\). For a multiple
counter machine, we can use a similar encoding with several memory access
functions \((M_i)\). 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
\((lo \land \text{eq} \land [M] \text{false} \land [F](\langle F \rangle hi)\) 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 \(\Sigma^1_1\)-complete, therefore also the problem whether any sentence
follows from a set of universally quantified multimodal formulas is \(\Sigma^1_1\)-hard.
Recall that such formulas are monadic \(\Pi^1_1\)-properties, so in this case the
problem is in \(\Sigma^1_1\) 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 correctly 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.

```
1 2
3 4 5
6 7 8

8 7 6
5 4 3
2 1
```

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 (rrddlluurrdlluurrdlluurrd) 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], ...
5.1. A COMBINATORIAL GAME

\section*{5.1. A COMBINATORIAL GAME}

\begin{figure}[h]
\begin{center}
\begin{verbatim}
MODULE main
DEFINE h := 3; v := 3;
VAR move: u,d,l,r;
    hpos: array 0..(h*\text{v}-1) of 1..h;
    vpos: array 0..(\text{h*v}-1) of 1..v;
ASSIGN
next(vpos[0]) := case
    (move=u) & !(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=u) & !(vpos[i]=1) & hpos[i]=hpos[0] & vpos[i]=vpos[0]+1 |
    (move=r) & !(vpos[i]=v) & hpos[i]=hpos[0] & vpos[i]=vpos[0]-1 : vpos[0];
1: vpos[i]; esac;
next(hpos[i]) := case
    (move=u) & !(hpos[i]=1) & vpos[i]=vpos[0] & hpos[i]=hpos[0]-1 |
    (move=d) & !(hpos[i]=h) & vpos[i]=vpos[0] & hpos[i]=hpos[0]+1 : hpos[0];
1: hpos[i]; esac;
init(hpos[0]) := i \div v; init(vpos[0]) := i \mod v;
DEFINE goal := (hpos[i] = 3 - (i \div v) & vpos[i] = 3 - (i \mod v))
SPEC !EF goal
\end{verbatim}
\end{center}
\caption{SMV Code for Loyds Puzzle}
\end{figure}

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](image)

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.
5.2. A SEQUENTIAL CIRCUIT

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 characteristic function. If both inputs are low, the bistable keeps its state.

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Q</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

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:

\[ A \ \mathcal{G}'(mc \land pc \rightarrow (bus[i] \leftrightarrow A((oc \rightarrow A \ X \ bus[i]) U^+ ic))) \]

\[ A \ \mathcal{G}'(\neg mc \land sc \rightarrow (Q[i] \leftrightarrow A(Q[i-1] U^+ ic))) \]
<table>
<thead>
<tr>
<th>MODULE main</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR q, bus: array 1..n of boolean; -- n SR-latches, n databits</td>
</tr>
<tr>
<td>inp, mc, pc, sc, oc: boolean; -- input lines</td>
</tr>
<tr>
<td>DEFINE out := q[i]; ic := (mc &amp; pc)</td>
</tr>
<tr>
<td>A[i] := mc &amp; pc &amp; bus[i]; B[i] := !mc &amp; q[i+1];</td>
</tr>
<tr>
<td>ASSIGN next(q[i]) := case ic:</td>
</tr>
<tr>
<td>case</td>
</tr>
<tr>
<td>!S[i] &amp; !R[i]: q[i]; --hold</td>
</tr>
<tr>
<td>S[i] &amp; !R[i]: 1; --set</td>
</tr>
<tr>
<td>!S[i] &amp; R[i]: 0; --reset</td>
</tr>
<tr>
<td>S[i] &amp; R[i]: {0,1}; esac; --undef</td>
</tr>
<tr>
<td>!ic: q[i]; esac; -- unchanged if no input</td>
</tr>
<tr>
<td>next(bus[i]) := case oc:</td>
</tr>
<tr>
<td>Q[i]; !oc: {0,1}; esac;</td>
</tr>
<tr>
<td>FAIRNESS ic FAIRNESS oc</td>
</tr>
</tbody>
</table>

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
5.3. A COMMUNICATION PROTOCOL

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:

\[ A \mathcal{G} \mathcal{E} \mathcal{F}^* \ \text{init} \]

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 combinatorial 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
### 5.3. A Communication Protocol


<table>
<thead>
<tr>
<th>$i$</th>
<th>$b$</th>
<th>$o$</th>
<th>$i'$</th>
<th>$b'$</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{nil}$</td>
<td>$\langle \rangle$</td>
<td>$\text{nil}$</td>
<td>$\langle \rangle$</td>
<td>$\text{nil}$</td>
<td>$\langle \rangle$</td>
</tr>
<tr>
<td>$x$</td>
<td>$\langle \rangle$</td>
<td>$\text{nil}$</td>
<td>$\langle \rangle$</td>
<td>$\text{nil}$</td>
<td>$\langle \rangle$</td>
</tr>
<tr>
<td>$\text{nil}$</td>
<td>$\langle x_1, \ldots, x_v \rangle$</td>
<td>$\text{nil}$</td>
<td>$\langle x_1, \ldots, x_{v-1} \rangle$</td>
<td>$x_v$</td>
<td>$x$</td>
</tr>
<tr>
<td>$x$</td>
<td>$\langle x_1, \ldots, x_v \rangle$</td>
<td>$\text{nil}$</td>
<td>$\langle x, x_1, \ldots, x_{v-1} \rangle$</td>
<td>$x_v$</td>
<td>$x$</td>
</tr>
<tr>
<td>$\text{nil}$</td>
<td>$\langle \rangle$</td>
<td>$y$</td>
<td>$\langle \rangle$</td>
<td>$\langle x \rangle$</td>
<td>$\langle x \rangle$</td>
</tr>
<tr>
<td>$x$</td>
<td>$\langle \rangle$</td>
<td>$y$</td>
<td>$\langle \rangle$</td>
<td>$\langle x \rangle$</td>
<td>$\langle x \rangle$</td>
</tr>
<tr>
<td>$\text{nil}$</td>
<td>$\langle x_1, \ldots, x_v \rangle$</td>
<td>$y$</td>
<td>$\langle x_1, \ldots, x_v \rangle$</td>
<td>$\langle x \rangle$</td>
<td>$\langle x \rangle$</td>
</tr>
<tr>
<td>$x$</td>
<td>$\langle x_1, \ldots, x_v \rangle$</td>
<td>$y$</td>
<td>$\langle x_1, \ldots, x_v \rangle$</td>
<td>$\langle x \rangle$</td>
<td>$\langle x \rangle$</td>
</tr>
</tbody>
</table>

Figure 5.5: Transition relation of a bounded buffer

When $i$, then this value is appended to the buffer, and $i$ is reset to $\text{nil}$. 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 $A \implies G$ ($i \neq \text{nil} \implies X$ ($i = \text{nil}$)) can be used to determine whether a buffer overflow can occur. If the output variable $o$ is $\text{nil}$ 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 $\text{nil}$.

