Principles of Model Checking

Joost-Pieter Katoen
Formal Methods and Tools Group
University of Twente

Lecture Notes
"Systemvalidation" (course 214012)
2002/2003

Contents

Prologue

It is fair to state, that in this digital era correct systems for information processing are more valuable than gold.

H. Barendregt. The quest for correctness.
Images of SMC Research 1996, pages 39–58, 1996.

March 2002

Joost-Pieter Katoen
Enschede, The Netherlands

Chapter 1

System Verification

This chapter discusses the need for system verification for software as well as for hardware systems. It surveys the main techniques in systematic system verification such as testing, simulation, and deductive methods and introduces model checking as a valuable technique for defect detection.

1.1 Introduction

Our reliance on the functioning of ICT systems (Information and Communication Technology) is growing rapidly. These systems are becoming more and more complex and are massively encroaching on daily life via Internet and all kinds of embedded systems such as smartcards, hand-held computers, mobile phones and high-end television sets. In 1995 it was estimated that we are confronted with about 25 ICT-devices on a daily basis. Services like electronic banking and tele-shopping have become reality. The daily cash flow via Internet is about 10^{12} million US dollar. Roughly 20% of the product development costs of modern transportation devices such as cars, high-speed trains and airplanes is devoted to information processing systems. ICT systems are universal and omnipresent. They control the stock exchange market, form the heart of telephone switches, are crucial to Internet technology, and are vital for several kinds of medical systems. Our reliance on embedded systems makes their reliable operation of large social importance. Besides offering a good performance in terms like response times and processing capacity, the absence of annoying errors is one of the major quality indications.

It is all about money. We are annoyed when our mobile phone malfunctions, or when our video recorder reacts unexpectedly and wrongly to our issued commands. These software and hardware errors do not threathen our lives, but may have substantial financial consequences for the manufacturer. Correct

ICT systems are essential for the survival of a company. Dramatic examples are known. The bug in Intel's Pentium-II floating-point division unit in the early nineties caused a loss of about 475 million US dollar to replace faulty processors, and severely damaged Intel's reputation as a reliable chip manufacturer. The software error in a baggage handling system postponed the opening of Denver's airport for 9 months, at a loss of 1.1 million US dollar per day. 24 hours of failure of the worldwide on-line ticket reservation system of a large airplane company will cause its bankruptcy because of missed orders.

Figure 1.1: The Ariane-5 launch on June 4, 1996; it crashed 36 seconds after the launch due to a conversion of a 64-bit floating point into a 16-bit integer value

It is all about safety: errors can be catastrophic too. The fatal defects in the control software of the Ariane-5 missile (cf. Figure 1.1), the Mars Pathfinder and the airplanes of the Airbus family led to headlines in the newspapers all over the world and are renowned by now. Similar software is used for the process control of safety-critical systems such as chemical plants, nuclear power plants, traffic control and alert systems, and storm surge barriers. Clearly, bugs in such software can have disasterous consequences. For example, a software flaw in the control part of the radiation therapy machine Therac-25 caused the death of 6 cancer patients between 1985 and 1987 as they were exposed to an overdosis of radiation.

The increasing reliance of critical applications on information processing leads us to state:

The reliability of ICT systems is a key issue in the system design process.

The magnitude of ICT systems grows excessively, but their complexity grows rapidly too. ICT systems are no longer stand alone, but are typically embedded in a larger context, connecting and interacting with several other components and systems. They thus become much more vulnerable to errors – the number of defects grows exponentially with the number of interacting system components. In particular, phenomena such as concurrency and non-determinism that

are central to modelling interacting systems, turn out to be very hard to handle with standard techniques, both in software engineering and in hardware design. Their growing complexity, together with the pressure to drastically reduce system development time ("time-to-market"), makes the delivery of low-defect ICT systems an enormously challenging and complex activity.

1.2 Hard- and Software Verification

System verification techniques are applied to design ICT systems in a more reliable way. To put it bluntly, system verification is used to establish that the design or product under consideration possesses certain properties. The properties to be validated can be quite elementary, e.g., a system should never be able to reach a situation in which no progress can be made (a deadlock scenario), and are mostly obtained from the *system's specification*. This specification prescribes what the system has to do and what not, and thus constitutes the basis for any verification activity. A defect is found once the system does not fulfill one of the specification's properties. The system is considered to be "correct" whenever it satisfies all properties obtained from its specification. So correctness is always relative to a specification, and is not an absolute property of a system. A schematic view on verification is depicted in Figure 1.2.