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

```haskell
next(b[j]) := case
  (i=\text{nil}) & (!o=\text{nil}) : b[j];
  (i=\text{nil}) & (o=\text{nil}) : b[j+1];
  (!i=\text{nil}) & (!o=\text{nil}) : if !(b[j]=\text{nil}) & b[j]=\text{nil} then i
    else b[j] fi;
  (!i=\text{nil}) & (o=\text{nil}) : if b[j]=\text{nil} then nil
    else if b[j+1]=\text{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 \( b_j = \text{nil} \), then for all \( k \geq j \), also \( b_k = \text{nil} \). 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 entries. 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 $\varphi$, 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) $\Phi$, and deciding whether $\Phi \vdash \varphi$. However, some experiments will quickly convince the reader that a naive approach of doing so is doomed to failure. Usually, the program axioms all have a very special form, such as

$$(\text{state}_i \rightarrow (X\text{ succ}_i \lor \cdots \lor X\text{ succ}_n))$$

in a linear time modelling, or

$$(\text{state}_i \rightarrow (\langle a_1 \rangle \text{ succ}_i \land \cdots \land \langle a_n \rangle \text{ succ}_i))$$

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 $I$, and current point $w_0 \in U$, is given, and we have to check whether the formula $\varphi$ is satisfied: $(U,I,w_0) \models \varphi$. In the universal definition, we are given universe and interpretation, and want to know
CHAPTER 6. MODEL CHECKING ALGORITHMS

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

Of course, any algorithm which calculates \( \varphi^M \) can also be used to decide whether \( (U, I, w) \models \varphi \): \( w \models \varphi \) if and only if \( w \in \varphi^M \). Vice versa, if we have an efficient algorithm to decide whether \( w \models \varphi \), we can calculate \( \varphi^M \) by an iteration on all states.

Model checking has two parameters: model \( M \) and formula \( \varphi \). Algorithms, which iterate on the structure of \( \varphi \) and in each step traverse the whole of \( M \) are sometimes called global. Algorithms, which iteratively extend the checked part of \( M \) and in each step determine the truth of each sub-formula of \( \varphi \) 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 \( M = (U, I) \) and a multimodal formula \( \varphi \), the set \( \varphi^M \triangleq \{ w \in U \mid w \models \varphi \} \) of points validating \( \varphi \) can be calculated by a recursive descent on the structure of \( \varphi \). If \( p \) is an atomic proposition, then \( p^M \triangleq I(p) \). Furthermore, \( \bot^M \triangleq \{ \} \) and \((\varphi \rightarrow \psi)^M \triangleq U \setminus \varphi^M \cup \psi^M \). Finally, \((\langle R \rangle \psi)^M \triangleq \{ w \in U \mid \exists w' \in \psi^M, (w, w') \in I(R) \}\).

This algorithm seems to be just a trivial rephrasing of the semantical definition for the logical operators. However, there are some important observations. Firstly, \((\langle R \rangle \psi)^M \) can be calculated from \( \psi^M \) in two ways: We can either check for each \( w \in U \), whether the intersection of \( \psi^M \) and \( R(w) \) is nonempty. Alternatively, we can calculate \( \bigcup \{ R^{-1}(w') \mid w' \in \psi^M \} \), where \( R^{-1}(w') \triangleq \{ w \mid (w, w') \in I(R) \} \) is the inverse image of point \( w \) under the relation \( R \). This inverse image calculation can be accomplished by a traversal of all arcs \( (w, w') \in I(R) \): If \( w' \in \psi^M \), then \( w \in (\langle R \rangle \psi)^M \). Secondly, to avoid recalculation of common subformulas, we use a table, where for each sub-formula \( \psi \) the set \( \psi^M \) 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 $\psi_1$ and the points satisfying $\psi_2$. To label the set of points satisfying $\varphi \triangleq E(\psi_2 \ U^+ \psi_1)$ or $\varphi \triangleq A(\psi_2 \ U^+ \psi_1)$, we use the fixpoint unfoldings

$$E(\psi_2 \ U^+ \psi_1) \leftrightarrow E \ X (\psi_1 \lor \psi_2 \land E(\psi_2 \ U^+ \psi_1))$$

$$A(\psi_2 \ U^+ \psi_1) \leftrightarrow A \ X (\psi_1 \lor \psi_2 \land A(\psi_2 \ U^+ \psi_1))$$

For $\varphi \triangleq E(\psi_2 \ U^+ \psi_1)$, we label all points with $\varphi$ which have a successor that is labelled with $\psi_1$, or with $\psi_2$ and also $\varphi$. This process is repeated until stabilisation is reached. For $\varphi \triangleq A(\psi_2 \ U^+ \psi_1)$, we label all points with $\varphi$ for which all successors are labelled with $\psi_1$, or with $\psi_2$ and also $\varphi$. 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 sub-formulas, 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 $E \ F^+$-operator, the problem of marking all points for which $E \ F^+ \varphi$ holds, given the set of point satisfying $\varphi$, 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 $E \ U^+$-operator, this idea can be refined to give an evaluation procedure of linear complexity. The $A \ U^+$-operator can be expressed by

$$A(\psi_2 \ U^+ \psi_1) \leftrightarrow \neg(E(\neg \psi_1 \ U^+(\neg(\psi_1 \land \psi_2)) \lor E \ G^+(\neg \psi_1))$$

Thus, we only need a procedure marking all points for which $E \ G^+ \varphi$ holds. This can be done as follows:
program CTL_check (Model (U, I, w₀), Formula ϕ) =
    if w₀ ∈ eval(ϕ)
    then print("ϕ is satisfied at w₀ in (U, I)")
    else print("ϕ not satisfied at w₀ in (U, I)");

procedure eval (Formula ϕ): Pointset =
    case ϕ of
      p : return I(p);
      ⊥ : return { };
      (ψ₁ → ψ₂) : return U \ eval(ψ₁) ∪ eval(ψ₂);
      E(ψ₂ U⁺ ψ₁) : E₁ := eval(ψ₁); E₂ := eval(ψ₂); E := { };
      repeat until stabilization
        E := E ∪ { w | (succ(w) ∩ (E₁ ∪ E₂) ∩ E) ≠ { } };
      return E;
      A(ψ₂ U⁺ ψ₁) : E₁ := eval(ψ₁); E₂ := eval(ψ₂); E := { };
      repeat until stabilization
        E := E ∪ { w | succ(w) ⊆ E₁ ∪ E₂ };
      return E;
procedure succ (Point w): Pointset = return { w' | (w, w') ∈ I(υ) };

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
6.2. LOCAL LINEAR TIME MODEL CHECKING

in the previous sections, which were used to constrain the set of possible models. For example, an \( \omega \)-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 \( F^+ \psi \), where \( \psi \) is a boolean combination of propositions. For example, the condition \( F^+ T \) 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 \( G^+ F^+ \psi_1 \rightarrow G^+ F^+ \psi_2 \) 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 \( \mathcal{M} = (U, I, w_0) \) and CTL-formula \( \varphi \), the relation \( \mathcal{M} \models \varphi \) holds iff the maximal tree generated from \( \mathcal{M} \) at \( w_0 \) satisfies \( \varphi \). 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 $\varphi$. 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, $F' \psi$ 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 $\mathcal{M} = (U, I, w_0)$ and a modal formula $\varphi$, we want to determine whether there is a maximal sequence generated from $\mathcal{M}$ at $w_0$ which satisfies $\varphi$ 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 $\alpha$ 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 $p \in SF(\varphi)$, then $p \in m$ iff $w \in I(p)$.

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. $(w, w') \in I(R)$,
2. if $(R) \psi \in SF(\varphi)$ and $\psi \in m'$, then $(R) \psi \in m$,
3. if $(R) \psi \in m$, then $\psi \in m'$,
4. some $(R) \psi \in m$.

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 $(R)$-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 $(\lnot R)$; it allows the generated sequence to be finite when no $(R) \psi$ is contained in a node.

Now we can construct a forest of atoms as follows:

• initial nodes are all initial atoms
• any node $\alpha$ has as children all $\alpha'$ such that $X_R(\alpha, \alpha')$
6.2. LOCAL LINEAR TIME MODEL CHECKING

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 \( R \) 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 \( \varphi \).

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 \( X \) and require additionally

5. if \( F \psi \in SF(\varphi) \) then \( F \psi \in m \) iff \( \psi \in m \) or \( XF \psi \in m \)

This requirement corresponds to the recursion axiom \( \vdash F \psi \leftrightarrow \psi \lor XF \psi \). 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 $F^* \phi$ are fulfilled. Since we cannot 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 $F^* \phi$ in some $\alpha \in W$ there exists some $\alpha' \in W$ with $\phi \in \alpha'$. Any atom which does not contain positive future obligations $X \phi$ 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 $F^* \phi \in \alpha$, also $\phi \in \alpha$. The given formula $\phi$ is satisfiable in $M$ iff there exists a self-fulfilling SCC which is reachable from some initial atom. In this case, a natural model for $\phi$ generated by $M$ 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 $U^+$-operators, each positive occurrence ($\psi_1 U^+ \psi_2$) 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 ($\psi_1 U^+ \psi_2$) 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 ($\psi_1 U^+ \psi_2$).

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 backtrace 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 $p \in P$ and one bit for each sub-formula ($\psi_1 U^+ \psi_2 \in SF(\varphi)$). The function children constructs for a given atom $\alpha$ 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 $\alpha$. When the procedure backtracks, $\alpha$ is the root of a maximal SCC iff there are no atoms $\beta$ in the subtree below $\alpha$ such that $\alpha$ is also in the subtree of $\beta$. In this case, the maximal SCC containing $\alpha$ consists of all nodes in the subtree below $\alpha$, 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 $\alpha$ has not occurred,
6.2. LOCAL LINEAR TIME MODEL CHECKING

program LTL_check (Model $\mathcal{M}$, Formula $\varphi$) =

Nat $\text{depth\_first\_count} := 0$; /* number of recursive call */
Atomset $\text{stack} := \{\}$; /* Stack of searched atoms */
Natarray $\text{table}$; /* Hashtable from atoms to natural numbers */
Atomset $\text{init} := \{\alpha \mid \alpha$ is an initial atom of $\mathcal{M}$ and $\varphi\}$;
for all $\alpha \in \text{init}$ do $\text{depth\_first\_search}(\alpha)$;
print("$\varphi$ is not satisfiable in $\mathcal{M}$");

procedure $\text{depth\_first\_search}$ (Atom $\alpha$) =

if ($\text{table}[\alpha]$ = UNDEFINED) then /* $\alpha$ is a new atom */

Nat $\text{dfnumber} := \text{depth\_first\_count}$; /* save current count */
$\text{depth\_first\_count} := \text{depth\_first\_count} + 1$;
$\text{table}[\alpha] := \text{dfnumber}$; /* initialize with current depth */
push($\text{stack}$, $\alpha$);
Atomset $\text{succ} := \text{children}(\alpha)$;
for all ($\beta \in \text{succ}$) do

$\text{depth\_first\_search}(\beta)$;
$\text{table}[\alpha] := \min(\text{table}[\alpha], \text{table}[\beta])$; /* $\beta$ above $\alpha$? */
if ($\text{table}[\alpha] = \text{dfnumber}$) then /* $\alpha$ is the root of an SCC */

Formulaset $\text{required} := \{\}$, $\text{fulfilled} := \{\}$;
repeat

$\beta := \text{pop}(\text{stack})$;
$\text{table}[\beta] := \text{MAXNAT}$;
$\text{required} := \text{required} \cup \{\psi_1 \mid (\psi_2 \ U^+ \psi_1) \in \beta\}$;
$\text{fulfilled} := \text{fulfilled} \cup \{\psi \mid \psi \in \beta\}$
until ($\alpha = \beta$); /* all elements of SCC are popped */
if $\text{required} \subseteq \text{fulfilled}$ /* SCC is self-fulfilling */

then print("$\varphi$ satisfiable in $\mathcal{M}$"); exit;

procedure $\text{children}$ (Atom ($w, m$)) =

if $\{(\psi_2 \ U^+ \psi_1) \in m\} = \{\}$ then return $\{\}$ /* no future obligations*/
else return $\{(w', m') \mid w \prec w',
(\psi_2 \ U^+ \psi_1) \in m$ iff $\psi_1 \in m'$ or $\psi_2 \in m'$ and $(\psi_2 \ U^+ \psi_2) \in m'\}$

Figure 6.3: Depth–first–search LTL model checking algorithm
• the depth-first-number of $\alpha$, when $\alpha$ is first encountered,
• the depth-first-number of the first encountered atom belonging to
  the same strongly connected component as $\alpha$, after return from the
  recursive call, and
• \text{MAXNAT} (any value for which $\min(n, \text{MAXNAT})$ is always $n$), after
  the maximal strong component containing $\alpha$ has been analysed.

The main program calls \texttt{depthfirstsearch} 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 \textit{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 \textit{LTL}-model checking is \textit{PSPACE}-complete in the size of
the formula and \textit{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 $\mu$-calculus

Both the local and the global model checking algorithms can be easily
adapted to monotonic $\mu$-\textit{TL}. Global model checking for \textit{CTL} unfolds the
fixpoint definition of the $A \textit{U}$ and $E \textit{U}$ operators. If we restrict our at-
tention to \textit{continuous $\mu$-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
\textit{BDDs} (see [BCM+92]).
According to the Knaster-Tarski theorem proved in Chapter 2.4,

\[(U, I, w) \models \nu q \varphi \iff w \in \bigcup \{Q \mid Q \subseteq \varphi^T[q := Q]\}\]

\[(U, I, w) \models \mu q \varphi \iff w \in \bigcap \{Q \mid \varphi^T[q := \bot] \subseteq Q\}\]

A function \(f : 2^U \rightarrow 2^U\) 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 \(\varphi\) is union-continuous, then the fixpoints can be obtained as

\[\nu q \varphi = \text{lim}_{i \rightarrow \omega} \varphi^i(T)\]

\[\mu q \varphi = \text{lim}_{i \rightarrow \omega} \varphi^i(\bot)\]

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:

\[\nu q \varphi = \text{lim}_{i \leq \text{card}(U)} \varphi^i(T)\]

\[\mu q \varphi = \text{lim}_{i \leq \text{card}(U)} \varphi^i(\bot)\]

Consequently, for finite domains model checking can be performed by extending the naive global algorithm. The result is depicted in Figure 6.4.

Since every repeat in this algorithm can iterate up to \(\text{card}(U)\) times, the complexity is of order \(|\varphi| \cdot \text{card}(U)^{qd(\varphi)}\), where \(qd(\varphi)\) is the depth of nesting of fixpoint quantors in \(\varphi\). 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 \(\text{CTL}^\nu(F (p_1 \land F p_2)).\)

\[\mu_{\text{CTL}}(F (p_1 \land F p_2)) = \mu q_1 (X q_1 \lor (p_1 \land \mu q_2 (X q_2 \lor p_2)))\]

In the inner fixed point formula \(\mu q_2 (X q_2 \lor p_2)\) there is no occurrence of \(q_1\). Therefore, in the evaluation of \(\mu q_1\), this formula has a constant value. In contrast, consider the \(\mu_{\text{TL}}\) formula

\[\mu q_1 (p_1 \land \mu q_2 (X q_1 \lor X q_2 \lor p_2))\]

Here the inner formula \(\mu q_2 (X q_1 \lor X q_2 \lor p_2)\) is re-evaluated for every new iteration of \(q_1\). That is, if \(\psi(q_1, q_2) \triangleq (X q_1 \lor X q_2 \lor p_2)^M\) and \(\varphi(q_1) \triangleq (p_1 \land \mu q_2 \psi(q_1, q_2))^M\), we can calculate \(\mu q_1 \varphi(q_1)\) by iterating

\[\varphi^0 \triangleq \{\}\]
procedure eval (Formula $\varphi$): Pointset =
case $\varphi$
of
$p : \text{return } I(p) ; /* interpretation of proposition } p */$
$q : \text{return } v(q) ; /* valuation of proposition variable } q */$
$\bot : \text{return } \{\}$;
$(\psi_1 \rightarrow \psi_2) : \text{return } U \setminus \text{eval}(\psi_1) \cup \text{eval}(\psi_2);$

$\langle R \rangle \psi : \text{return } R^{-1}(\text{eval}(\psi));$

$\nu q(\psi) : H := U;$
repeat until stabilization
   $H := \text{eval}(\psi\{q := H\});$
return $H;$

$\mu q(\psi) : H := \{\}$;
repeat until stabilization
   $H := \text{eval}(\psi\{q := H\});$
return $H;$

Figure 6.4: naïve global branching time $\mu$TL model checking algorithm

\[
\begin{align*}
\psi^{0,0} & \triangleq \{\} \\
\psi^{0,1} & \triangleq \psi (\varphi^0, \psi^{0,0}) = (X \perp \lor X \perp \lor p_2), \\
\psi^{0,2} & \triangleq \psi (\varphi^0, \psi^{0,1}) = (X \perp \lor X (X \perp \lor p_2) \lor p_2), \\
& \quad \cdots \\
\psi^{0,n+1} & \triangleq \psi (\varphi^0, \psi^{0,n}) = \mu q_2(X \perp \lor X q_2 \lor p_2), \text{ if } \psi^{0,n+1} = \psi^{0,n}, \\
\varphi^1 & \triangleq \varphi (\varphi^0) = (p_1 \land \mu q_2(X \perp \lor X q_2 \lor p_2)) = (p_1 \land \psi^{0,n}), \\
\psi^{1,0} & \triangleq \{\} \\
\psi^{1,1} & \triangleq \psi (\varphi^1, \psi^{1,0}) = (X (p_1 \land \psi^{0,n}) \lor X \perp \lor p_2), \\
& \quad \cdots
\end{align*}
\]

and so on. A more sophisticated algorithm was given in [EL86]. A sequence $\nu q_1, \ldots, \nu q_n$, or $\mu q_1, \ldots, \mu q_n$, of nested fixpoints of the same type can be calculated by a single loop. Since $\psi$ 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 $\bot$. 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 $\varphi$. 
6.3. MODEL CHECKING FOR $\mu$-CALCULUS

In [LBC$^+$94] the authors observe that by storing even more intermediate values, the time complexity for evaluating fixpoint formulas can be reduced to $O(|U|^{m/2} + 1)$. 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, $\psi$ is a sub-formula of the given formula and $\Delta$ 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 $\psi_i$ uses at most variables from $q_1, ..., q_{i-1}$. For simplicity, we use $\vee$, $\wedge$, $\langle R \rangle$, $[R]$, $\mu$ and $\nu$ as basic operators and assume that negations only occur in literals. Furthermore, we assume that in the formula to be checked each $\mu$ and $\nu$ quantification binds a different proposition variable.

Since in [SW91] the $\mu$-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 $\Delta, w \models (\psi_1 \lor \psi_2)$ 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 $\varphi$, 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.

![Tableau rules for branching time $\mu$TL](image)

Figure 6.5: Tableau rules for branching time $\mu$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 \( ([R]) \) can only be applied if \( w' \in R(w) \).

• Similarly, in rule \( ([R]) \), it must hold that \( R(w) = \{ w_1, \ldots, w_n \} \).

• In rule \( (\mu) \) and \( (\nu) \), \( \Delta' \triangleq \Delta \cup \{ q = \psi \} \).

• 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 \( \psi \)).

That is, to check whether \( p \psi \psi \) holds in point \( w \), we record that \( q \) must be interpreted as a fixpoint of \( \psi(q) \), and check whether \( \psi \) holds in \( w \). Whenever we hit upon the proposition variable \( q \) in the further decomposition of \( \psi(q) \), we can unfold this occurrence to \( \psi \). 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

- \( \psi = p \in \mathcal{P} \) and \( w \in I(p) \), or \( \psi = \neg p \) and \( w \notin I(p) \),
- \( \psi = q \in \mathcal{Q}, q \notin \Delta, w \in v(q), \) or \( \psi = \neg q, q \notin \Delta, w \notin v(q), \) or
- \( \psi = (R) \psi' \) and \( R(w) = \{ \} \) (Rule \( ([R]) \) produces no children),
- \( \psi = q \in \mathcal{Q} \) and \( q \) was included in \( \Delta \) by rule \( (\nu) \).

In other words, a maximal tableau is not successful if it contains some unsuccessful leaf \( \Delta, w \models \psi \) which satisfies

- \( \psi = p \in \mathcal{P} \) and \( w \notin I(p) \), or \( \psi = \neg p \) and \( w \in I(p) \),
- \( \psi = q \in \mathcal{Q}, q \notin \Delta, w \notin v(q), \) or \( \psi = \neg q, q \notin \Delta, w \in v(q), \) or
- \( \psi = (R) \psi' \) and \( R(w) = \{ \} \) (Rule \( ([R]) \) not applicable),
- \( \psi = q \in \mathcal{Q} \) and \( q \) was included in \( \Delta \) by rule \( (\mu) \).

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** \( w \in \varphi^M \) 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 $v_{i1}, \ldots, v_{ik}$, 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 $V = \{v_1, \ldots, v_n\}$ of a shared variables program are over a binary domain, then any propositional formula $\varphi$ over $P = \{v_1, \ldots, v_n\}$ 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 $V = \{v_1, \ldots, v_n\}$ can be represented as an ordinary propositional formula over $P = \{v_1, \ldots, v_n, v'_1, \ldots, v'_{n}\}$. If the transition relation is given as a propositional formula with equalities, we replace $0$ by $\perp$, and 1 by $\top$, and $(v = v')$ by $(v \leftrightarrow v')$. For example, the formula

$$v_1 = 0 \rightarrow ((v'_1 = 1) \land (v'_2 = v_2) \land (v'_3 \neq v_3))$$

in this notation becomes

$$\neg v_1 \rightarrow (v'_1 \land (v'_2 \leftrightarrow v_2) \land \neg (v'_3 \leftrightarrow v_3))$$

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 $2^{10} \approx 10^{30}$. 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 \(10^{40}\).

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 \(O(|D| \cdot \log_2(|D|))\) 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 \(n = n_4n_3n_2n_1\) of the domain \(D\), the above explicit enumeration is

\[
S = \{0000, 0010, 0100, 0110, 1000, 1010, 1100, 1110, 1111\}
\]

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

\[
S = \{n \mid n_1 = 0 \lor n_4 = 1 \land n_3 = 1\}
\]

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 \( \texttt{he} \) ("if-then-else"), such that
\[
\texttt{he}(\varphi, \psi_1, \psi_2) \triangleq ((\varphi \rightarrow \psi_1) \land (\neg \varphi \rightarrow \psi_2)).
\]
Equivalently, \( \texttt{he}(\varphi, \psi_1, \psi_2) \leftrightarrow ((\varphi \land \psi_1) \lor (\neg \varphi \land \psi_2)) \). Then \( (\varphi \rightarrow \psi) \leftrightarrow \texttt{he}(\varphi, \psi, \top) \), hence all boolean operators can be expressed with \( \texttt{he} \), \( \bot \) and \( \top \). A formula \( \psi \) is said to be in tree form, if \( \psi = \bot \), or \( \psi = \top \), or \( \psi = \texttt{he}(p, \psi_1, \psi_2) \), where \( p \in \mathcal{P} \) and \( \psi_1 \) and \( \psi_2 \) are in tree form. In other words, a formula \( \psi \) is in tree form, if it uses only \( \texttt{he} \), \( \bot \), \( \top \), and propositions, and, additionally, for every subformula \( \texttt{he}(\psi, \psi_1, \psi_2) \) of \( \psi \), the formula \( \varphi \) is a proposition, and \( \psi_1 \) and \( \psi_2 \) are not propositions. A tree form formula can be drawn as binary decision tree, where for each subformula \( \texttt{he}(p, \psi_1, \psi_2) \) there is a node labelled \( p \) which has \( \psi_2 \) and \( \psi_1 \) 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 \( \texttt{he}(p, \varphi_1, \varphi_2) \) of \( \varphi \), and every subformula \( \texttt{he}(q, \psi_1, \psi_2) \) of \( \varphi_1 \) or \( \varphi_2 \), it holds that \( p < q \). An ordered tree form formula is called reduced, if it does not contain any redundant subformula \( \texttt{he}(p, \psi, \psi) \) (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:
\[
\varphi \leftrightarrow \texttt{he}(p, \varphi(p := \top), \varphi(p := \bot)),
\]
and boolean transformations and simplifications like \( \texttt{he}(p, \psi, \psi) \leftrightarrow \psi \) and \( (\bot \rightarrow q) \leftrightarrow \top \).

For example, truth table and tree form formula for the above set are given in Figure 6.6.
<table>
<thead>
<tr>
<th>$n_4$</th>
<th>$n_3$</th>
<th>$n_2$</th>
<th>$n_1$</th>
<th>$S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

$S = \phi(n_4,$
$\phi(n_3,$
$\phi(n_2,$
$\phi(n_1, \top, \top)$,
$\phi(n_1, \top, \top)),$
$\phi(n_2,$
$\phi(n_1, \bot, \top)$,
$\phi(n_1, \bot, \top)),$
$\phi(n_3,$
$\phi(n_2,$
$\phi(n_1, \bot, \top)$,
$\phi(n_1, \bot, \top))$)

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 $\phi(p, \psi, \psi)$ in the above tree form formula by $\psi$:

$$S = \phi(n_4, \phi(n_3, \top, \phi(n_1, \bot, \top)), \phi(n_1, \bot, \top))$$

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 $\mathcal{P}_i \triangleq \mathcal{P} \cup \{\delta_1, \ldots, \delta_n\}$, together with a (nonrecursive) list of abbreviations $(\delta_1 \triangleq \psi_1, \ldots, \delta_n \triangleq \psi_n)$. In each abbreviation, $\psi_i$ is an abbreviated formula $\phi(p, \varphi, \varphi')$ over the alphabet $\mathcal{P}_i \triangleq \mathcal{P} \cup \{\delta_{i+1}, \ldots, \delta_n\}$. A formula is *maximally abbreviated*, if

1. no compound subformula $\phi(p, \varphi_1, \varphi_2)$ appears twice, and
2. no two abbreviations have the same right hand side.

For the above example, a maximally abbreviated formula is

$$S = \phi(n_4, \phi(n_3, \top, \delta), \delta), \quad \text{where} \quad \delta \triangleq \phi(n_1, \bot, \top)$$

Figure 6.6: Truth table and tree form formula
A maximally abbreviated formula is in BDD form, if for all subformulas \( \text{le}(p, \varphi_1, \varphi_2) \), both \( \varphi_1 \) and \( \varphi_2 \) are from \( \{ \bot, \top, \delta_1, \ldots, \delta_n \} \). This normal form can be obtained by introducing further definitions:

\[
S = \text{le}(n_4, \delta_1, \delta_2), \text{ where } \delta_1 \triangleq \text{le}(n_3, \top, \delta_2) \text{ and } \delta_2 \triangleq \text{le}(n_1, \bot, \top)
\]

BDD form formulas can be drawn as binary decision diagrams: For any \( \delta \triangleq \text{le}(p, \delta_1, \delta_2) \), draw a node labelled \( p \) with reference \( \delta \), which has the nodes referenced by \( \delta_2 \) and \( \delta_1 \) 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 “+” leaves). 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:

\[
\varphi(v_1...v_n) \leftrightarrow \text{le}(v_1, \varphi\{v_i := \top\}(v_{i+1}...v_n), \varphi\{v_i := \bot\}(v_{i+1}...v_n))
\]

This gives the unique binary decision tree for the chosen ordering. To obtain the BDD for \(\varphi(v_1...v_n)\) we recursively calculate the BDD \(\delta_1\) for \(\varphi\{v_i := \top\}(v_{i+1}...v_n)\) and \(\delta_2\) for \(\varphi\{v_i := \bot\}(v_{i+1}...v_n)\). Upon backtrack, a new node \(\delta \triangleq \text{le}(v, \delta_1, \delta_2)\) 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 \(\delta \triangleq \text{le}(v, \delta_1, \delta_2)\) as a hash table from \((v, \delta_1, \delta_2)\) to \(\delta\).

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, \(\delta\) (of type “Bdd”) is the number of the BDD node, \(i\) (of type “Bddvar”) is the number of a BDD variable, and \(\delta_1\) and \(\delta_2\) (of type “Bdd”) are links to other BDD nodes. For each \((i, \delta_1, \delta_2)\) the hash table returns the pointer \(\delta\), if this node exists in the BDD.
6.4. BINARY DECISION DIAGRAMS

The resulting algorithm is given in Figure 6.7. It takes as input a PL formula with \( \mathcal{P} = \{v_1, \ldots, v_n\} \) and calculates the table of BDD nodes and a pointer to the topmost node for the variable ordering \((v_1, \ldots, v_n)\).

```
function PL2BDD (Formula \( \varphi \)) : (Nodeset, Bdd) =
    /* Calculates the BDD of \( \varphi \)
        as a set of nodes and a pointer to the topmost node */
    Nodeset table := \{\} /* Table of BDD nodes \( (\delta, i, \delta_1, \delta_2) \) */
    Bdd max := 1; /* Index of maximal table entry */
    Bdd result := EDD(\( \varphi \), 1); /* Index of topmost BDD node */
    return (table, result);

function BDD (Formula \( \varphi \), Bddvar \( i \)) : Bdd =
    /* \( \varphi \) 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(\( \varphi \)) /* \( \varphi \) is a boolean constant */
    else \( \delta_1 := \) BDD(\( \varphi \{v_i := \bot \} \), \( i + 1 \))
        \( \delta_2 := \) BDD(\( \varphi \{v_i := \top \} \), \( i + 1 \))
        if \( \delta_1 = \delta_2 \) then return \( \delta_1 \)
        elsif \( \exists \delta : (\delta, i, \delta_1, \delta_2) \in \) table then return \( \delta \)
        else max := max + 1; table := table \cup \{ (\max, i, \delta_1, \delta_2) \};
        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 \( \bar{w} \) 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 \( \Box \)-operator commutes with other boolean connectives:

\[
(\Box (p, \varphi_1, \varphi_2) \to \psi) \iff \Box (p, (\varphi_1 \to \psi), (\varphi_2 \to \psi))
\]

\[
(\psi \to \Box (q, \varphi_1, \varphi_2)) \iff \Box (q, (\psi \to \varphi_1), (\psi \to \varphi_2))
\]

Similar equivalences hold for \( \land, \lor \), etc. We prove only the first one of these equivalences. Recall that \( \Box (p, \varphi_1, \varphi_2) \) is defined by \( \Box (p, \varphi_1, \varphi_2) \iff ((p \to \psi_1) \land (\neg p \to \psi_2)) \).

\[
(\Box (p, \varphi_1, \varphi_2) \to \psi) \iff (((p \land \varphi_1) \lor (\neg p \land \varphi_2)) \to \psi)
\]
\[ \leftrightarrow ( ((\neg p \lor \neg \varphi_1) \land (p \lor \neg \varphi_2)) \lor \psi ) \]
\[ \leftrightarrow ( ((\neg p \lor \neg \varphi_1 \lor \psi) \land (p \lor \neg \varphi_2 \lor \psi)) \]
\[ \leftrightarrow ( (p \land (\varphi_1 \rightarrow \psi)) \lor (\neg p \land (\varphi_2 \rightarrow \psi))) \]
\[ \leftrightarrow \mathbf{b}(p, (\varphi_1 \rightarrow \psi), (\varphi_2 \rightarrow \psi)) \] \hfill \Box

Given BDDs for \( \varphi \) and \( \psi \), the BDD for \( (\varphi \rightarrow \psi) \) can be constructed as follows. Since \( BDD(\varphi) \) and \( BDD(\psi) \) can be either 0, 1, or \( \mathbf{b}(v, \delta_1, \delta_2) \), 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 \( \mathbf{b}(v, \delta_1, \delta_2) \), and \( BDD(\psi) \) is the leaf 0, we use the equivalence:

\[ (\mathbf{b}(v, \delta_1, \delta_2) \rightarrow \bot) \leftrightarrow \mathbf{b}(v, (\delta_1 \rightarrow \bot), (\delta_2 \rightarrow \bot)) \]

Since \( \neg \varphi \triangleq (\varphi \rightarrow \bot) \), this means that the BDD for \( \neg \varphi \) is constructed from the BDD for \( \varphi \) by exchanging all leaves 0 and 1. The only remaining case is that both \( BDD(\varphi) = \mathbf{b}(v, \varphi_1, \varphi_2) \) and \( BDD(\psi) = \mathbf{b}(v', \psi_1, \psi_2) \) are internal nodes. There are three subcases:

1. \( v = v' \): \( (\mathbf{b}(v, \varphi_1, \varphi_2) \rightarrow \mathbf{b}(v, \psi_1, \psi_2)) \leftrightarrow \mathbf{b}(v, (\varphi_1 \rightarrow \psi_1), (\varphi_2 \rightarrow \psi_2)) \)

2. \( v < v' \) in the order of variables:
   \[ (\mathbf{b}(v, \varphi_1, \varphi_2) \rightarrow \mathbf{b}(v', \psi_1, \psi_2)) \leftrightarrow \mathbf{b}(v, \varphi_1 \rightarrow \mathbf{b}(v', \psi_1, \psi_2), \varphi_2 \rightarrow \mathbf{b}(v', \psi_1, \psi_2)) \]

3. \( v > v' \) in the order of variables:
   \[ (\mathbf{b}(v, \varphi_1, \varphi_2) \rightarrow \mathbf{b}(v', \psi_1, \psi_2)) \leftrightarrow \mathbf{b}(v', \mathbf{b}(v, \varphi_1, \varphi_2) \rightarrow \psi_1, \mathbf{b}(v, \varphi_1, \varphi_2) \rightarrow \psi_2) \]

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 \( \text{BDD}_{\text{imp}} \) 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 \( \varphi \) and \( \psi \) can be calculated from the BDDs of \( \varphi \) and \( \psi \) using the definition \( (\varphi \lor \psi) \leftrightarrow \neg(\varphi \rightarrow \neg \psi) \). In practice,
function BDD_imp (Bdd \(\varphi, \psi\)) : Bdd =
  /* Calculates the BDD of \((\varphi \rightarrow \psi)\) from the BDDs of \(\varphi\) and \(\psi\) */
  if \(\varphi = 0\) or \(\psi = 1\) then return 1
  elsif \(\varphi = 1\) then return \(\psi\)
  elsif \(\psi = 0\) and \((\varphi, i, \varphi_1, \varphi_2) \in table_\varphi\)
    then return new_node\((i, BDD_imp(\varphi_1, 0), BDD_imp(\varphi_2, 0))\)
  else \((\varphi, i, \varphi_1, \varphi_2) \in table_\varphi\) and \((\psi, j, \psi_1, \psi_2) \in table_\psi\)
    if \(i = j\) then
      return new_node\((i, BDD_imp(\varphi_1, \psi_1), BDD_imp(\varphi_2, \psi_2))\)
    elsif \(i < j\) then
      return new_node\((i, BDD_imp(\varphi_1, \psi), BDD_imp(\varphi_2, \psi_2))\)
    elsif \(i > j\) then
      return new_node\((j, BDD_imp(\varphi, \psi_1), BDD_imp(\varphi, \psi_2))\);

function new_node (Bddvar \(i\), Bdd \(\delta_1, \delta_2\)) : Bdd =
  /* \(i\) is number of a BDD variable, \(\delta_1, \delta_2\) pointers to BDD nodes */
  if \(\delta_1 = \delta_2\) then return \(\delta_1\)
  elsif \(\exists \delta : (\delta, i, \delta_1, \delta_2) \in table\) then return \(\delta\)
  else max := max + 1; table := table \cup \{\text{max}, i, \delta_1, \delta_2\};
  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 φ, Bdd 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(φ, f₀, g₀);  δ₂ := BDD_apply(ψ, 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, ψ₁, ψ₂) ∈ table and ψ ∈ {0,1} then return i
    elsif (φ, i, ψ₁, ψ₂) ∈ 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

\[ v_1 = 0 \rightarrow ((v'_1 = 1) \land (v'_2 = v_2) \land (v'_3 \neq v_3)) \]

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 leaves.)

![BDD Diagram]

For the variable ordering \((v_1, v'_1, v_2, v'_2, v_3, v'_3)\), however, we obtain the following much smaller BDD:

![Smaller BDD Diagram]

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 feasible. 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 \( \mu \)-calculus, all set operations can be directly performed with BDDs. Calculation of the BDD for \( \langle R \rangle \psi \) from the BDD for \( \psi \) amounts to calculating the inverse image of \( \psi \) under the relation \( R \). This is done using propositional quantification: Recall that the BDD for \( \psi \) is using variables \( (v_1, \ldots, v_n) \), and the BDD for \( R \) is defined over the variables \( (v_1, \ldots, v_n, v'_1, \ldots, v'_n) \). To get the BDD for \( \langle R \rangle \psi \), we first rename all variables \( v_i \) in the BDD for \( \psi \) by \( v'_i \), then build the intersection of this BDD with the BDD for \( R \) to obtain a BDD over \( (v_1, \ldots, v_n, v'_1, \ldots, v'_n) \), 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 \( \mu \)-calculus. Given a finite relational frame \( F = (S, \mathcal{I}) \) and a relational term \( \rho \) or formula \( \varphi \), model checking can be used to determine the denotation \( \rho^F \) or \( \varphi^F \), 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 \( I \) of a relation of type \( (D_1, \ldots, D_n) \) is represented by a BDD with variables \( v_1, v_2, \ldots, v_n \).

Likewise, a function of type \( (D_0, \ldots, D_m) \) can be coded as a BDD over \( v_0, \ldots, v_m \). Constants are represented as BDDs with a single node, namely \( \delta \triangleq \mathbf{e}(v, 0, 1) \) or \( \delta \triangleq \mathbf{e}(v, 1, 0) \). Function application \( f(t_0, \ldots, t_{m-1}) \) 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 \( \mathbf{e}(v, \delta_1, \delta_2) \) by \( \delta_1 \) or \( \delta_2 \), respectively. In the BDD representation of functions, for each node \( \delta \triangleq \mathbf{e}(v_m, \delta_1, \delta_2) \) it holds that either \( \delta_1 \) or \( \delta_2 \) 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 \( x_1^i, \ldots, x_n^i \).