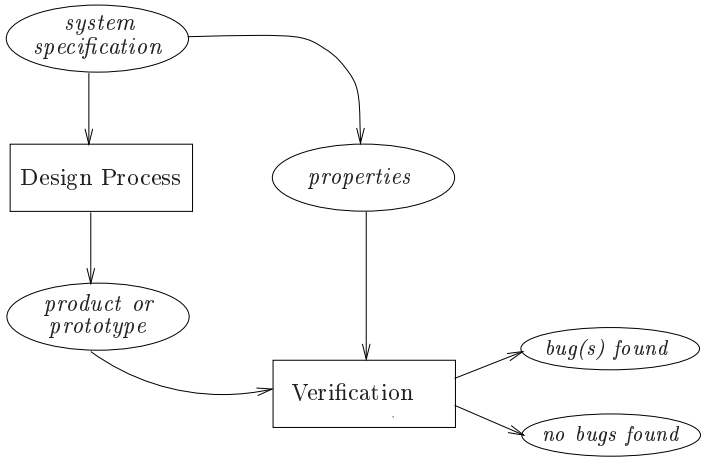


Figure 1.2: Schematic view of a posteriori system verification

This book deals with a verification technique, called model checking, that starts from a formal system specification. Before introducing this technique and discussing the role of formal specifications, we briefly review software and hardware verification.

1.2.1 Software Verification

Software verification techniques. Peer reviewing and testing are the major software verification techniques used in practice.

A *peer review* amounts to a software inspection carried out by a team of software engineers that preferably has not been involved in the development of the software under review. The uncompiled code is not executed, but analyzed completely statically. Empirical studies indicate that peer review provides an effective technique that catches between 31 and 93 percent of the defects with a median around 60%. While mostly applied in a rather ad-hoc manner, more dedicated types of peer review procedures, e.g., those that are focused at specific error-detection goals, are even more effective. Despite its almost complete manual nature, peer review is thus a rather useful technique. It is therefore not surprising that some form of peer review is used in almost 80% of all software engineering projects. Due to its static nature, experience has shown that subtle errors such as concurrency and algorithm defects are hard to catch using peer review.

Software testing constitutes a significant part of any software engineering project. Between 30% and 50% of the total software project costs are devoted to testing. As opposed to peer review that analyzes code statically without executing it, testing is a dynamic technique that actually runs the software. Testing takes the piece of software under consideration and provides its compiled code with inputs, called tests. Correctness is thus determined by forcing the software to traverse a set of execution paths, sequences of code statements representing a run of the software. Based on the observations during test execution, the actual output of the software is compared to the output as documented in the system specification. Although test generation and test execution can partly be automated, the comparison is usually performed by human beings. The main advantage of testing is that it can be applied to all sorts of software ranging from application software (e.g., e-business software) to compilers and operating systems. As exhaustive testing of all execution paths is practically infeasible, in practice only a small subset of these paths is treated. Testing can thus never be complete. That is to say, testing can only show the presence of errors, not their absence. Another problem with testing is to determine when to stop. Practically, it is hard, and mostly impossible, to indicate the intensity of testing to reach a certain defect density – the fraction of defects per number of uncommented code lines.

Studies have provided evidence that peer review and testing catch different classes of defects at different stages in the development cycle. They are therefore often used both. To increase the reliability of software, these software verification approaches are complemented with software process improvement techniques, structured design and specification methods (such as the Unified Modeling Language) and the use of version- and configuration management

control systems. Formal techniques are used, in one form or the other, in about 10 – 15% of all software projects. These techniques are discussed later on in this chapter.

Catching software errors: the sooner, the better. It is of great importance to locate software bugs. The slogan is: the sooner, the better. The costs of repairing a software flaw during maintenance are roughly 500 times higher than a fix after detection in an early design phase, cf. Figure 1.3. System verification should thus take place at an early stage in the design process.

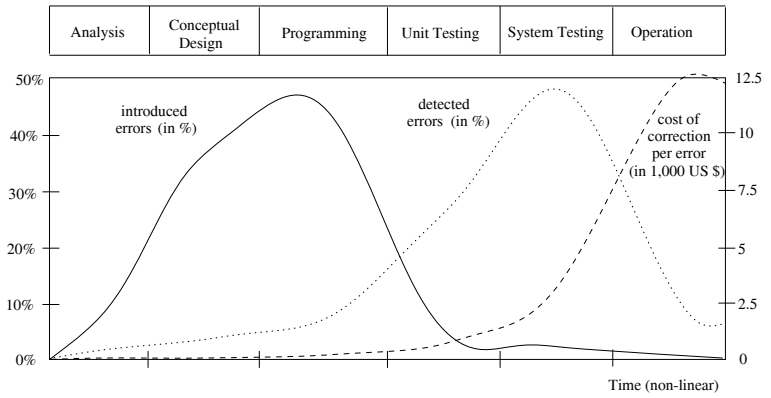


Figure 1.3: Software life-cycle and error introduction, detection and repair-costs [126]

About 50% of all defects are introduced during programming, the phase in which actual coding takes place. Whereas just 15% of all errors are detected in the initial design stages, most errors are found during testing. At the start of unit testing, which is oriented to discovering defects in the individual software modules that make up the system, a defect density of about 20 defects per 1,000 lines of (uncommented) code is typical. This has been reduced to about 6 defects per one thousand code lines at the start of system testing, where a collection of such modules that constitutes a real product is tested. On launching a new software release, the typical accepted software defect density is about one defect per 1,000 lines of code lines[1].