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 \( x_1, \ldots, x_m \) is represented as a BDD with additional BDD variables \( x_1, \ldots, x_m \). 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 \( \delta \) is the name of this node, \( i \) is a variable from the set \( \{v_1, \ldots, v_n, x_1, \ldots, x_m\} \), and each \( \delta_j \) 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.

```haskell
function BDD_form (Formula \( \varphi \), Interpretation \( I \)) : Bdd =
    /* Calculates the BDD of formula \( \varphi \) in the interpretation \( I \) */
    case \( \varphi \) of
        \( x \in V \): return \( \mathbf{b}(x, 1, 0) \);
        \( (x_1 = x_2) \): return \( \mathbf{b}(x_1, \mathbf{b}(x_2, 1, 0), \mathbf{b}(x_2, 0, 1)) \);
        \( \bot \): return \( 0 \);
        \( (\varphi_1 \rightarrow \varphi_2) \): return BDD_imp(BDD_form(\( \varphi_1 \), \( I \)), BDD_form(\( \varphi_2 \), \( I \)));
        \( \exists x \ \varphi \): return BDD_exists(\( x \), BDD_form(\( \varphi \), \( I \)));
        \( \rho_{x_1 \ldots x_n} \): return BDD_term(\( \rho \), \( I \))\{\( v_1 := x_1 \} \ldots \{ v_n := x_n \} \};

function BDD_term (RelationalTerm \( \rho \), Interpretation \( I \)) : Bdd =
    /* Calculates the BDD of term \( \rho \) in the interpretation \( I \) */
    case \( \rho \) of
        \( R \in R \): return \( I(R) \) /* pointer to BDD for \( R \) */;
        \( X \in V \): return \( X \) /* name of \( X \) */;
        \( \lambda x_1 \ldots x_n \ \varphi \): return BDD_form(\( \varphi \), \( I \))\{\( x_1 := v_1 \} \ldots \{ x_n := v_n \} \};
        \( \mu X \ \rho \): \( r := BDD_term(\( \rho \), \( I \)) \); return BDD_lfp(\( r \), \( 0 \));

function BDD_lfp (BDD \( r \), BDD \( X^i \)) : BDD =
    /* Fixpoint iteration of BDD \( r \) for \( \rho \) 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 \( \varphi \) and (the BDD representation of) the interpretation \( I \) in frame \( \mathcal{F} \), and returns a BDD which is satisfied by a given valuation \( \mathbf{v} \) iff \( (S, I, \mathbf{v}) \models \varphi \). 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 \( \rho \), uses the function `BDD_term(\( \rho \), \( I \))` to find a representation of the relational term \( \rho \) (under the
interpretation \( I \), then substitutes the argument variables \( x_1, \ldots, x_n \) for the place-holder variables \( v_1, \ldots, v_n \), producing a BDD which is satisfied iff \( \rho \) holds for \( x_1, \ldots, x_n \).

The function \( \text{BDD}_{\text{term}} \) takes as arguments a relational term \( \rho \) and the BDD representation of the interpretation \( I \). It returns a BDD which represents the relation term in the manner described above. The first and second case in the definition of \( \text{BDD}_{\text{term}} \), 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, \( \lambda \)-abstraction, produces a BDD with variables \( v_1, \ldots, v_n \) substituted for the variables \( x_1, \ldots, x_n \). This is the representation for an \( n \)-ary relation which holds iff its arguments satisfy the formula \( \varphi \) when assigned to \( x_1, \ldots, x_n \). The most interesting case is the last: the fixed point operator \( \mu \). 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 \( \text{BDD}_{\text{term}}(\rho, I) \) to get a BDD \( r \) for \( \rho \). Then we compute the fixed point by a series of approximations \( X^0, X^1, \ldots, \), 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 \( \rho \) 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 \( \mu \text{cke} \) model checker [Bie97] is one of the first tools for the relational \( \mu \)-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, \( \mu \text{cke} \) 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 \( \lambda \)-abstraction as shown above. Since \( \mu \text{cke} \) 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 \( \mu \)-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, Ahl91] to represent timing constraints. Real time systems often are modelled with timed automata [Ahn98] 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 $Q$ be the set of rational numbers, and $T$ the set of nonnegative rational numbers. $T$ is the so-called *time-domain*. Mostly, we will use $\tau$ as a metavariable to range over time points (this is completely unrelated to the CSP $\tau$-event). In our context, a time transition system $T$ is a tuple \(T = (\Sigma, S, \Delta, S_0)\), where

- $\Sigma$ 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*.
- $\text{Eft}, \text{Lft} : \Delta \rightarrow T$ are functions for the *earliest* and *latest firing times* of transitions, satisfying $\text{Eft}(t) \leq \text{Lft}(t)$ for all $t \in \Delta$.

Time transition systems generate *timed \(\omega\)-words*: finite or infinite sequences of events. An event $e$ is a tuple $e = (a, \tau)$, where $a \in \Sigma$ and $\tau \in T$. Let $\sigma = (e_1, e_2, e_3, ...)$, where $e_i = (\sigma_i, \tau_i)$, be such an \(\omega\)-word over $\Sigma$. We say that $\sigma$ 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

• \( \tau_i + \text{Eft}(t_i) \leq \tau_{i+1} \leq \tau_i + \text{Lft}(t_i) \).

Usually, the following non-Zeno'ness condition is imposed on generated words:

• Time increases beyond any bound: if \( \sigma \) is infinite, then for any \( \tau \in T \) 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 \( \omega \)-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 \( \lambda \) of such an automaton is a tuple \( \lambda \triangleq (s, \text{clock}) \), where \( s \in S \) is a state and clock is an assignment of clock values to the time domain \( T \). 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 \( \text{Eft}(t) \leq c_s \leq \text{Lft}(t) \).

Unfortunately, concurrency cannot 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
7.2. TIMED AUTOMATA AND TIME PETRI NETS

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, $N = (P, T, F, \text{Eft}, \text{Lft}, \mu_0)$, 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;
- $\text{Eft}, \text{Lft} : T \to \mathbb{T}$ are functions for the earliest and latest firing times of transitions, satisfying $\text{Eft}(t) \leq \text{Lft}(t)$ 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 $\mu$ of $N$ is any subset of $P$. A transition is enabled in a marking $\mu$ if $\bullet t \subseteq \mu$ (all its input places have tokens in $\mu$); otherwise, it is disabled. Let $\text{enabled}(\mu)$ be the set of transitions enabled in $\mu$.

A location $\lambda$ of a time Petri net is a pair $(\mu, \text{clock})$, where $\mu$ is a marking and clock is a function $T \to \mathbb{T}$. The initial location $\lambda_0$ is $(\mu_0, \text{clock}_0)$, where $\text{clock}_0(t) = 0$ for all $t \in T$.

The locations of time Petri nets change, if time passes or if a transition fires. In location $\lambda = (\mu, \text{clock})$, time $\tau \in \mathbb{T}$ can pass, if for all $t \in \text{enabled}(\mu)$, $\text{clock}(t) + \tau \leq \text{Lft}(t)$. In this case, location $\lambda' = (\mu', \text{clock}')$ is obtained by passing $\tau$ from $\lambda$, if

1. $\mu = \mu'$, and
2. for all $t \in T$, $\text{clock}'(t) = \text{clock}(t) + \tau$.

In location $\lambda = (\mu, \text{clock})$, transition $t \in T$ can fire, if $t \in \text{enabled}(\mu)$, and $\text{clock}(t) \geq \text{Eft}(t)$. In this case, location $\lambda' = (\mu', \text{clock}')$ is obtained by firing $t$ from $\lambda$, if

1. $\mu' = (\mu \setminus \bullet t) \cup t \bullet$, and
2. for all $\hat{t} \in T$, $\text{clock}'(\hat{t}) = \begin{cases} 0 & \text{if } \hat{t} \in \text{enabled}(\mu'), \hat{t} \not\in \text{enabled}(\mu) \\ \text{clock}(\hat{t}) & \text{else} \end{cases}$.
Intuitively, this can be interpreted as follows: Firing a transition \( t \) consumes no time, but updates \( \mu \) and clock such that the clocks associated with newly enabled transitions (i.e., transitions which are enabled in \( \mu' \) but not in \( \mu \)) 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 \( \lambda_0 \) is the initial location, and \( \lambda_{i+1} \) is obtained from \( \lambda_i \) by passing time \( \tau \) and then firing transition \( t \). We write \( \lambda_i(\rho) \) for the \( i \)-th location of \( \rho \), and similarly \( \mu_i(\rho) \) and \( \text{clock}_i(\rho) \), and omit the argument \( (\rho) \) 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 \( \rho \) and \( i \geq 0 \), we define \( \text{time}_i(\rho) \) to be the sum of all times \( \tau \) passed between \( \lambda_0(\rho) \) and \( \lambda_i(\rho) \); that is, \( \text{time}_0(\rho) = 0 \) and \( \text{time}_{i+1}(\rho) = \text{time}_i(\rho) + \text{clock}_{i+1}(t) - \text{clock}_i(t) \) for some \( t \) which is not newly enabled in \( \mu_{i+1} \).

A location \( \lambda \) is reachable if there exists a finite run whose last location is \( \lambda \). A time Petri net is one-safe, if for every location \( \lambda = (\mu, \text{clock}) \) obtained by passing time from any reachable location \( \lambda' \), and for every transition \( t \) which can fire in \( \lambda \), \( t \bullet \cap \mu = \emptyset \). 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 \( t_1 \bullet \cap t_2 \neq \emptyset \), \( t_2 \bullet \cap t_3 \neq \emptyset \), \ldots, \( t_n \bullet \cap t_1 \neq \emptyset \) it holds that \( \text{Eft}(t_1) + \text{Eft}(t_2) + \cdots + \text{Eft}(t_n) > 0 \). 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 \( N_2 \). 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 \( B(N_2) \) contains an infinite number of runs. Furthermore, since \( \text{Eft}(t_0) > \text{Lft}(t_1) \), \( t_0 \) can never fire, and thus every run of \( N_2 \) is infinite.
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 $\varphi$, we want to find whether there exists a run $\rho$ of $N$ satisfying $\varphi$ (written $\rho \models \varphi$). In general there are infinitely many runs of $N$, therefore we group these into a finite number of equivalence classes $[\rho_1], [\rho_2], \ldots, [\rho_c]$, such that the existence of a satisfying run $\rho$ implies that every element of the equivalence class $[\rho]$ satisfies $\varphi$. 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 \( \lambda_i \) of a run \( \rho \) is defined. Two runs \( \rho \) and \( \rho' \) 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 \( \lambda_{i+1} \) in a run \( \rho \) 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 \( \rho \) and \( \rho' \) are stuttering equivalent w.r.t. \( \{p_1, \ldots, p_k\} \), if the two sequences obtained by eliminating all stuttering locations from \( \rho \) and \( \rho' \) are strongly equivalent w.r.t. \( \{p_1, \ldots, p_k\} \). Define a formula \( \varphi \) to be stuttering invariant, if for any two runs \( \rho \) and \( \rho' \) which are stuttering equivalent with respect to the atomic propositions in \( \varphi \) 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 \( N = (P, T, F, E, ft, \mu_0) \), let \( \mathcal{P} = \{p^* \mid p \in P\} \cup \{p^\circ \mid p \in P\} \) 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 \( c \in \mathbb{Q} \), then \( x - y \leq c \) is a formula.
- Every propositional variable is a formula.
- \( \bot \) is a formula.
- If \( \varphi_1 \) and \( \varphi_2 \) 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 \( c \in \mathbb{Q} \) are called atomic propositions. Additional boolean connectives \( \mathbf{true}, \neg, \wedge, \vee, \rightarrow, \) 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 \( \text{TNL} \), the value of time variables in a location of a run has to be defined. Intuitively, \( p^* \) and \( p^i \in V \) represent the time when the place \( p \) got or lost the latest token, respectively.

Let \( \rho \) be a run of \( N \), \( i \geq 0 \), and let \( x \in \mathcal{P} \).

\[
eval_i(x) = \begin{cases} 
0 & \text{if } i = 0 \\
\text{time}_i(\rho) & \text{if } x = p^*, \ p \in \mu_i, \ p \notin \mu_{i-1} \\
\text{time}_i(\rho) & \text{if } x = p^i, \ p \notin \mu_i, \ p \in \mu_{i-1} \\
\eval_{i-1}(x) & \text{otherwise}
\end{cases}
\]

Validity of a \( \text{TNL} \) formula \( \varphi \) in a run \( \rho \) at point \( i \geq 0 \), denoted by \( (\rho, i) \models \varphi \), is now defined by induction on \( \varphi \) as usual:

1. \( (\rho, i) \models x - y \leq c \text{ iff } \eval_i(x) - \eval_i(y) \leq c \)
2. \( (\rho, i) \models p \text{ iff } p \in \mu_i \text{ for } p \in \mathcal{P} \)
3. \( (\rho, i) \not\models \bot \)
4. \( (\rho, i) \models (\varphi_1 \rightarrow \varphi_2) \text{ iff } (\rho, i) \models \varphi_1 \text{ implies } (\rho, i) \models \varphi_2 \)
5. \( (\rho, i) \models (\varphi_1 \mathbf{U} \varphi_2) \text{ iff there exists } j \geq i \text{ such that } (\rho, j) \models \varphi_2, \text{ and for all } k \text{ such that } i \leq k < j, (\rho, k) \models \varphi_1 \)

\( \rho \) satisfies \( \varphi \), denoted by \( \rho \models \varphi \), if \( (\rho, 0) \models \varphi \). \( \varphi \) 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 \( \diamond (p_{10} \land p_{10}^* - p_{10}^i \leq 8) \) is satisfiable if it can be reached within 8 time units, which is not the case (Note that \( \eval_i(p_{10}^i) = 0 \) for all \( i \)). \( \Box (p_{10}^i - p_{10}^* > 2) \) 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, $c \in Q$ and $\sim$ is a relation symbol from $\{\leq, <, =, >, \geq\}$. If $I$ is a set of inequalities, then $\text{var}(I)$ denotes the set of variables that $I$ contains; we say that $I$ is a set of inequalities over $\text{var}(I)$.

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 $c_i \in Q$, 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 $\varphi$, denoted by $\text{Cl}(\varphi)$, is the smallest set of inequalities such that for every inequality $x - y \leq c$ appearing in $\varphi$, both $x - y \leq c \in \text{Cl}(\varphi)$ and $x - y > c \in \text{Cl}(\varphi)$. A maximal consistent set of
\( \varphi \) is a maximal set \( F \subseteq \text{Cl}(\varphi) \) of inequalities which is consistent. Given any set \( I \) of inequalities, a complete extension \( \hat{I} \) of \( I \) and \( \varphi \) is any consistent set \( \hat{I} = I \cup I' \), such that \( I' \) is a maximal consistent set of \( \varphi \). \( CE(I, \varphi) \) denotes the set of all complete extensions of \( I \) and \( \varphi \). 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 \( \varphi \).

An atom is a pair \( \alpha = (\mu, I) \), where \( \mu \) is a marking and \( I \) is a set of inequalities. The initial atom is \( \alpha_0 = (\mu_0, \hat{I}_0) \), where \( \mu_0 \) is the initial marking of the net, and \( \hat{I}_0 \) is the unique complete extension of the following set \( I_0 \) of inequalities:

\[
I_0 = \{ "x - y = 0" \mid x, y \in \mathcal{P} \} \cup \\
\{ "t - x \geq \text{Eft}(t)" \mid t \in \text{enabled}(\mu_0), x \in \mathcal{P} \} \cup \\
\{ "t - x \leq \text{Lft}(t)" \mid t \in \text{enabled}(\mu_0), x \in \mathcal{P} \}
\]

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 \( \alpha' \) of an atom \( \alpha \) 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 \( I' = (I \setminus U) \) of inequalities over \( \text{var}(I) \setminus U \), such that the solution set of \( I' \) is equal to the solution set of \( I \), projected on \( \text{var}(I) \setminus U \). For example, if \( I = \{ "y - x \geq 2", \ "y - x \leq 7", \ "y - z < 3", \ "z - y \leq 11" \} \), then \( (I \setminus \{y\}) = \{ "x - z < 1", \ "z - y \leq 18" \} \).

As shown in [JM87], \( I' \) can be computed by a graph-based algorithm in time \( O(|\text{var}(I)|^2) \).

If \( \alpha = (\mu, I) \) is an atom, then \( \text{firable}(\alpha) = \{ t_f \mid t_f \in \text{enabled}(\mu), I \cup \{ \text{"t - } t_f \text{ \geq } 0\" \mid t \in \text{enabled}(\mu) \} \text{ is consistent} \} \) 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 \( \text{firable}(\alpha) \), \( \mu' = (\mu - \bullet t_f) \cup t_f \bullet \), and \( U_f = \{ p^c \mid p \in \bullet t_f \} \cup \{ p^* \mid p \in t_f \bullet \} \). We define the following sets of inequalities:
8.1. MODEL CHECKING FOR TIME NETS

- \( J_1 = I \cup \{ "t - t_f \geq 0" \mid t \in \text{enabled}(\mu) \} \)
- \( J_2 = (J_1 \setminus U_f) \)
- \( J_3 = J_2 \cup \{ "x - t_f = 0" \mid x \in U_f \} \)
- \( J_4 = (J_3 \setminus \{ t \mid t \notin \text{enabled}(\mu') \}) \)
- \( J_5 = J_4 \cup \{ "t - x \geq \text{Eft}(t)" \mid t \in \text{enabled}(\mu'), t \notin \text{enabled}(\mu), x \in U_f \} \)
  \( \cup \{ "t - x \leq \text{Lft}(t)" \mid t \in \text{enabled}(\mu'), t \notin \text{enabled}(\mu), x \in U_f \} \)
- \( J_6 = (J_5 \setminus \mathcal{P} \setminus \mathcal{P}_\varphi) \)

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 \( (I \setminus U) \) only once. For any \( \alpha \) and \( t_f \), \( J_6 \) is uniquely determined (up to isomorphism); we say \( J_6 \) is obtained by firing \( t_f \) from \( \alpha \). 

\( \alpha' = (\mu', I') \) is a successor atom of \( \alpha \), if \( I' \in CE(J_6, \varphi) \) for some \( J_6 \) obtained by firing an enabled transition \( t_f \) from \( \alpha \). An atom sequence \( \varrho \) is a finite or infinite sequence \( \varrho = (\alpha_0, \alpha_1, \alpha_2, \ldots) \), such that \( \alpha_0 \) is the initial atom and \( \alpha_{i+1} \) is a successor atom of \( \alpha_i \) 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 \( \varrho \), satisfaction of \( \varphi \) in \( \varrho (\varrho \models \varphi) \) is defined in an obvious way. Moreover, it can be proved that for any run \( \varrho \) there exists an atom sequence \( \varrho \) such that \( \varrho \models \varphi \) if \( \varrho \models \varphi \) and vice versa. Thus the question of whether there exists a run of \( N \) satisfying \( \varphi \) can be reduced to the question of whether there exists a satisfying atom sequence.

If \( \varphi \) 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. \( \alpha_1 = (\mu, I \cup \{ "x - y > 5" \}) \), \( \alpha_2 = (\mu, I \cup \{ "x - y > 17" \}) \), \( \alpha_3 = (\mu, I \cup \{ "x - y > 99" \}) \), 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 \( \alpha_i \) in which \( x - y \) surpasses a certain threshold value can be considered to be equivalent.
Let $\text{max}_\text{const}$ be the absolute value of the maximal constant appearing in any subformula of $\varphi$, and let $I$ be a set of inequalities. A time variable $x \in \mathcal{P}_x$ is called saturated in $I$, if there is no transition variable $t \in \text{var}(I)$ such that the set $I \cup \{t - x \leq \text{max}_\text{const}\}$ is consistent. For any two atoms $\alpha_1 = (\mu, I_1)$ and $\alpha_2 = (\mu, I_2)$, let $D = \{x \mid x$ is saturated in $I_1$ and $I_2\}$. $\alpha_1$ and $\alpha_2$ are equivalent, denoted by $\alpha_1 \simeq \alpha_2$, if $I_1 \cap \text{Cl}(\varphi) = I_2 \cap \text{Cl}(\varphi)$ and $(I_1 \setminus D) = (I_2 \setminus D)$, that is, if the same maximal consistent set of $\varphi$ 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 $\alpha_1$ and $\alpha_2$ such that $\alpha_1 \simeq \alpha_2$, and for any $\alpha'_1$ which is a successor of $\alpha_1$ there exists a successor $\alpha'_2$ of $\alpha_2$ 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 $\alpha$ reachable from the initial atom there is an equivalent atom $\alpha' \in G$, and for any atom sequence $\varphi = (\alpha_0, \alpha_1, \alpha_2, \ldots)$ there is a corresponding sequence $\varphi' = (\alpha'_0, \alpha'_1, \alpha'_2, \ldots)$ in $G$ such that $\alpha_i \simeq \alpha'_i (i \geq 0)$ and that $\varphi' \models \varphi$ iff $\varphi' \models \varphi$. $G$ 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 $\gamma$ of subformulas of $\varphi$, eliminating from this product all pairs $(\alpha, \gamma)$ inconsistent with $\varphi$, and decomposing the resulting graph into maximal strongly connected components. $\varphi$ is satisfiable by $N$ iff there is a self-fulfilling strong component, i.e. one which contains with any pair $(\alpha_1, \gamma_1)$ and any formula $(\varphi_1 \cup \varphi_2) \in \gamma_1$ also an pair $(\alpha_2, \gamma_2)$ 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
8.2. **STUBBORN ANALYSIS**

Equivalent (w.r.t. atomic propositions in $\varphi$) atom sequence in the reduced state space, and vice versa.

Given any atom $\alpha_0$ and transitions $t, t'$ enabled in $\alpha_0$, we say that $t$ is **independent from** $t'$ with respect to $\alpha$ and $\varphi$, if for any atom sequence $\varphi' = (\alpha_0, \alpha_1, \alpha_2, \ldots)$ such that $\alpha'_1$ is obtained by firing $t'$ from $\alpha_0$ there exists a stuttering equivalent (w.r.t. atomic propositions in $\varphi$) atom sequence $\varphi = (\alpha_0, \alpha_1, \alpha_2, \ldots)$ such that $\alpha_1$ is obtained by firing $t$ from $\alpha_0$. Otherwise, $t$ is called **dependent on** $t'$ (w.r.t. $\alpha, \varphi$). 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 $\alpha$ 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 $\alpha$. Therefore, subsequently we give an algorithm to compute an approximation, that is a set dependent($t, \alpha, \varphi$), or 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, 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 $\varphi$ 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 $N_2$ (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 $t'$ areirable in $\alpha$, 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 $N_2$, 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 $t_5$ 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 $t_5$. It takes at least $\text{Eft}(t_4) + \text{Eft}(t_6)$ (i.e. 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
$I \cup \{t_3 - t_4 \geq \text{diff}(t_4, t_6)\}$ is consistent, where $\text{diff}(t, t')$ is the minimal value of sums of earliest firing times in the paths from $t$ to $t'$ ($Eft(t)$ 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 $\Upsilon$ of transitions is necessary for $t$, if $\Upsilon = \{t' \mid p \in t' \circ \}$ for some $p \in \bullet t \setminus \mu$. necessary$^+(t, \alpha)$ is any set of transitions containing $t$ which is transitively closed under necessity, that is, for any $t' \in \text{necessary}^+(t, \alpha)$ such that $t'$ is disabled in $\mu$ there exists a set $\Upsilon$ of transitions necessary for $t'$ with $\Upsilon \subseteq \text{necessary}^+(t, \alpha)$. For example, necessary$^+(t_6, \alpha) = \{t_6, t_5, t_4\}$ in Fig. 7.1.

A transition $t_h$ in necessary$^+(t, \alpha)$ is harmful for $t_f$, if it is firable, and $I \cup \{t_f - t_h \geq \text{diff}(t_h, t)\}$ is consistent. If $t$ is in conflict with $t_f$, then all harmful transitions for $t_f$ in 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 $\varphi$. Usually, $\varphi$ contains only a few propositional and time variables. A transition $t$ is visible for $\varphi$ if $\bullet t \cup \bullet$ contains any place $p$ such that $p$ or $p^*$ or $p^\circ$ appears in $\varphi$. 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 $(p_1 \ U^\circ p_3)$ in the example net, thus the firing order between $t_2$ and $t_3$ is relevant for the evaluation of $(p_1 \ U^\circ p_3)$, 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 $\text{conflict}^+(t)$ be the set $\{t' \mid \bullet t' \cap \bullet t \neq \emptyset\} \cup \{t\}$, if $t$ is not visible, else $\text{conflict}^+(t)$ is $\{t' \mid \bullet t' \cap \bullet t \neq \emptyset\} \cup \{t' \mid t'$ is visible $\}$. Then dependent$(t_f, \alpha)$ is any set of transitions such that for every $t \in \text{conflict}^+(t_f)$ there exists a set necessary$^+(t, \alpha)$ such that all harmful transitions for $t_f$ in necessary$^+(t, \alpha)$ are contained in 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 ready$(\alpha)$ be a smallest set of firable transitions, such that for any $t_f \in \text{ready}(\alpha)$, dependent$(t_f, \alpha) \subseteq \text{ready}(\alpha)$.

Now, we can prove:
8.2. STUBBORN ANALYSIS

Theorem 8.2 For any atom sequence $\varphi \in G$ there exists a stuttering equivalent atom sequence $\varphi' = (\alpha_0, \alpha_1, \alpha_2, \ldots)$ such that for any $i \geq 0$, $\alpha_{i+1}$ is obtained by firing some transition in $\text{ready}(\alpha_i)$.

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, $\text{firable}(\alpha_0) = \{t_2, t_3, t_4, t_7\}$, whereas $\text{ready}(\alpha_0) = \{t_2\}$.

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 $\text{firable}(\mu)$, $\mu' = (\mu - \bullet t_f) \cup t_f \bullet$, and $U_f = \{p^* \mid p \in \bullet t_f\} \cup \{p^* \mid p \in t_f \bullet\}$. We define the following sets:

- $J_1 = I \cup \{t - t_f \geq 0 \mid t \in \text{ready}(\alpha)\}$
- $J_2 = (J_1 \ \parallel \ \ U_f)$
- $J_3 = J_2 \cup \{x - t_f = 0 \mid x \in U_f\}$
- $J_4 = (J_3 \ \parallel \ \{t \mid t \notin \text{enabled}(\mu')\})$

- For $t \in T$ and $x \in P$, $T_1(t, x) = \{t - x \geq \text{Eft}(t), \ t - x \leq \text{Lft}(t)\} \cup \{y^* \leq x \mid y \in \bullet t\}$

- $p \in P$ is called a candidate of true parents of a transition $t_c$ in $\mu$ and $I$, if $p \in \mu \cap \bullet t_c$ and for some enabled transition $t$ in $\mu$, $I \cup \{p^* \geq t + \text{diff}(t, t_c) - \text{Eft}(t_c)\}$ is consistent.

- $T_2 = \prod_{t \in \text{enabled}(\mu'), t \notin \text{enabled}(\mu)} T_1(t, p^*) \mid p$ is a candidate of true parents of $t$ in $\mu'$ and $J_4$, where $\prod$ represents the Cartesian product.

- $J_5 = \{\hat{I} \mid \hat{I} = J_4 \cup a_1 \cup a_2 \cup \cdots \cup a_i, (a_1, a_2, \ldots, a_i) \in T_2, \hat{I}$ is consistent $\}$

- $J_6 = \{(\hat{I} \ \parallel \ P \ \parallel \ P_\varphi \ \parallel \ D) \mid \hat{I} \in J_5\}$, where $D = \{p^* \mid p$ is a candidate of true parents of some disabled transition in $\mu'$ and $\hat{I}\}$. $\alpha' = (\mu', I)$ is a successor atom of $\alpha$, if $P \in \bigcup_{\hat{I} \in J_6} CE(\hat{I}, \varphi)$. 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 $\text{ready}(\alpha)$ is $O(|P| \cdot |T|^3)$, 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 $t_5$ and $t_{16}$ 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 $t_{12}$) 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 an 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 $\neg G^* [finish \rightarrow (\neg activity) \mathbf{U} (activity^* \rightarrow finish^* \leq max)]$ 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).

<table>
<thead>
<tr>
<th>$t_1$</th>
<th>$t_7$</th>
<th>$t_{13}$</th>
<th>$t_{19}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[0,0]$</td>
<td>$[0,0]$</td>
<td>$[16,24]$</td>
<td>$[0,0]$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$t_8$</td>
<td>$t_{14}$</td>
<td>$t_{20}$</td>
</tr>
<tr>
<td>$[260,300]$</td>
<td>$[0,0]$</td>
<td>$[0,10]$</td>
<td>$[0,0]$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>$t_9$</td>
<td>$t_{15}$</td>
<td>$t_{21}$</td>
</tr>
<tr>
<td>$[0,0]$</td>
<td>$[50,100]$</td>
<td>$[50,100]$</td>
<td>$[0,0]$</td>
</tr>
<tr>
<td>$t_4$</td>
<td>$t_{10}$</td>
<td>$t_{16}$</td>
<td>$t_{22}$</td>
</tr>
<tr>
<td>$[16,24]$</td>
<td>$[16,24]$</td>
<td>$[16,24]$</td>
<td>$[0,0]$</td>
</tr>
<tr>
<td>$t_5$</td>
<td>$t_{11}$</td>
<td>$t_{17}$</td>
<td>$t_{18}$</td>
</tr>
<tr>
<td>$[0,10]$</td>
<td>$[0,10]$</td>
<td>$[50,55]$</td>
<td>$[0,0]$</td>
</tr>
</tbody>
</table>

* $\uparrow [86+180,86i+220]$ 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 [TKI97, 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


<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>P</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>P</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\(P:\) previous value

(a)

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 \(\omega\)-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 Section 1.2.3. In Dill’s method, the specification of a circuit is given
9.1. TIMED TRACE THEORY

as an I/O-automaton, i.e. a finite automaton over an input alphabet \( \Sigma^\text{in} \) and output alphabet \( \Sigma^\text{out} \). 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 \( \omega \)-words are used instead of automata and \( \omega \)-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 \( \Sigma \) be the alphabet, and \( T \) the time domain: the set of non-negative rational numbers. Recall that for any \( a \in \Sigma \) and clock \( \tau \in T \), the tuple \((a, \tau)\) is called an event. In an asynchronous circuit, \((a, \tau)\) represents the change of the value of wire \( a \) at time \( \tau \).

A (timed) \( \omega \)-word \( \sigma \) over \( \Sigma \) 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 \( \sigma \) is infinite, then for every \( \tau \in T \) there exists an index \( i \) such that \( \tau_i > \tau \).
In this definition, $|z|$ denotes the length of $\omega$-word $x$. If $|z| = 0$, then $x$ is the empty word $\varepsilon$. For any finite $\omega$-word $x$, $\omega$-word $y$, and event $\varepsilon$, 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 $\omega$-word $z$.

Recall that a trace is the projection of a (timed) $\omega$-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 cannot 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 $\mathcal{M} \triangleq \{M_1, \ldots, M_n\}$ of I/O-modules, where $M_k \triangleq (\Sigma^i_k, \Sigma^{\text{out}}_k, T_k)$, where $\Sigma^i_k \triangleq \Sigma^i \cup \Sigma^{\text{out}}_k$, and $\Sigma^{\text{out}}_j \cap \Sigma^{\text{out}}_k = \emptyset$. 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 $\sigma \in \Sigma^{\text{out}}_k$ 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 $\sigma \in \Sigma^i_k$ for some $k \leq n$, such that $\mathcal{M} \setminus \sigma \not\models x$, and $M_k \models x$, but there is no $x' \triangleq y \circ (\sigma', \tau')$, where $\sigma' \in \Sigma^i_k$, 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 $\mathcal{M} \triangleq \{M_1, \ldots, M_n\}$ of I/O-modules, failure($\mathcal{M}$) is the set of all safety and timing failures of $\mathcal{M}$. $\mathcal{M}$ is failure-free if failure($\mathcal{M}$) = $\emptyset$.

Next, we define a conformance relation between a system consist-
9.1. TIMED TRACE THEORY

ing of a set of I/O-modules and a specification given as a single I/O-module. Consider a set \( \mathcal{M}_C = \{M_1, \ldots, M_n\} \) of I/O-modules, where \( M_i \triangleq (\Sigma^i, T_i) \), and an I/O-module \( M_S \triangleq (\Sigma_S^{in}, \Sigma_S^{out}, T_S) \) such that \( \Sigma_S^{in} \sqcup \Sigma_S^{out} \neq \Sigma^i \sqcup \Sigma^{out} \). 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 \( M_E \triangleq (\Sigma_E^{out}, \Sigma_E^{in}, T_E) \), 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 \( \sigma_i \in \Sigma^i \) 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 \( M \triangleq (\Sigma^i, \Sigma^{out}, T) \), \( M_1 \), and \( M_2 \), we say that \( M \) admits \( x \) \((M \models x)\), if project\( (x, \Sigma^i \cup \Sigma^{out}) \in T \). An I/O-module \( M \) can wait after trace \( x \) for at least \( \tau \) time units, denoted by \( M \models x \circ \tau \), 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 \( \tau \) 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 analyses 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 \( M_E_1 \) 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 cannot 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 \( \Sigma_E^{\text{in}} \subseteq \Sigma_S^{\text{out}} \) and \( \Sigma_E^{\text{out}} \supseteq \Sigma_S^{\text{in}} \) 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.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{fig9.2.png}
\caption{Examples of failures.}
\end{figure}

In the definition of conformance, only admissible environments \( (\Sigma_E^{\text{in}} \subseteq \Sigma_S^{\text{out}} \text{ and } \Sigma_E^{\text{out}} \supseteq \Sigma_S^{\text{in}}) \) are considered for the following reason. Consider the environment \( M_{E2} \) 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
9.1. TIMED TRACE THEORY

trace \((b, 0)\), I/O-module \(M_C\) may decide to produce an output \(x\) at time 1, but the environment \(M_E\) can not accept it (note that \(M_E\) must produce \(a\) before being ready to receive \(x\). On the other hand, \(\{M_S, M_E\}\) is failure-free (\(M_S\) does not accept the trace \((b, 0))\). We conclude that without admissibility constraints on the environment, \(M_C\) would not conform to \(M_S\). 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 \(M \triangleq (\Sigma^m, \Sigma^{out}, T)\) is the I/O-module \(M^m \triangleq (\Sigma^{out}, \Sigma^m, T)\); 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 \(\{M_1, M_3^m\}\) is failure-free and \(\{M_2, M_3^{m^m}\}\) is failure-free, then \(\{M_1, M_3^{m^m}\}\) is failure-free.

The proof can be found in [YZZ99].

From the hierarchy theorem, the following mirror theorem can be obtained. It gives a similar characterization of conformance as in [Dil89b]:

**Theorem 9.2** \(M_C\) conforms to \(M_S\) iff \(M_C \cup \{M_S^m\}\) is failure-free.

**Proof:** Assume that \(M_C \cup \{M_S^m\}\) has a failure. Then for the environment \(M_E = M^m_S\) we have that \(\{M_S, M_E\}\) is failure-free, but \(M_C \cup \{M_E\}\) is not failure-free, i.e., \(M_C\) does not conform to \(M_S\).

In the other direction, we have to show that failure-freeness of \(M_C \cup \{M_S^m\}\) implies that \(M_C\) conforms to \(M_S\). Since \(M_S\) is a specification for the system \(M_C, \Sigma_S^{in} \subseteq \bigcup_k \Sigma_k^{in} \subseteq \bigcup_k \Sigma_k^{out}\) and \(\Sigma_S^{out} \subseteq \bigcup_k \Sigma_k^{out}\). If \(M_C \cup \{M_S^m\}\) is failure-free, then the hierarchy lemma asserts that for any I/O-module \(M_E\) such that \(M_S = M_E\) and \(\{M_S, M_E\}\) is failure-free, \(M_C \cup \{M_E\}\) must also be failure-free. Thus, \(M_C\) 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 \(o \triangleq (\sigma_0, \tau_0), \sigma_0 \in \Sigma_S^{out}\), 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 $o' \triangleq (\sigma'_o, \tau'_i) \in \Sigma_S^{\text{out}}$ such that \( \{ M_S, M_C \} \models x \circ o' \), and

• if $M_C \models x \circ i$, then there exists a $i' \triangleq (\sigma'_i, \tau'_i) \in \Sigma_S^{\text{in}}$ 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 $\text{wire}$ 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 $((\text{wire}(t_1), \text{time}_1(\rho)), (\text{wire}(t_2), \text{time}_2(\rho)), \cdots)$. We also say that a net $N$ represents the I/O-module consisting of all traces generated by maximal runs of $N$.

![Diagram](image)

**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 \( w \) 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 detailed 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, \( c \in T \) and \( \sim \) is a relation symbol from \( \{<, \geq\} \). If \( I \) is a set of inequalities, then \( \text{var}(I) \) denotes the set of variables that \( I \) contains; we say that \( I \) is a set of inequalities over \( \text{var}(I) \). Let \( I \) be a set of inequalities over \( \{x_1, x_2, \ldots, x_m\} \). A feasible vector for \( I \) is a tuple \((c_1, c_2, \ldots, c_m)\) of constants \( c_i \in T \), 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.
9.2. ANALYSIS OF TIME PETRI NETS

If the net $N \triangleq (P, T, F, Eft, Lft, \mu^0)$ represents the I/O-module $M \triangleq (\Sigma^m, \Sigma^w, T)$, we denote this by $M = (\Sigma^m, \Sigma^w, N, \text{wire})$. An abstract state of the net is a pair $(\mu, I)$, 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 $(\mu^0, I_0)$, where $I_0 \triangleq \{ \text{"Eft}(t) \leq \underline{t} - v \leq \text{Lft}(t)" \mid t \in \text{enabled}(\mu^0) \}$. 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 $\sigma'$ of an abstract state $\sigma$ 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 $t_f \in \text{enabled}(\mu)$. Then, $\text{first}(\mu, t_f) \triangleq \{ \text{"}\underline{t} - \underline{t}_f \geq 0" \mid t \in \text{enabled}(\mu) \}$ is a set of inequalities describing that $t_f$ is the first transition which fires in $\mu$. \text{firable}(\sigma) \triangleq \{ t_f \mid t_f \in \text{enabled}(\mu), I \cup \text{first}(\mu, t_f) \text{ 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 \text{firable}(\sigma).
- $\mu'$ is the marking of $N$ obtained by firing transition $t_f$.
  That is, $\mu' \triangleq (\mu - t_f \bullet) \cup t_f \bullet$.
- $R$ is a set of newly enabled transitions obtained by the firing of $t_f$.
  That is, $R \triangleq \text{enabled}(\mu') - \text{enabled}(\mu - t_f)$.
- $J \triangleq \{ \text{"}\underline{t} - \underline{t}_out \geq \text{Eft}(t)" \mid t \in R \} \cup \{ \text{"}\underline{t} - \underline{t}_out \leq \text{Lft}(t)" \mid t \in R \}$.
- $J' \triangleq I \cup \text{first}(\mu, t_{out}) \cup J$.
- $D \triangleq \{ \underline{t} \mid t \text{ made some transition } t' \text{ enabled, and } t' \text{ is still enabled in } \mu' \}$.
- $I' \triangleq (J_3 \backslash \{ \underline{t} \mid t \notin \text{enabled}(\mu') \} - D)$

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 $\mu'$, 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 $t_f$.
CHAPTER 9. VERIFICATION WITH TIMED TRACES

We now describe how conformance can be checked, using this successor relation between abstract states. We consider a set \( \{M_0, M_1, \ldots, M_n\} \) of I/O-modules, where \( M \equiv (\Sigma_{in}^m, \Sigma_{out}^m, N_i, \text{wire}_i), N_i \equiv (P_i, T_i, Eft_i, Lift_i, \mu_i^\text{eq}), \) 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 \( \text{wire} \) instead of \( \text{wire}_i \), 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 \( \text{wire}_{m(t)}(t) \in \Sigma_{out}^{m(t)} \), and an input transition if \( \text{wire}_{m(t)}(t) \in \Sigma_{in}^{m(t)} \). If \( \sigma_i \equiv (\mu_i, I_i), i \leq n, \) are abstract states of the nets \( N_i \), and \( K \) is a set of inequalities, we say that \( s \equiv (\sigma_0, \ldots, \sigma_n, K) \) is an abstract state of the I/O-module set \( \{M_0, M_1, \ldots, M_n\} \).

The initial abstract state is \( s_0 \equiv (\sigma_0^0, \ldots, \sigma_n^0, \emptyset) \). We extend the definitions of \( \text{enabled}(\mu) \) and \( \text{firable}(\sigma) \) with respect to \( s \equiv (\sigma_0, \ldots, \sigma_n, K) \) as follows.

\[
\text{enabled}(s) \equiv \{ t \mid t \in \text{enabled}(\mu_{m(t)}) \}, \quad \text{and}
\]

\[
\text{globally-firable}(s) \equiv \{ t \mid t \in \text{enabled}(s), \text{first}(s, t) \cup \bigcup_{i=0}^n I_i \cup K \text{ is consistent} \},
\]

where \( \text{first}(s, t) \equiv \{ "t - t' \leq 0" \mid t' \in \text{enabled}(s) \} \). Furthermore, for an output transition \( t_O \) such that \( t_O \in \text{globally-firable}(s) \),

\[
\text{sync-trans}(t_O, s) \equiv \{ t \mid \text{wire}(t) = \text{wire}(t_O), t \in \text{globally-firable}(s) \}.
\]

When \( \{M_0, M_1, \ldots, M_n\} \) is at \( s \equiv (\sigma_0, \ldots, \sigma_n, K) \), it moves to \( s' \equiv (\sigma_0', \ldots, \sigma_n', K') \) with respect to \( t_O \in \text{globally-firable}(s) \) by firing all transitions in \( \text{sync-trans}(t_O, s) \).

- for \( 1 \leq i \leq n \)
  - if \( t \in \text{sync-trans}(t_O, s) \cap T_i \), then \( \sigma_i \rightarrow t \sigma_i' \), and
  - if \( \text{sync-trans}(t_O, s) \cap T_i = \emptyset \), then \( \sigma_i' = \sigma_i \).
- \( K' \equiv K \cup \{ "t - t' \leq 0" \mid t, t' \in \text{sync-trans}(t_O, s) \} \).

Let \( s \xrightarrow{t} s' \) denote this state transition relation of the I/O-module set.

For any transition \( t \) and abstract state \( \sigma \), the variable \( \text{parent}(t, \sigma) \) indicates which transition enabled \( t \). Formally, if \( \sigma \equiv (\mu, I), \sigma' \equiv (\mu', I'), \sigma \rightarrow t \sigma' \), and \( t' \in \text{enabled}(\sigma') \), then

\[
\text{parent}(t', \sigma') \equiv \begin{cases} t, & \text{if } t' \in \text{enabled}(\mu') - \text{enabled}(\mu - \bullet t) \\ \text{parent}(t', \sigma), & \text{otherwise} \end{cases}
\]

184
9.2. ANALYSIS OF TIME PETRI NETS

For a set $I$ of inequalities, let $earlier(x, y, I)$ be the predicate expressing that $solution\left(\{"x > y"\} \cup I\right) = \emptyset$, 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 $earlier(x, y, \bigcup_{i=0}^{m} I_i \cup K)$, where $s \triangleq (\sigma_0, \ldots, \sigma_n, K)$. Let $t \in M_i$, $\sigma_i \triangleq (\mu_i, I_i)$, and $t \in enabled(s)$.

- $earliest\_firing\_time(s, t) \triangleq parent(t, \sigma_i) + Ef(t)$, and
- $latest\_firing\_time(s, t) \triangleq parent(t, \sigma_i) + Lf(t)$.

A state $s \triangleq (\sigma_0, \ldots, \sigma_n, K)$ 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 wire$(t_o) \in \Sigma_j^m$, there exists an input transition $t_i$ such that wire$(t_i) = wire(t_o), t_i \in enabled(s)$, $earlier(earliest\_firing\_time(s, t_i), t_o, s)$ 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 $s \triangleq (\sigma_0, \ldots, \sigma_n, K)$ 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 $M_1, \ldots, M_n$ be represented by nets $N_1, \ldots, N_n$. 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, $failure(M_1, M_2, \ldots, M_n)$ is empty, iff every state which is reachable from the initial state of $(N_1, \ldots, N_n)$ is both safe and live. Therefore, the verification of conformance between I/O-modules can be done by traversing the state space of $(N_1, \ldots, N_n)$ 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 $\{M_1, \ldots, M_{k-1}, M_k, M_{k+1}, \ldots, M_n\}$ conforms to $M_S$, $\{M_{k_1}, \ldots, M_{k_m}\}$ conforms to $M_k$, and $(\bigcup_{j=1}^{m} \Sigma_{k_j} - \Sigma_k) \cap \bigcup_{j=1}^{m} \Sigma_j = \emptyset$, then $\{M_1, \ldots, M_{k-1}, M_{k_1}, \ldots, M_{k_m}, M_{k+1}, \ldots, M_n\}$ 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 $\{M_1, \ldots, M_{k-1}, M_k, M_{k+1}, \ldots, M_n\}$ conforms to $M_S$ and $\{M_{k_1}, \ldots, M_{k_m}\}$ conforms to $M_k$ is significantly smaller than the computation of whether $\{M_1, \ldots, M_{k-1}, M_{k_1}, \ldots, M_{k_m}, M_{k+1}, \ldots, M_n\}$ 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.

![Diagram](image)

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.
9.3. EXPERIMENTAL RESULTS

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>
<thead>
<tr>
<th></th>
<th>flat</th>
<th></th>
<th></th>
<th>hierarchical</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>size</td>
<td>states</td>
<td>CPU(s)</td>
<td>size</td>
<td>states</td>
<td>CPU(s)</td>
</tr>
<tr>
<td>C-element</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>p:30, t:34</td>
<td>51</td>
<td>1.3</td>
</tr>
<tr>
<td>ASM</td>
<td>p:78, t:90</td>
<td>391</td>
<td>81.8</td>
<td>p:34, t:34</td>
<td>58</td>
<td>1.2</td>
</tr>
<tr>
<td>Total</td>
<td>p:78, t:90</td>
<td>391</td>
<td>81.8</td>
<td>p:64, t:68</td>
<td>109</td>
<td>2.5</td>
</tr>
</tbody>
</table>

\[\dagger\]: "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 [Di89b] 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 $\mathcal{M}_C = \{M_1, \ldots, M_n\}$ of I/O-modules, and that an admissible specification for this system is given by the
I/O-module $M_S = (\Sigma_S^\mathrm{in}, \Sigma_S^\mathrm{out}, T_S)$. 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 $M(\Sigma^\mathrm{in}, \Sigma^\mathrm{out}, T)$, $M_1$, and $M_2$, the notion of admittance of $x$ by $M$ was defined as follows: $M \models x$, if $\text{project}(x, \Sigma^\mathrm{in} \cup \Sigma^\mathrm{out}) \in T$, 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 $M_2^m \cap M_1 \models ((w,5)(v,1))$ and $M_I \models ((a,1)(u,9))$. Note that in the latter example, $(a,1)$ is projected out since $a \notin \Sigma_I^\mathrm{in} \cup \Sigma_I^\mathrm{out}$.

Also, recall from the previous chapter that for a given set $\mathcal{M} = \{M_1, \cdots, M_n\}$ of I/O-modules, where $M_k = (\Sigma_k^\mathrm{in}, \Sigma_k^\mathrm{out}, T_k)$, a safety fail-
10.2. CHECKING SAFETY PROPERTIES

ure of $\mathcal{M}$ is a finite trace $x = y(w, \tau)$, where $w \in \Sigma^{\text{out}}$ for some $k \leq n$, such that $\mathcal{M} \models y$, $\mathcal{M} \not\models y \circ \tau$, and $\mathcal{M}_k \models x$, but $\mathcal{M} \not\models x$. We denote the class of all safety failures of $\mathcal{M}$ by $\text{failure}\mathcal{0}(\mathcal{M})$. A set $\mathcal{M}$ of I/O-modules is safety failure-free if $\text{failure}\mathcal{0}(\mathcal{M}) = \emptyset$. If $\mathcal{M} = \{\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2\}$ and $\mathcal{M}' = \{\mathcal{M}_1, \mathcal{M}_2\}$, we write $\text{failure}\mathcal{0}(\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2)$ or $\text{failure}\mathcal{0}(\mathcal{M}_0, \mathcal{M}')$ for $\text{failure}\mathcal{0}(\mathcal{M})$ etc.

Intuitively, a safety failure occurs if after trace $x$ one I/O-module $\mathcal{M}_k$ sends an output $w$ at time $\tau$, 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 $\mathcal{M}_i$ of $\mathcal{M}$ must admit some symbol $w'$ before $\tau$ (i.e., if $\mathcal{M}$ can not wait after $y$ for $\tau$ time units): In this case, the event $(w', \tau')$ will change the state of $\mathcal{M}_j$ and $\mathcal{M}$ may be able to accept $(w, \tau)$ after $y(w', \tau')$.

In the example in Fig. 10.2 below, the one-element trace $((u, 8))$ is a safety failure of $\{\mathcal{M}_E, \mathcal{M}_I\}$: note that $u$ is an output of $\mathcal{M}_I$ ($u \in \Sigma^{\text{out}}$), initially $\{\mathcal{M}_E, \mathcal{M}_I\}$ can wait for 8 time units ($\{\mathcal{M}_E, \mathcal{M}_I\} \models \varepsilon \circ 8$), and $\mathcal{M}_I$ can send symbol $u$ at time point 8 ($\mathcal{M}_I \models ((u, 8))$, but at that time $u$ is not enabled in $\mathcal{M}_E$ ($\mathcal{M}_E \not\models ((u, 8))$). On the other hand, $((u, 8))$ is not a safety failure of $\{\mathcal{M}_S^n, \mathcal{M}_I\}$, since $\mathcal{M}_S^n$ can not wait for 8 time units in the initial state ($\{\mathcal{M}_S^n, \mathcal{M}_I\} \not\models \varepsilon \circ 8$). Actually, in $\{\mathcal{M}_S^n, \mathcal{M}_I\}$ symbol $w$ will be sent by $\mathcal{M}_S^n$ before $\mathcal{M}_I$ can send $u$. After receiving $w$, no $u$-labelled transition is enabled in $\mathcal{M}_I$. Instead, $\mathcal{M}_I$ sends $v$, which is safely received by $\mathcal{M}_S^n$. Thus, there are no safety failures in $\{\mathcal{M}_S^n, \mathcal{M}_I\}$.

From this example it follows that for the class of all safety failures, the mirror property does not hold. Since in $\{\mathcal{M}_E, \mathcal{M}_S\}$ no outputs are enabled and safety failure only occur when an output is sent, $\text{failure}\mathcal{0}(\mathcal{M}_E, \mathcal{M}_S) = \emptyset$. As shown above, $((u, 8)) \in \text{failure}\mathcal{0}(\mathcal{M}_E, \mathcal{M}_I)$. From the definition of conformance, it follows that $\mathcal{M}_I$ does not conform to $\mathcal{M}_S$. However, $\text{failure}\mathcal{0}(\mathcal{M}_S^n, \mathcal{M}_I) = \emptyset$. 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 $\text{failure}\mathcal{0}$ holds.

In order to prove this theorem, we first prove two lemmata, which will be needed also in the next section. Let $\mathcal{M}_S = (\Sigma_S^{\text{in}}, \Sigma_S^{\text{out}}, T_S)$, $\mathcal{M}_E = \ldots$
Figure 10.2: An example in which the mirror property defined by failure0 does not hold.

\[
(\Sigma_{\text{in}}^E, \Sigma_{\text{out}}^E, T_E), \text{ and } \mathcal{M}_C = \{M_1, M_2, \ldots, M_n\} \text{ with } \Sigma_{\text{out}}^C = \bigcup_{k=1}^n \Sigma_{\text{out}}^k, \\
\Sigma_{\text{in}}^C = \bigcup_{k=1}^n \Sigma_{\text{in}}^k \setminus \Sigma_{\text{out}}^C \text{ such that } \Sigma_{\text{in}}^E \subseteq \Sigma_{\text{out}}^C, \Sigma_{\text{out}}^E \supseteq \Sigma_{\text{in}}^C, \Sigma_{\text{out}}^C \cap \Sigma_{\text{out}}^E = \emptyset, \\
\Sigma_{\text{in}}^C \subseteq \Sigma_{\text{out}}^C, \Sigma_{\text{in}}^C \supseteq \Sigma_{\text{in}}^C, \text{ and } \Sigma_{\text{in}}^C \cap \Sigma_{\text{out}}^C = \emptyset \text{ hold. For simplicity, we use } \mathcal{M}_C \text{ in several notations as if it were one I/O-module. We assume that those notations are extended appropriately for a set of I/O-modules.}
\]

**Lemma 10.2** If \( \mathcal{M}_C \) conforms to \( M_S \), then 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 failure\((M_E, \mathcal{M}_C) = \emptyset \) holds for any \( M_E \) such that failure\((M_E, M_S) = \emptyset \). Since failure\((M_S^m, M_S) = \emptyset \), \( \Sigma_{\text{in}}^m \subseteq \Sigma_{\text{in}}^S \), \( \Sigma_{\text{out}}^m \subseteq \Sigma_{\text{out}}^S \), and \( \Sigma_{\text{out}}^m \cap \Sigma_{\text{in}}^m = \emptyset \) hold, we can consider \( M_S^m \) as \( M_E \), and have failure\((M_S^m, \mathcal{M}_C) = \emptyset \). \( \square \)

**Lemma 10.3** Suppose that failure\((M_E, M_S) = \emptyset \), 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 \( u \in \Sigma_{\text{in}}^{\text{in}} \cup \Sigma_{\text{out}}^{\text{out}} \), \( M_S \models z \), 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, \( M_S \models z \circ \tau_u \) follows from \( M_S \models z \). If \( u \in \Sigma_{\text{in}}^{\text{out}} \), then \( u \in \Sigma_{\text{out}}^{\text{out}} \) holds from \( \Sigma_{\text{in}}^{\text{out}} \subseteq \Sigma_{\text{out}}^{\text{out}} \). \( M_S \models z \circ \tau_u \) is equivalent to \( M_S^m \models z \circ \tau_u \). Thus, from \( u \in \Sigma_{\text{out}}^{\text{out}} \), \( M_S^m \models z \circ \tau_u \), \( \mathcal{M}_C \models z(u, \tau_u) \), but \( z(u, \tau_u) \not\in M_S^m \) (from \( z(u, \tau_u) \not\in M_S \)), there must exist a safety failure in \( \{M_S^m, \mathcal{M}_C\} \), but this contradicts failure\((M_S^m, \mathcal{M}_C) = \emptyset \).
10.3 Timing Verification