Errors are typically concentrated in a few software modules – about half of the modules are defect free, and about 80% of the defects arise in a small fraction (about 20%) of the modules – and often occur when interfacing modules. The repair of errors that are detected prior to testing can be done rather economically. The repair cost significantly increases from about 1,000 US dollar (per error repair) in unit testing to a maximum of about 12,500 US dollar when the defect is demonstrated during system operation only. It is of vital importance

[1] For some products this is much higher, though. Microsoft has acknowledged that Windows 95 contained at least 5,000 defects. Despite the fact that users were confronted with anomalous behaviour daily, Windows 95 was very successful.

to seek techniques that find defects as early as possible in the software design process: the costs to repair them are substantially lower, and their influence on the rest of the design is less substantial.

1.2.2 Hardware Verification

The importance of hardware verification. Preventing errors in hardware design is vital. Hardware is subject to high fabrication costs, fixing defects after delivery to customers is difficult, and quality expectations are high. Whereas software defects can be repaired by providing users with patches or updates – nowadays users even tend to anticipate and accept this – hardware bug fixes after delivery to customers are very difficult and mostly require refabrication and redistribution. This has immense economic consequences. The replacement of the faulty Pentium II processors caused Intel a loss of about 475 million US dollar. Moore's law – the number of logical gates in a circuit doubles every 18 months – has proven to be true in practice and is a major obstacle for producing correct hardware. Empirical studies have indicated that more than 50% of all ASICs (Application-Specific Integrated Circuit) do not work properly after initial design and fabrication. It is not surprising that chip manufacturers invest a lot in getting their designs right. Hardware verification is a well-established part of the design process. The design effort in a typical hardware design comprises only 27% of the total time spent on the chip; the rest is devoted to error detection and prevention.

Hardware verification techniques. Emulation, simulation and structural analysis are the major techniques used in hardware verification.

Structural analysis comprises several specific techniques such as synthesis, timing analysis, and equivalence checking that are not described in further detail here.

Emulation is a kind of testing. A re-configurable generic hardware system (the emulator) is configured such that it behaves like the circuit under consideration and is then extensively tested. Like with software testing, emulation amounts to providing a set of stimuli to the circuit and comparing the generated output with the expected output as laid down in the chip specification. To fully test the circuit, all possible input combinations in every possible system state should be examined. This is impractical and the number of tests needs to be reduced significantly, yielding potential undiscovered errors.

With *simulation*, a model of the circuit at hand is constructed and simulated. Models are typically provided using hardware description languages such as Verilog or VHDL that are both standardized by IEEE. Based on stimuli, execution paths of the chip model are examined using a simulator. These stimuli may be provided by a user, or by automated means such as a random generator.

A mismatch between the simulator's output and the output described in the specification determines the presence of errors. Simulation is like testing, but is applied to models. It suffers from the same limitations, though: the number of scenarios to be checked in a model to get full confidence goes beyond any reasonable subset of scenarios that can be examined in practice.

Simulation is the most popular hardware verification technique and is used in various design stages, e.g., at register-transfer level, gate and transistor level. Typically, about 21% of the verification time is spent on emulation, 63% on simulation and 16% on structural analysis. Besides these error detection techniques, *hardware testing* is needed to find fabrication faults resulting from layout defects in the fabrication process.

1.3 Formal Verification Techniques

In software and hardware design of complex systems, more time and effort is spent on verification than on construction. Techniques are sought to reduce and ease the verification efforts while increasing their coverage. Formal methods offer a large potential to obtain an early integration of verification in the design process, to provide more effective verification techniques, and to reduce the verification time. This section presents a survey of the main formal verification techniques.

1.3.1 Formal Methods

Let us first briefly discuss the role of formal methods. To put it in a nutshell, formal methods can be considered as "the applied mathematics for modeling and analyzing ICT systems". Their aim is to establish system correctness with mathematical rigour. Their great potential has led to an increasing use by engineers of formal methods for the verification of complex software and hardware systems. Besides, formal methods are one of the "highly recommended" verification techniques for software development of safety-critical systems according to e.g., the best practices standard by the IEC (International Electrotechnical Commission) and standards by the ESA (European Space Agency). The resulting report of an investigation by the FAA (Federal Aviation Authority) and NASA (North-Atlantic Space Agency) about the use of formal methods concludes that

> *"Formal methods should be part of the education of every computer scientist and software engineer, just as the appropriate branch of applied maths is a necessary part of the education of all other engineers."*

During the last decade, research in formal methods has led to the development of some very promising verification techniques that facilitate the early detection of defects. These techniques are accompanied by powerful software tools that can be used to automate various verification steps. Investigations have shown that formal verification procedures would have revealed the exposed defects in e.g., the Ariane-5 missile, Mars Pathfinder, Intel's Pentium II processor and the Therac-25 therapy radiation machine.