Similarly, if \( u \in \Sigma^m_S \), a contradiction can be derived from the assumptions \( u \in O_E, M_S \models z_{o \tau}, M_E \models z(u, \tau'), M_S \not\models z(u, \tau), \) and 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 “\( MC \) conforms to \( M_S \Rightarrow \) failure \((M^m_S, M_C) = \emptyset \)” follows from Lemma 10.2. For the other direction “\( \text{failure} (M^m_S, M_C) = \emptyset \Rightarrow MC \) conforms to \( M^m_S \), we assume that \( MC \) does not conform to \( M_S \), even if failure \((M^m_S, M_C) = \emptyset \) holds. That is, there exists an I/O-module \( ME \) such that failure \((M_E, M_S) = \emptyset \) but failure \((M_E, M_C) \neq \emptyset \). Let \( x = y(w, \tau) \) be a safety failure of \( ME \) and \( MC \). There are two cases: \( w \in \Sigma^m_C \cap \Sigma^\text{out}_E \) and \( w \in \Sigma^\text{out}_C \).

In the first case, from \( w \in \Sigma^m_C \cap \Sigma^\text{out}_E \), \( M_E \models x \) but \( MC \not\models x \) hold as well as \( M_E \models y \circ \tau \) and \( MC \models y \circ \tau \). Since \( M_S \) is unbounded, \( M_S \models y \) holds from Lemma 10.3, and also \( M_S \models y \circ \tau \) holds. From this together with \( w \in \Sigma^\text{out}_E \), \( M_E \models x \), and failure \((M_E, M_S) = \emptyset \), we have \( M_S \models x \), i.e., \( M^m_S \models x \). From \( w \in \Sigma^m_C \subseteq \Sigma^m_S \), \( w \) is an output of \( M^m_S \). Hence, from \( MC \models y \circ \tau \), \( M^m_S \models x \), and failure \((M^m_S, M_C) = \emptyset \), we have \( MC \models x \), but this contradicts the hypothesis.

In the second case, if \( M_E \not\models x \), then it implies \( w \in \Sigma^m_C \), and the remaining proof is very similar to the above one (i.e., \( M_E \models x \), which is a contradiction, is derived from \( M_E \models y \circ \tau \), \( M_S \models x \), and failure \((M_E, M_S) = \emptyset \)). Otherwise, \( MC \not\models x \) holds, that is, for some \( M_k, M_j \in MC \) such that \( w \in \Sigma^\text{out}_k \cap \Sigma^m_j \), \( M_k \models x \) but \( M_j \not\models x \) hold. Also in this case, a contradiction (failure \((M^m_S, M_C) \neq \emptyset \)) is derived similarly from \( \{M^m_S, M_C\} \models y \circ \tau \), \( M_k \models x \), but \( \{M^m_S, M_C\} \not\models x \).

Since both cases lead to a contradiction, the assumption that \( MC \) 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 $\text{latest}(y, w, M)$ to be the maximal time $\tau$ such that $M \models y(w, \tau')$ for all $d \leq \tau' \leq \tau$, where $d$ is the duration of $y$. Intuitively, $\text{latest}(y, w, M)$ 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 $\text{latest}(y, w, M) = \infty$.

Given a set $\mathcal{M} = \{M_1, \ldots, M_n\}$ of I/O-modules, where $M_k = (\Sigma_k^{in}, \Sigma_k^{out}, T_k)$, a type $\mathbf{n}$ 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 $\mathbf{n}$ is satisfied.

**Condition 1:** $\mathcal{M} \models y \circ \tau$, but $\mathcal{M} \not\models x$ holds.

**Condition 2:** $\mathcal{M} \models y \circ \tau'$ for every $\tau'$ such that $M_k \models y(w, \tau')$, but there does not exist $\tau''$ such that $\mathcal{M} \models y(w, \tau'')$ holds.

**Condition 3:** There are no $j$, $w'$, and $\tau'$ such that $w' \in \Sigma_j^{out}$ and $\mathcal{M} \models y(w', \tau')$.

**Condition 4:** There are no $j$, $w'$, and $\tau'$ 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 $\tau'$ such that $w' \in \Sigma_j^{out}$, $\mathcal{M} \models y(w', \tau')$, and $\text{latest}(y, w', M_j) \leq \text{latest}(y, w, M_k)$.

**Condition 6:** There are no $j$, $w'$, and $\tau'$ such that $j \neq k$, $w' \in \Sigma_j^{out}$, $\mathcal{M} \models y(w', \tau')$, and $\text{latest}(y, w', M_j) \leq \text{latest}(y, w, M_k)$.

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 ($\tau$) when the input is expected. That is, for each 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 $M_2^n$. 
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 \( \mathcal{M} \models \neg(y \circ \tau) \)).

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 \( \{M_1, M_2\} \models \varepsilon \circ 6 \) 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 \( \text{latest}(\varepsilon, u, M_1) = \text{latest}(\varepsilon, w, M_1) = 6 \) (Intuitively, here we assume that if it doesn't 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 \( \text{latest}(\varepsilon, w, M_2) = 20 > 10 = \text{latest}(\varepsilon, w, M_1) \) (\( M_1 \) 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 \( \text{failures}(M_1, \ldots, M_n) \) be the union of all safety failures and type \( n \) timing failures of \( \{M_1, \ldots, M_n\} \). 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 \( \{M_{S'}^n, M_I\} \) has any safety failures, because \( w \) is accepted by the receiver, and \( M_S \) or \( M_{S'}^n \) can not wait for 6 time units. \( \{M_I, M_E\} \) has a safety failure \((u, 6)\), because \( M_I \models (u, 6) \), \( M_E \models \varepsilon \circ 6 \), but \((u, 6) \not\in M_E \). Concerning tim-
ing failures, \( \text{failure}_n(M_S, M_E) = \emptyset \) 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 \( M_S^n \) and \( M_I \), \( \text{failure}_n(M_S^n, M_I) = \emptyset \) holds for any \( n \), because \( w \) of \( M_S^n \) is the only input and it is given by \( M_S^n \) in time. Although \( \{M_I, M_E\} \) is timing failure-free, \( \text{failure}_n(M_I, M_E) \neq \emptyset \) for any \( n \) due to the safety failures of them (remember that \( \text{failure}_n \) includes safety failures). Hence, the mirror properties defined by \text{failure}, 2, 3 do not hold in this example, because for \( n \in \{1, 2, 3\} \), \( \text{failure}_n(M_S, M_E) = \emptyset \) and \( \text{failure}_n(M_I, M_E) \neq \emptyset \) (i.e., \( M_I \) does not conform to \( M_S \), but \( \text{failure}_n(M_S^n, M_I) = \emptyset \).

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 \( n \in \{2, 3, 4\} \), \( \text{failure}_n(M_S^n, M_I) = \text{failure}_n(M_S, M_E) = \emptyset \),
and \((w, 10)\) is in failure1\((M_S, M_E)\), failure6\((M^n_S, M_I)\), and failure5\((M^n_S, M_I)\). The latter two are due to \(\text{latest}(\varepsilon, w, M^n_S) > \text{latest}(\varepsilon, w, M_I)\). On the other hand, \((w, 10) \in \text{failure}(M_E, M_I)\) for \(n \in \{1, 2, 3, 4, 5, 6\}\). Therefore, \(M_I\) does not conform to \(M_S\) with respect to failure2, 3, 4. Hence, the mirror properties defined by failure2, 3, 4 do not hold in this example.

![Diagram showing mirror property for failure 3, 5.](image)

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 failure3, 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, failure5\((M_S, M_E) = \emptyset\) holds, because for the expected input \(w\) of \(M^n_S\), some other output \(u\) is produced in time (i.e., \(\text{latest}(\varepsilon, u, M^n_S) \leq \text{latest}(\varepsilon, w, M^n_S)\)), and after receiving \(u\), \(M_E\) immediately produce \(w\), which is also in time again from \(\text{latest}(\varepsilon, u, M^n_S) \leq \text{latest}(\varepsilon, w, M^n_S)\). This implies failure5\((M_S, M_E) = \emptyset\). Similarly, we can derive failure\((M^n_S, M_I) = \emptyset\) for \(n \in \{3, 5\}\). Hence, the mirror properties defined by failure3, 5 do not hold. failure\((M_S, M_E) \neq \emptyset\) for \(n \in \{1, 2\}\), because \(M_E\) can not produce \(w\) without receiving \(u\), and failure\((M_S, M_E) \neq \emptyset\) 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 failure6\(\left(M_E, M_S\right) = \emptyset\), failure6\(\left(M^n_S, M_C\right) = \emptyset\). Then \(M_E \cap M_C \models y\) 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 \(u \in \Sigma^n_S \cup \Sigma^\text{out}_S\), \(M_S \models z\), but \(M_S \not\models z(u, \tau_u)\). From \(M_E \cap M_C \models y\), we have \(M_E \cap M_C \models z(u, \tau_u)\). Again, first we show that \(M_S \models z \circ \tau_u\) holds. Assume that \(M_S \not\models z \circ \tau_u\). Then, there must exist \(u_1 \in \Sigma^n_S \cup \Sigma^\text{out}_S\) such that \(\text{latest}(z, u_1, M_S) < \tau_u\). If \(u_1 \in \Sigma^n_S\), then from failure6\(\left(M_E, M_S\right) = \emptyset\), \(M_E\) has an output \(u_2\) such that \(\text{latest}(z, u_2, M_E) \leq \text{latest}(z, u_1, M_S)\) (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 \(u_2 \in \Sigma^\text{out}_E\) as above (i.e., \(M_S\) may have an output \(u'_2\) without violating failure6\(\left(M_S, M_E\right) = \emptyset\)). This contradicts \(M_E \models z(u, \tau_u)\) because \(\text{latest}(z, u_2, M_E) < \tau_u\). If \(u_1 \in \Sigma^\text{out}_S\), then \(u_1\) is an input of \(M^n_S\). Thus, from failure6\(\left(M^n_S, M_C\right) = \emptyset\), \(M_C\) has \(u_2 \in \Sigma^\text{in}_C\) such that \(\text{latest}(z, u_1, M_C) \leq \text{latest}(z, u_1, M^n_S)\) (also holds only in case \(n = 6\)) but this contradicts \(M_C \models z(u, \tau_u)\) because \(\text{latest}(z, u_1, M_C) < \tau_u\). Hence, we have \(M_S \not\models z \circ \tau_u\). 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 “\(M_C\) conforms to \(M_S \Rightarrow\) failure6\(\left(M^n_S, M_C\right) = \emptyset\)” follows directly from Lemma 10.2. For the other direction “failure6\(\left(M^n_S, M_C\right) = \emptyset \Rightarrow M_C\) conforms to \(M^n_S\), we assume that \(M_C\) does not conform to \(M_S\), even if failure6\(\left(M^n_S, M_C\right) = \emptyset\). That is, there exists an I/O-module \(M_E\) such that failure6\(\left(M_E, M_S\right) = \emptyset\) but failure6\(\left(M_E, M_C\right) \neq \emptyset\). Let \(x = y(w, \tau) \in\) failure6\(\left(M_E, M_C\right)\). There are two cases: \(w \in \Sigma^n_C \cap \Sigma^\text{out}_E\) and \(w \in \Sigma^\text{out}_C\). 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 \(M_C \models \neg y \circ \tau, M_E \models x\), but \(x \notin M_C\). From failure6\(\left(M_E, M_S\right) = \emptyset\), \(w \in \Sigma^\text{out}_E\), and \(M_E \models x\), either \(M_S \models x\) or \(M_S \not\models y \circ \tau\) must hold. If \(M_S \models x\), then from \(w \in \Sigma^n_C \subseteq \Sigma^n_S\), \(M_C \models y \circ \tau\), failure6\(\left(M^n_S, M_C\right) = \emptyset\), we have \(M_C \models x\), but it contradicts the hypothesis. If \(M_S \not\models y \circ \tau\) holds, then from \(M_S \models y\) (from Lemma 10.3) for some \(w \in \Sigma^n_S \cup \Sigma^\text{out}_S\), \(\text{latest}(y, w, M_S) < \tau\) must hold. If \(w \in \Sigma^n_S\), then from failure6\(\left(M_S, M_E\right) = \emptyset\), for some \(w \in \Sigma^\text{out}_S\), \(\text{latest}(y, w, M_E) \leq \text{latest}(y, w, M_S)\) holds. (This holds only if \(n = 6\).) This contradicts \(M_E \models x\) from \(\text{latest}(y, w, M_E) < \tau\).

If \(w \in \Sigma^\text{out}_C\), then from failure6\(\left(M^n_S, M_C\right) = \emptyset\), for some \(w \in \Sigma^\text{out}_C\), \(\text{latest}(y, w, M_C) \leq \text{latest}(y, w, M_S)\) holds. (Again, this holds only if \(n = 6\).) This contradicts \(M_C \models \neg y \circ \tau\) from \(\text{latest}(y, w, M_C) < \tau\).
2. In the case that \( x \) is a type 6 timing failure:

From \( w \in \Sigma^u_S, \mathcal{M}_C \models x \) must hold. From failure6(\( M_S^u, \mathcal{M}_C \)) = \( \emptyset \), failure6(\( M_E, \mathcal{M}_C \)) \( \neq \emptyset \), and \( M_S \models y \) (from Lemma 10.3), for some \( w, w_\in \Sigma^u_S \), latest(\( y, w, M_S \)) \( \leq \) latest(\( y, w, \mathcal{M}_C \)) holds. (This holds even if \( n \neq 6 \).) From this and from failure6(\( M_E, M_S \)) = \( \emptyset \), for some \( w_\in \Sigma^u_E \), latest(\( y, w_\in, M_E \)) \( \leq \) latest(\( y, w, M_S \)) holds. (This holds only if \( n = 6 \).) From the above discussion, failure0(\( M_E, \mathcal{M}_C \)) = \( \emptyset \) holds. This and latest(\( y, w_\in, M_E \)) \( \leq \) latest(\( y, w, \mathcal{M}_C \)) derives \( \mathcal{M}_C \models y(w_\in, \tau_\in) \).

However, this means that we have \( w_\in \in \Sigma^u_E \), latest(\( y, w_\in, M_E \)) \( \leq \) latest(\( y, w, \mathcal{M}_C \)), and \( M_E \cap \mathcal{M}_C \models y(w_\in, \tau_\in) \), 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. \( \square \)

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 failure(\( M_S, M_E \)) = failure(\( M_S^u, M_I \)) = \( \emptyset \) and failure(\( M_E, M_I \)) = \( \emptyset \) 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 \( u \in \Sigma^u_S \) and \( w \in \Sigma^u_E \) 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.
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 $\mathcal{M} = \{M_1, \ldots, M_n\}$ of I/O-modules and a trace $x$, we call $w \in \Sigma^\text{out}_j$ a limited output, if $\text{latest}(x, w, M_j) \leq \text{latest}(x, w', M_k)$ holds for any $k$ and $w' \in \Sigma^\text{out}_k$. 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 $M_S^n$ for $\{M_S^n, M_I\}$ and a trace $\varepsilon$. In Fig. 10.7 $w$ is a limited output of $M_S^n$ for $\{M_S^n, M_I\}$ and a trace $\varepsilon$, but not a real limited output because $u$ of $M_I$ is also a limited output and it is not in $M_S^n$.

Given a set $\mathcal{M} = \{M_1, \ldots, M_n\}$ of I/O-modules, where $M_k =$
$(\Sigma_k^\text{in}, \Sigma_k^\text{out}, T_k)$, and another I/O-module $M_0 = (\Sigma_0^\text{in}, \Sigma_0^\text{out}, T_0)$, a pseudo timing failure of $\{M_0\} \cup \mathcal{M}$ with respect to $M_0$ is a finite trace $x = y(w, \tau)$, where $w \in \Sigma_0^\text{in}$, such that $\mathcal{M} \models y$, $M_0 \models x$, and $\text{latest}(y, w, M_0) = \text{latest}(y, u, M_0)$ 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 $\{M_S^\alpha, M_I\}$ with respect to $M_S^\alpha$, because $w$ is a real limited output of $M_S^\alpha$ and has the same latest value as the input $u$ (i.e., $\text{latest}(\epsilon, u, M_S^\alpha) = \text{latest}(\epsilon, w, M_S^\alpha)$ holds). On the other hand, $\{M_S^\alpha, M_I\}$ in Fig. 10.7 has no pseudo timing failure with respect to $M_S^\alpha$, because $w$ is not a real limited. Note that a pseudo timing failure is not included in the type 5 timing failures. Let $\text{failure}_P(M_0; M_C)$ be the union of $\text{failure}_5(M_0, M_C)$ and the set of pseudo timing failures of $\{M_0\} \cup M_C$ with respect to $M_0$. With these definitions, we can prove the following theorem.

**Theorem 10.8** Suppose that conformance is defined by $\text{failure}_5$. Then $M_C$ conforms to $M_S$ iff $\text{failure}_P(M_S^\alpha; M_C) = \emptyset$.

According to this theorem, conformance checking for $\text{failure}_5$ can be done by constructing $\text{failure}_P(M_S^\alpha; M_C)$. 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 \((2^{32})^{100} \approx 10^{1000}\). 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
CHAPTER 11. MODEL CHECKING OF PROGRAM RUNS

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](image)

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
- \( R \subseteq (S \times A) \cup (A \times S) \) is the causal dependency relation.

State action nets can be seen as 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
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 $\text{in}(m)$ expresses that a method $m$ 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 $\text{start}(m)$ and $\text{term}(m)$, 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 $X$, $F^+$ and $W$. 

- 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: $(\text{p if cond}(t, t'))$. The semantics of TIVs is defined by a thread identifier variable assignment (TIVA) which assigns a thread $\bar{x}(t)$ 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 $G_N = (\mathcal{S}, \mathcal{R}, s_I, s_T)$, where $\mathcal{S}$ is the set of all slices of $N$, and $\mathcal{R} = \mathcal{S} \times \mathcal{S}$.
describes the firing of actions. Here \((l, l') \in \mathcal{R}\), if an action \(a\) exists, which is activated in \(l\) and \(l'\) is generated from \(l\) by firing \(a\). The initial slice is \(s_I\), and the terminal slice is \(s_T\). The existence of initial and terminal slice is guaranteed since the slice-graph is a finite lattice.

For a given slice \(l\) and TIVA \(\mathcal{I}\), a formula \(p\) of the global tier can be either true \((l, \mathcal{I}) \models p\), or false \((l, \mathcal{I}) \not\models p\) or undefined \((l, \mathcal{I}) \not\vdash p\). Formally, these three relations are defined as follows:

- \((l, \mathcal{I}) \models t : p_i\) if a local state \(s \in l\) exists where \(\mathcal{I}(t) = tid(s)\), and \(s \models p_i\);
- \((l, \mathcal{I}) \not\models t : p_i\) if \(s \in l\) exists where \(\mathcal{I}(t) \neq tid(s)\), and \(s \not\models p_i\);
- \((l, \mathcal{I}) \not\models t : p_i\) otherwise.

- \((l, \mathcal{I}) \models \neg p\) if \((l, \mathcal{I}) \not\models p\);
- \((l, \mathcal{I}) \not\models \neg p\) if \((l, \mathcal{I}) \models p\);
- \((l, \mathcal{I}) \not\models \neg p\) otherwise.

- \((l, \mathcal{I}) \models (p \land q)\) if \((l, \mathcal{I}) \models p\) and \((l, \mathcal{I}) \models q\);