Roughly speaking, two brands of formal verification approaches can be distinguished: deductive and model-based methods.

With *deductive* methods, the correctness of systems is determined by properties in a mathematical theory. These properties are proven with the highest possible precision using tools such as theorem provers and proof checkers.

Model-based techniques are based on models describing the possible system behaviour in a mathematical precise and unambiguous manner. It turns out that – prior to any form of verification – the accurate modelling of systems often leads to the discovery of incompleteness, ambiguities and inconsistencies in informal system specifications. Such problems are usually only discovered at a much later stage of the design. The system models are accompanied by algorithms that systematically explore all states of the system model. This provides the basis for a whole range of verification techniques ranging from an exhaustive exploration (model checking) to experiments with a restrictive set of scenarios in the model (simulation), or in reality (testing). Due to unremitting improvements of underlying algorithms and data structures together with the availability of faster computers and larger computer memories, model-based techniques that a decade ago only worked for very simple examples, are nowadays applicable to realistic designs. As the starting-point of these techniques is a model of the system under consideration, we have as a given fact that:

> *Any verification using model-based techniques is only as good as the model of the system.*

1.3.2 Model-based Simulation

As argued before, one of the most well-known and practically used verification techniques is simulation. The software tool, the simulator, allows the user to study the system behaviour. This happens by determining, on the basis of the system model, how the system will react on certain specific scenarios (stimuli). These scenarios are either provided by the user or are generated by tools such as random scenario generators.

Simulation of formal models is typically useful for a first, quick assessment of

the quality of the (prototype) design. It is, however, less suited to find subtle errors because it is mostly impossible to generate all possible system scenarios, let alone simulate them all. The number of scenarios easily gets out of hand. For a mobile phone or remote control unit with a very restricted number, five say, of choices per step, the number of scenarios with 20 steps already equals 5^{20} (almost 100,000,000,000,000 possibilities). The exhaustive generation and simulation of scenarios is time-consuming and costly. In practice, only a small subset of all possible scenarios is actually examined. Consequently, there is a realistic risk that subtle defects remain hidden. Unexplored scenarios might reveal the fatal error.

Besides, after examining a restricted number of scenarios, it is hard to quantify the degree of the system's correctness. Quantitative measures of the number of errors left in the system are difficult to obtain, let alone indications about the probability that such errors will be discovered when the system is in operation. In practice, this often means that the criterion to stop simulation is simply when the project runs out of money!

1.3.3 Model Checking

Model checking is a verification technique that explores all possible system states in a brute force manner. Similar to a computer chess program that checks possible moves, a model checker, the software tool that performs the model checking, examines all possible system scenarios in a systematic manner. In this way, it can be shown that a given system model truly satisfies a certain property. It is a real challenge to examine the largest possible state spaces that can be treated with current means, i.e., processors and memories. State-of-the-art model checkers can handle state spaces of about 10^8–10^9 states with explicit state-space enumeration. Using clever algorithms and tailored data structures, larger state spaces (10^{20} upto even 10^{476} states) can be handled for specific problems. Even the subtle errors that remain undiscovered using emulation, testing and simulation can potentially be revealed using model checking.

Typical properties that can be checked using model checking are of a qualitative nature: Is the generated result ok?, Can the system reach a deadlock situation, e.g., when two concurrent programs are mutually waiting for each other and thus halt the entire system? But also timing properties can be checked: Can a deadlock occur within 1 hour after a system reset?, or Is a response always received within 8 minutes? Model checking requires a precise and unambiguous statement of the properties to be examined. As with making an accurate system model, this step often leads to the discovery of several ambiguities and inconsistencies in the informal documentation. For instance, the formalization of all system properties for a subset of the ISDN user part protocol revealed that 55% (!) of the original, informal system requirements were inconsistent.