- \((l, \mathcal{I}) \not\models (p \land q)\) if \((l, \mathcal{I}) \not\models p\) or \((l, \mathcal{I}) \not\models q\);
- \((l, \mathcal{I}) \not\models (p \land q)\) otherwise.

- \((l, \mathcal{I}) \models X p\) if a slice \(l'\) exists where \((l, l') \in \mathcal{R}\) and \((l', \mathcal{I}) \models p\);
- \((l, \mathcal{I}) \not\models X p\) otherwise, i.e., if for all slices \(l'\) with \((l, l') \in \mathcal{R}\) either \((l', \mathcal{I}) \not\models p\) or \((l', \mathcal{I}) \not\vdash p\).

- \((l, \mathcal{I}) \models F^+ p\) if a slice \(l'\) exists such that \((l, l') \in \mathcal{R}^+\) and \((l', \mathcal{I}) \models p\);
- \((l, \mathcal{I}) \not\models F^+ p\) otherwise.

- \((l, \mathcal{I}) \models (p \lor^+ q)\) if a slice \(l'\) exists where \([l, l') \in \mathcal{R}^+,\) and \((l', \mathcal{I}) \models q\), and for all slices \(l''\) where \([l, l'') \in \mathcal{R}^+\) and \((l'', l') \in \mathcal{R}^+\), either \((l'', \mathcal{I}) \models p\), or \((l'', \mathcal{I}) \not\vdash p\);
- \((l, \mathcal{I}) \not\models (p \lor^+ q)\) otherwise.

- \((l, \mathcal{I}) \models (p \text{ if } \text{cond}(t, t'))\) if \((l, \mathcal{I}) \models p\) and \(\text{cond}(\mathcal{I}(t), \mathcal{I}(t'))\);
- \((l, \mathcal{I}) \not\models (p \text{ if } \text{cond}(t, t'))\) if \((l, \mathcal{I}) \not\models p\) and \(\text{cond}(\mathcal{I}(t), \mathcal{I}(t'))\);
- \((l, \mathcal{I}) \not\models (p \text{ if } \text{cond}(t, t'))\) otherwise.

According to this definition, all formulas \(X p\), \(F^+ p\) or \((p \lor^+ q)\) are either true or false in any slice \(l\) and TIVA \(\mathcal{I}\). A formula \(p\) is defined to be valid in an SAN \(N\), if \(G^+ p\) is true in the initial slice of \(N\), where \(G^+ p\) is \(\neg F^+ \neg p\) as usual (cf. Chapter 1). Since the truth value of \(G^+ p\) in any slice is not undefined, the notion of validity in state action nets is two valued: A formula \(p\) is valid in an SAN \(N\) if for all \(l\) and \(\mathcal{I}\) of \(N\) either \((l, \mathcal{I}) \models p\) or \((l, \mathcal{I}) \not\vdash p\).
For example, consider the formula

\[ ((t1: \text{in.get} \land \text{term.update}) \text{ before } \neg p \land \text{start.update}) \text{ if } t1 \neq t2) \]

Here \((p \text{ before } q) \triangleq \neg (p U^+ q)\) (for similar abbreviations, cf. Chapter 2). The formula specifies a precedence order on executions of the method \text{update} which has to be true if \text{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 \(\forall, \exists^t\) and \(G^+\), and replace negated subformulas by the respective dual formulas. For example, \(\neg F^+ p\) is replaced by \(G^+ \neg p\).

The slices are calculated bottom-up using the syntactical structure of the negated and transformed formula. This syntactical structure can be described as 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 \(G^+ p\), \(\text{all.next } p \lor (p \text{ before } q)\) is labelled by the transformed formula logically equivalent to \(\neg p\). 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.

\[ ((p \text{ before } q) \text{ if } t1=t2) \]

\[ t1: (\text{not (in.get and term.update)}) \]

\[ t2: (\text{in.put and start.update}) \]

\[ := \neg p \]

\[ := q \]

Figure 11.3: A formula tree

In [Fre96], model checking is done "post mortem", after generating the
11.2 SAFETY REQUIREMENT

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, cannot become true by the same slice and TIVA in a continuation of this run. Recall from Chapter 3, that a formula
\( \varphi \) is called a safety property, if for every SAN \( N \),

\[
N \models \varphi \quad \text{if} \quad (\forall M < N)(\exists N' > M) \quad N' \models \varphi
\]

In this definition, \( M \) is an initial subnet of \( N \), and \( N' \) is any continuation of this subnet \( M \). In other words, \( \varphi \) is a safety property if for every net not satisfying \( \varphi \) 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 \( \varphi \) 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 \( \varphi \) is a safety property, then the formula \( G^+ \varphi \), which is equivalent to \( \neg F^- \neg \varphi \), is a safety property. In general, \( F^+ \varphi \) is not a safety property: if in a net no slice exists in which \( \varphi \) is satisfied, then \( F^+ \varphi \) is false in the initial slice. However, if an extension is constructed in which \( \varphi \) is true, then \( F^+ \varphi \) is true in the initial slice. Similarly, the formula \( (\varphi \text{ before } \psi) \) is a safety property, but \( (\varphi \text{ U } \psi) \) is not a safety property.

As an example for a formula \( \varphi \) such that neither \( \varphi \) nor \( \neg \varphi \) is a safety property, consider \( \varphi = F^+ (p \land G^+ p) \). Since \( F^+ p \) is not a safety property, \( \varphi \) is not a safety property. The formula \( \neg \varphi = G^+ (\neg p \lor F^+ \neg p) \) is also not a safety property: \( G^+ F^+ p \) may be true in the initial slice of a net extension, because the terminal slice of the extension is the only one satisfying \( \neg p \). In all initial subnets, \( G^+ (\neg p \lor F^+ \neg p) \) is false, because in all of its slices \( p \) 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 \( N \)

\[
(l, \mathfrak{T}) \models N \varphi \quad \text{if} \quad \forall M > N(l, \mathfrak{T}) \models_M \varphi
\]

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 \( (\varphi \land \psi) \), \( (\varphi \lor \psi) \), \( X \varphi \), \( \Diamond \varphi' \), \( F^+ \varphi \)
and \((\varphi' \cup^+ \psi)\) are slice-stable if \(\varphi\) and \(\psi\) are slice-stable and \(\varphi'\) specifies a property of a finite part of the SAN. If \(\varphi\) 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.children.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
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_l1(s)` checks whether in the local state of the parameter the formula of the node is satisfied.

**Nodes labelled with** $(p \land q)$

Slices of a node labelled with $(p \land q)$ 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 TIVs assign
a different thread identification. This check is done by the method comp.
The merging of TIVs is done by the method merge.

For an on-the-fly generation of the slices of nodes labelled with \((p \land q)\) 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 \(F^+p\): If a
slice of the new extension satisfies \(p\), all slices which are predecessors of it
satisfy \(F^+p\). 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.

\[
\begin{align*}
\text{procedure simple_and}( & varp, varq, slp, slq, sl_{rec}) \\
& \text{for all } (l, \overline{\Sigma}) \in sl_p \text{ do} \\
& \quad \text{for all } (l', \overline{\Sigma}') \in sl_q \text{ do} \\
& \quad \quad \text{if } l = l' \text{ and comp}(\overline{\Sigma}, \overline{\Sigma}', var_p \cup var_q) \text{ then} \\
& \quad \quad \quad \overline{\Sigma}'' = \text{merge}(\overline{\Sigma}, var_p, \overline{\Sigma}', var_q); \\
& \quad \quad \quad sl_{rec} := sl_{rec} \cup (l, \overline{\Sigma}''); \\
\end{align*}
\]

\[
\begin{align*}
\text{procedure calc_and}( & n, \text{version}) \\
& n.child_p.slice_set := n.child_p.slice_set\cup \\
& n.child_p.queue[version]; \\
& n.child_q.slice_set := n.child_q.slice_set\cup \\
& n.child_q.queue[version]; \\
& \text{simple_and}(n.child_p.var, n.child_q.var, \\
& n.child_p.queue[version], n.child_q.slice_set, sl); \\
& \text{simple_and}(n.child_p.var, n.child_q.var, \\
& n.child_p.slice_set, n.child_q.queue[version], sl1); \\
& n.queue[version] := sl \cup sl1;
\end{align*}
\]

Figure 11.6: Calculation of intersection of slices
11.3. ON-THE-FLY MODEL CHECKING

Nodes labelled with \((p \uparrow q)\)

There is only one source of new slices for a node labelled \((p \uparrow q)\). The slices \(l\) and TIVAs \(\mathcal{T}\) of a new extension of the child node labelled with \(q\) can lead to new slices which are a subset of the predecessors of \(l\) w.r.t. \(\mathcal{A}\).

The subset is given by all slices \(l'\) and TIVAs \(\mathcal{T}'\) where \(\mathcal{T}'\) is a TIVA for all TIVs of the node and it is an extension of \(\mathcal{T}\). Furthermore, no slice \(l''\) exists between \(l\) and \(l'\) such that \((l'', \mathcal{T}')\) is an element of the child node labelled with \(\neg p\). To check this condition, it is not necessary to know the slices and TIVAs of the child node labelled with \(\neg p\) 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 \((p \uparrow q)\) on-the-fly.

To calculate the slices for the dual operator \(W^+\) is somewhat more complicated; for more information, the reader is referred to [FS98]. On the other hand, nodes labelled with formulas \(p\) if \(\text{cond}(t, t')\) 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(l1, san.final.slice[version], length);
    tiva_all := generate_tiva(empty, san.all_tiv[version], n.var);
    n.childp.slice_set := n.childp.slice_set \cup n.childq.queue[version];
    for all (l, \exists) \in n.childq.queue[version] do
        for all (l', e) \in sl_all \times tiva_all do
            if comp(\exists, \exists', n.childq.var) and (l', l) \in \mathbb{N}^+
                then
                    C := true;
                    for all (l'', \exists'') \in n.childp.slice_set do
                        if comp(\exists', \exists'', n.childp.var) and
                            (l'', l') \in \mathbb{N}^+ and (l', l) \in \mathbb{N}^+
                    then C := false;
                    if C and l \in sl_all then
                        n.queue[version] := n.queue[version] \cup (l', \exists')
    
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_e 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
to control that the splitting is done correctly, the user would specify

\( t_1: ((\text{start.sort} \land \text{is.e}) \rightarrow \text{even}(\text{input})) \land \\
      t_2: ((\text{start.sort} \land \neg \text{is.e}) \rightarrow \text{odd}(\text{input})). \)

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 $T_c$ must be terminated before $T_m$ is started:

$$(t: (\text{start.split}) \rightarrow (t': (\text{term.sort} \land \text{is}\_e) \land \text{before} t'': (\text{start.merge}))).$$

If the synchronization between $T_c$ 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:

$$\neg (t1: (\text{term.sort} \land \text{is}\_e) \land (t2: \neg (\text{term.sort} \land \text{is}\_e))).$$

The thread for merging can be advanced a single step with the specification

$$(t: (\text{start.merge}) \rightarrow \text{all.next} t: (\text{term.merge})).$$

The whole program can be advanced $n$ steps with the following formula:

$$(t: (\text{start.split}) \rightarrow \text{all.next}^n t': (\text{term.merge})).$$

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 11. MODEL CHECKING OF PROGRAM RUNS
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 [T85, 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 T Zi 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
12.1. DESCRIPTION OF THE SYSTEMS

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.

![Diagram of ABRIXAS-PTC system](image)

**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](image)

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-SCO (Subsystem Check-Out Equipment) 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 DESCRIPTION OF THE SYSTEMS

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 multimedia 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 suggests a layered architecture, where each layer is defined by service primitives to and from upper and lower layers, 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 sources, 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.
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-SCOPE. 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.
12.3  FORMALIZATION OF THE REQUIREMENTS

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 adaptation 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
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
12.3. FORMALIZATION OF THE REQUIREMENTS

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 adaption 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.

<table>
<thead>
<tr>
<th>At any given moment, it is possible to send telecommands for turning any consumer on or off. All switching operations necessary to activate or deactivate this consumer must be performed within a given time constant ( T_{\text{Switch Consumer}} ).</th>
</tr>
</thead>
</table>

**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 discharged 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.

<table>
<thead>
<tr>
<th>During the sun phase the battery is charged with a charge current of ( I_{\text{Charge}} ), until the amount of energy consumed during the last shade phase is recharged, and the minimal absolute pressure ( P_{\min abs} ) is reached or the sun phase ends.</th>
</tr>
</thead>
</table>

**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
12.3. FORMALIZATION OF THE REQUIREMENTS

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 $H_{\text{CCD,opt}}-H_{\text{CCD,tol}}$, the corresponding heater is switched on. If the temperature of the CCD is greater than $H_{\text{CCD,opt}}+H_{\text{CCD,tol}}$, 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 [(1 Tau_nextTC)] TCTIM ) ||| TIMCHK

SWITCHCONS = Tau_nextTC -> ((Com_PYRO_PWR_CONS_ON -> setTimSw
-> Swt_BS_ON_MAIN_ON -> Set_PYRO_PWR_MAIN_ON
-> Set_PYRO_PWR_MAIN_ON -> resTimSw -> SWITCHCONS)
|"| (Com_PYRO_PWR_CONS_OFF -> setTimSw -> Set_PYRO_PWR_MAIN_OFF
-> resTimSw -> SWITCHCONS)
|"| ... )

TIMCHK = elasTimSet -> errorSwitchTimer -> TIMCHK

TCTIM = Tau_nextTC -> setTimTick -> elasTimTick -> 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\_Switch\_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\_FWR\_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\_FWR\_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 elatTimSwt 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-
12.3. FORMALIZATION OF THE REQUIREMENTS

\[
\text{CHARGECONTROL} = \text{tau\_sun\_on} \rightarrow \text{setChargeControl} \\
\rightarrow \text{MAINCHARGE} (\text{false}, \text{false})
\]

\[
\text{MAINCHARGE}(\text{PressureOK}, \text{Recharged}) = \\
(\text{Evt\_I\_BATT\_MAIN\_IS\_I\_Charge} \\
\rightarrow \text{resetChargeControl} \rightarrow \text{MAINCHARGE}(\text{PressureOK}, \text{Recharged}))
\]

\[
(\text{Evt\_I\_BATT\_MAIN\_IS\_I\_Charge} \\
\rightarrow \text{setChargeControl} \rightarrow \text{MAINCHARGE}(\text{PressureOK}, \text{Recharged}))
\]

\[
(\text{Evt\_P\_BATT\_PRESS\_I\_OK} \rightarrow \\
\begin{cases}
\text{if Recharged then SUPPLEMENTARYCHARGE} \\
\text{else MAINCHARGE(true, Recharged)}
\end{cases}
\]

\[
(\text{Evt\_Recharged} \rightarrow \\
\begin{cases}
\text{if PressureOK then SUPPLEMENTARYCHARGE} \\
\text{else MAINCHARGE(PressureOK, true)}
\end{cases}
\]

\[
(\text{tau\_sun\_off} \rightarrow \text{DISCHARGECONTROL})
\]

Figure 12.10: CSP-code for the requirement in Fig. 12.7

If adequate 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\_I\_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 -> restTim1 -> TC2_CCD(ok.main.redu)
Evt_cold -> setTim1 -> TC2_CCD(cold.main.redu)
Evt-too_cold -> errorTooCold -> setTim1 -> TC2_CCD(temp.main.redu)
e1aTim1
  -> 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));

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
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

VVT-RT checks that certain bounds are respected. For example, if the first upper limit ($L_{	ext{CCD, opt}} + L_{	ext{CCD, tol}}$) is reached, the simulation component generates the event $\text{Evt\_warm}$. 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 ($\text{Evt\_too\_warm}$) 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 ($\text{Evt\_EXP\_PWR\_xxx\_ON}$), then the waste heat suffices to warm it. Process $\text{TCC\_CCD\_ON}$ checks that in this case the heat pads are never on. Event $\text{Evt\_const\_OK}$ causes the simulation to assume that the temperature of the operating camera is constant.
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 absolute 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 revealed: The software to store updated parameter tables or new software in the E²proms during the mission was too fast for the hardware. Although the software timing was correct with respect to the specification of the E²proms, the actually written bytes where sometimes different from the source. It turned out that the employed E²proms 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


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY