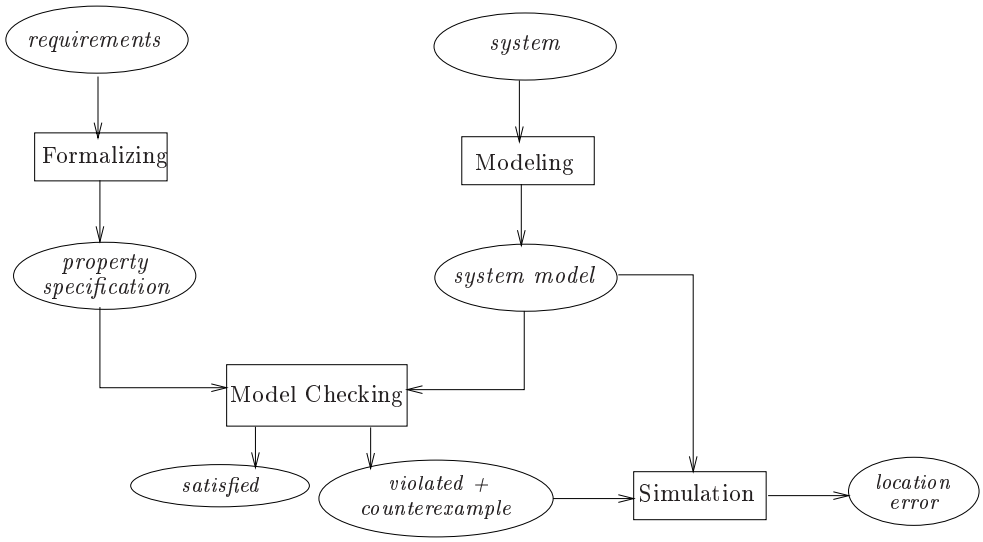


Figure 1.4: Schematic view of the model-checking approach

The system model is usually automatically generated from a model description that is specified in some appropriate dialect of programming languages like C or Java or hardware description languages such as Verilog or VHDL. Note that the property specification prescribes *what* the system should do, and what it should not do, whereas the model description addresses *how* the system behaves. The model checker examines all relevant system states to check whether they satisfy the desired property. If a state is encountered that violates the property under consideration, the model checker provides a counterexample that indicates how the model could reach the undesired state. The counterexample describes an execution path that leads from the initial system state to a state that violates the property being verified. With the help of a simulator, the user can replay the violating scenario, in this way obtaining useful debugging information, and adapt the model (or the property) accordingly, cf. Figure 1.4.

Model checking has been successfully applied to several ICT systems and their applications. For instance, deadlocks have been detected in on-line airline reservation systems, modern e-commerce protocols have been verified, and several studies of international IEEE standards for in-house communication of domestic appliances have led to significant improvements of the system specifications. Five previously undiscovered errors were identified in an execution module of the Deep Space 1 space-craft controller (cf. Figure 1.5), in one case identifying a major design flaw. A bug identical to one discovered by model checking escaped testing and caused a deadlock during a flight experiment 96 million kilometers from earth. In the Netherlands, model checking has revealed several serious design flaws in the control software of a storm surge barrier that protects the main port of Rotterdam for flooding.

Example 1.1. Most errors, such as the ones exposed in the Deep Space-1 space-

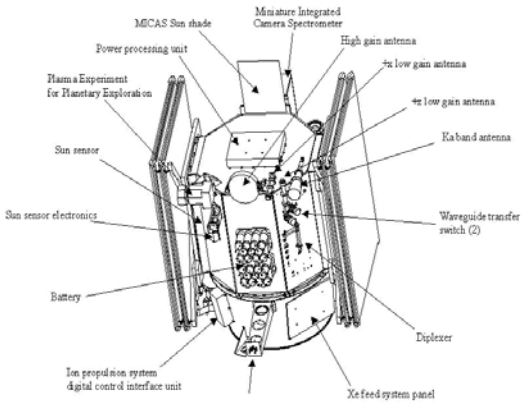


Figure 1.5: Modules of NASA's Deep Space 1 space-craft (launched in October 1998) have been thoroughly examined using model checking

craft, are concerned with classical concurrency errors. Unforeseen interleavings between processes may cause undesired events to happen. This is exemplified by analysing the following concurrent program, in which three processes, Inc, Dec *and* Reset *cooperate. They operate on the shared integer variable x with arbitrary initial value, that can be accessed (i.e., read), and modified (i.e., write) by each of the individual processes. The processes are:*

$$\begin{aligned}
\textbf{process}\ \mathsf{Inc} &= \textbf{while}\ \mathit{true}\ \textbf{do if}\ x < 200\ \textbf{then}\ x := x + 1\ \textbf{fi od}\\
\textbf{process}\ \mathsf{Dec} &= \textbf{while}\ \mathit{true}\ \textbf{do if}\ x > 0\ \textbf{then}\ x := x - 1\ \textbf{fi od}\\
\textbf{process}\ \mathsf{Reset} &= \textbf{while}\ \mathit{true}\ \textbf{do if}\ x = 200\ \textbf{then}\ x := 0\ \textbf{fi od}
\end{aligned}$$

Process Inc *increments x if its value is smaller than 200,* Dec *decrements x if its value is at least 1, and* Reset *resets x once it has reached the value 200. They all do so repetitively.*

Is the value of x always between (and including) 0 and 200? At first sight this seems to be true. A more thorough inspection, though, reveals that this is not the case. Suppose x equals 200. Process Dec *tests the value of x, and passes the test, as x exceeds 0. Then, control is taken over by process* Reset*. It tests the value of x, passes its test, and immediately resets x to zero. Then, control is returned to process* Dec *and this process decrements x by one, resulting in a negative value for x (viz. -1). Intuitively, we tend to interpret the tests on x and the assignments to x as being executed atomically, i.e., as a single step, whereas in reality this is (mostly) not the case. (End of example.)*

1.3.4 Model-based Testing

Whereas formal verification techniques such as simulation and model checking are based on a model description from which all possible system states can be generated, the well-established verification technique of testing is even applicable in cases where it is hard (e.g., in case of physical devices) or even impossible (e.g., when the model is proprietary) to obtain a system model. With testing, products or parts thereof are subject to scenarios to check whether there is an appropriate reaction.

An important parameter of testing is the extent to which access to the internal state of the system under test can be obtained. In *white box* testing the internal structure of an implementation can be fully accessed, while in *black box* testing the internal structure is completely hidden. In practice, intermediate scenarios are often encountered, referred to as *grey box* testing. The main advantage of testing is its broad applicability, in particular to final products and not restricted only to models. The drawback is comparable to simulation, as exhaustive testing is practically impossible. Like simulation, testing can show the presence of errors, not their absence.

Most currently available testing methods are rather ad-hoc and not very systematic. As a result, testing is a labour-intensive, error-prone and hardly manageable activity. In particular, the manual generation and maintenance of appropriate test cases causes a bottleneck. This leads to an increasing interest in model-based testing, as this allows a much more systematic treatment by mechanizing the generation of tests as well as the test execution phase. Analogous to model checking, the starting-point of model-based testing is a precise, unambiguous model description. With traditional testing methods such a basis is often absent. Based on this formal specification, test generation algorithms generate provably valid tests, i.e., tests that test what should be tested and no more than that. Testing tools support these algorithms, thus providing automatic, faster and less error-prone test generation. In this way, a test process in which the system under test and its formal model are the only required input parameters becomes possible, cf. Figure 1.6.

Model-based testing has important advantages also for *regression testing*. Regression testing involves checking the correct behaviour of a modified version of an existing system. This typically involves the adaptation, selection and repetition of existing tests. In model-based testing, a small modification of the system only requires an adaptation of its model, for which a new test suite (a set of tests) can be automatically generated.

In practice, model-based testing has been implemented in several software tools and has demonstrated its power in various case studies. For several systems, like embedded systems that control the exchange of information between high-end television sets and VCRs, errors have been found that remained undiscovered

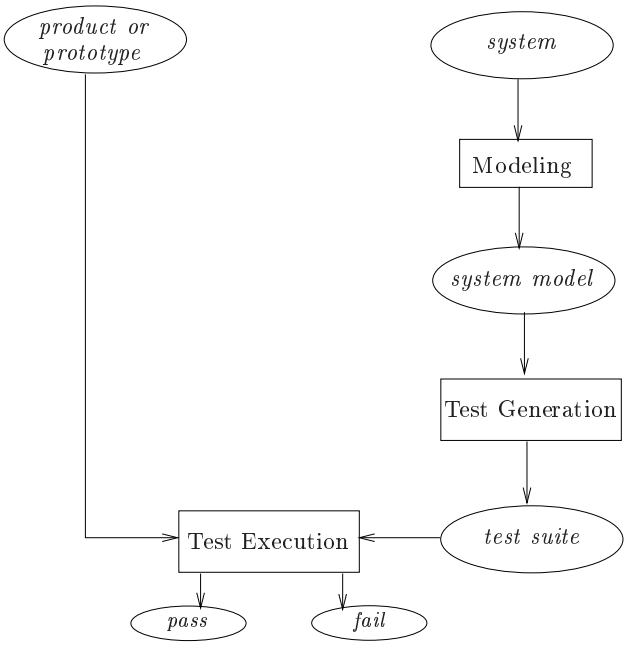


Figure 1.6: Schematic view of the model-based testing

with conventional testing techniques.

1.3.5 Theorem Proving

With deductive methods, the verification problem is interpreted as a mathematical theorem that typically has the form: *system specification* $\Rightarrow$ *desired property*. Trying to establish this result is referred to as *theorem proving*. In order to apply theorem proving, it is a prerequisite that the system specification has the form of a mathematical theory, or should be transformable into such form. Using a set of axioms (the basic theorems), a theorem prover (the software tool) tries to either construct a proof of the theorem by generating the intermediate proof steps, or to refute it. The axioms are either built-in or are provided by the user. Theorem provers are also called *proof assistants*. The general demand to prove theorems of a rather general type and the use of undecidable logics requires some user interaction. Different variants exist: highly automated, general-purpose proof assistants, and interactive programs with special-purpose capabilities.

Proof checkers are highly automated proof assistants that require a limited interaction with the user. The checker basically checks whether a user-provided proof suggestion is valid or not. The capability of proof checkers to generate intermediate proof steps in an automatic way is rather limited.

General-purpose proof assistants incorporate search components. In order to

reduce the search in theorem proving, interaction with the user takes place. The user may well be aware of what is the best strategy to conduct a proof. Usually, interactive proof assistants help in giving a proof by keeping track of the things still to be done and by providing hints on how these remaining (intermediate) theorems can be proven. Moreover, each proof step is automatically verified. Typically many small and detailed steps have to be taken in order to arrive at a fully proof-checked proof. The degree of interaction with the user is usually rather high. This is due to the fact that human beings see much more structure in their subject than logic or theorem provers do. This covers not only the content of the theorem, but also how it is used. In addition, the use of theorem provers or proof checkers requires much more scrunity than users are used to. Typically, human beings skip certain small parts of proofs ("trivial" or "analogous to") whereas the proof assistant requires these steps explicitly.

The main advantage of theorem proving is that it can deal with infinite state spaces (relying on proof principles such as structural induction) and can verify the validity of properties for arbitrary parameter values. Their main drawback is that the verification process is usually slow, error-prone and labour-intensive to apply. Besides, the mathematical logic used by the proof assistant requires a rather high degree of user expertise. Although some successful applications of theorem proving have been reported, like the thorough verification of smartcard software, these characteristics have restricted their use mainly to the academic world.

Logics for proof assistants. Logics used by proof assistants are variants of first-order logic and thus mostly undecidable. This logic ranges over an infinite set of variables and a set of function and predicate symbols of given arities. The arity specifies the number of arguments of a function or predicate symbol. A term is either a variable or of the form $f(t_1, \ldots, t_n)$ where f is a function symbol of arity n and t_i is a term. Constants can be viewed as functions of arity 0. A predicate is of the form $P(t_1, \ldots, t_n)$ where P is a predicate symbol of arity n and t_i is a term. Sentences in first-order predicate logic are either predicates, logical combinations of sentences, or existential or universal quantifications over sentences. In *typed* logics there is, in addition, a set of types and each variable, function and predicate symbol is typed. In these typed logics, quantifications are over types (rather than over variables), since the variables are typed. This enables to quantify over these types, which makes the logic more expressive than first-order predicate logic. Many theorem provers use *higher-order* logics: typed first-order logics where variables can range over function types or predicate types. There does not exist a canonical higher-order logic. Various syntactic and semantic differences do exist. Examples of prominent proof assistants for higher-order logics are PVS, Coq, HOL and Isabelle.

The internals of proof assistants. Most theorem provers have *algorithmic* and *search* components. The algorithmic components are used to apply proof rules and to obtain conclusions from this. Important techniques for this purpose are

natural deduction (e.g., from the validity of Φ and the validity of Ψ we may conclude the validity of $\Phi \wedge \Psi$), resolution, unification (a procedure which is used to match two terms with each other by providing all substitutions of variables under which two terms are equal), rewriting (where equalities are considered to be directed; in case a system of equations satisfies certain conditions, the application of these rules is guaranteed to yield a normal form).

These techniques are not sufficient to find the proof of a given theorem, even if the proof exists. The tool needs to have a strategy (a tactic) which describes how to proceed to find a proof. Such strategy may suggest to use rules backwards, starting with the sentence to be proven. This leads to goal-directed proof attempts. The strategies that humans use in order to find proofs are not formalized. Strategies that are used by theorem provers are simple strategies, e.g., based on breadth-first and depth-first search principles.

1.4 Characteristics of Model Checking

This book is devoted to the principles of model checking:

> *Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.*

The next chapters treat the elementary technical details of model checking. This section describes the process of model checking (how to use it), presents its main advantages and drawbacks, and discusses its role in the system development cycle.

1.4.1 The Model Checking Process

In applying model checking to a design the following different phases can be distinguished:

- *Modeling* phase:
  - model the system under consideration using the model description language of the model checker at hand
  - as a first sanity check and quick assessment of the model perform some simulations
  - formalise the property to be checked using the property specification language

- *Running* phase: run the model checker to check the validity of the property in the system model
- *Analysis* phase:
  - property satisfied? → check next property (if any)
  - property violated? →
    1. analyse generated counterexample by simulation
    2. refine the model, design, or property
    3. and repeat the entire procedure
  - out of memory? → try to reduce the model and try again

In addition to these steps, the entire verification should be planned, administered and organized. This is called *verification organization*. We discuss these phases of model checking in somewhat more detail below.

Modeling

The prerequisite inputs to model checking are a model of the system under consideration and a formal characterization of the property to be checked.

Models of systems describe the behaviour of systems in an accurate and unambiguous way. They are mostly expressed using *finite-state automata*, consisting of a finite set of states and a set of transitions. States comprise information about the current values of variables, the previously executed statement (e.g., a program counter), and the like. Transitions describe how the system evolves from one state into another. For realistic systems, finite-state automata are described using a model description language such as an appropriate dialect/extension of C, Java, VHDL, or the like. Modeling systems, in particular concurrent ones, at the right abstraction level is rather intricate and is really an art; it is treated in more detail in Chapter 2.

In order to improve the quality of the model, a simulation prior to the model checking can take place. Simulation can be used effectively to get rid of the simpler category of modelling errors. Eliminating these simpler errors before any form of thorough checking takes place may reduce the costly and time consuming verification effort.

To make a rigorous verification possible, properties should be described in a precise and unambiguous manner. This is typically done using a property specification language. We focus in particular on the use of a *temporal logic* as property specification language, a form of modal logic that is appropriate to specify relevant properties of ICT systems. In terms of mathematical logic, one checks that the system description is a model of a temporal logic formula. This

explains the term "model checking". Temporal logic is basically an extension of traditional propositional logic with operators that refer to the behaviour of systems over time. It allows for the specification of a broad range of relevant system properties such as: functional correctness (does the system do what it is supposed to do?), reachability (is it possible to end up in a deadlock state?), safety ("something bad never happens"), liveness ("something good will eventually happen"), fairness (does, under certain conditions, an event occur repeatedly?), and real-time properties (is the system acting in time?).

Although the aforementioned steps are often well understood, in practice it may be a serious problem to judge whether the formalized problem statement (model + properties) is an adequate description of the actual verification problem. This is also known as the *validation* problem. The complexity of the involved system as well as the lack of precision of the informal specification of the system's functionality may make it hard to answer this question satisfactorily. Verification and validation should not be confused. Verification amounts to check that the design satisfies the requirements that have been identified, i.e., verification is "check that we are building the thing right". In validation, it is checked whether the formal model is consistent with the informal conception of the design, i.e., validation is "check that we are verifying the right thing".

Running the model checker

The model checker first has to be initialised by appropriately setting the various options and directives that may be used to carry out the exhaustive verification. Subsequently, the actual model checking takes place. This is basically a solely algorithmic approach in which the validity of the property under consideration is checked in all states of the system model.

Analyzing the results

There are basically three possible outcomes: the specified property is either valid in the given model or not, or the model turns out to be too large to fit within the physical limits of the computer memory.

In case the property is valid, the following property can be checked, or, in case all properties have been checked, the model is concluded to possess all desired properties.

Whenever a property is falsified, the negative result may have different causes. There may be a *modeling error*, i.e., upon studying the error it is discovered that the model does not reflect the design of the system. This implies a correction of the model, and verification has to be restarted with the improved model. This

re-verification includes the verification of those properties that were checked before on the erroneous model and whose verification may be invalidated by the model correction! If the error analysis shows that there is no undue discrepancy between the design and its model, then either a *design error* has been exposed, or a *property error* has taken place. In case of a design error, the verification is concluded with a negative result, and the design (together with its model) has to be improved. It may be the case that upon studying the exposed error it is discovered that the property does not reflect the informal requirement that had to be validated. This implies a modification of the property, and a new verification of the model has to be carried out. As the model is not changed, no re-verification of properties that were checked before has to take place. The design is verified if and only if all properties have been checked with respect to a valid model.

Whenever the model is too large to be handled – state spaces of real-life systems may be many orders of magnitude larger than what can be stored by currently avalaible memories – there are various ways to proceed. A possibility is to apply techniques that try to exploit implicit regularities in the structure of the model. Examples of these techniques are the representation of state spaces using symbolic techniques such as binary decision diagrams or partial-order reduction. Alternatively, rigorous abstractions of the complete system model are used. These abstractions should preserve the (non-)validity of the properties that need to be checked. Often, abstractions can be obtained that are sufficiently small with respect to a single property. In that case, different abstractions need to be made for the model at hand. Another way of dealing with too large state spaces is to give up the precision of the verification result. The probabilistic verification approaches explore only part of the state space while making a (often negligible) sacrifice in the verification coverage. The most important state-space reduction strategies are discussed in Chapters 11 through 14 of this book.

Verification organization

The entire model-checking process should be well organized, well structured and well planned. Industrial applications of model checking have provided evidence that the use of version and configuration management is of particular relevance. During the verification process, for instance, different model descriptions are made describing different parts of the system, various versions of the verification models are available (e.g., due to abstraction) and plenty of verification parameters (e.g., model checking options) and results (diagnostic traces, statistics) are available. This information needs to be documented and maintained very carefully in order to manage a practical model checking process and to allow the reproduction of the experiments that were carried out.

1.4.2 Strengths and Weaknesses

The strengths of model checking.

- It is a *general* verification approach that is applicable to a wide range of applications such as embedded systems, software engineering, and hardware design.
- It supports *partial* verification, i.e., properties can be checked individually, thus allowing to focus on the essential properties first. No complete requirement specification is needed.
- It is not vulnerable to the likelihood with which an error is exposed; this contrasts with testing and simulation that are aimed at tracing the most probable defects.
- It provides *diagnostic information* in case a property is invalidated; this is very useful for debugging purposes.
- It is a potential *"push-button" technology*; the use of model checking requires neither a high degree of user-interaction nor a high degree of expertise.
- It enjoys a rapidly increasing *interest by industry*; several hardware companies started their in-house verification labs, job offers with required skills in model checking frequently appear, and commercial model checkers become available.
- It can be easily *integrated* in existing development cycles; its learning curve is not very steep, and empirical studies indicate that it may lead to shorter development times.
- It has a *sound and mathematical underpinning*; it is based on theory of graph algorithms, data structures, and logic.

The weaknesses of model checking.

- It is mainly appropriate to *control-intensive* applications and less suited for data-intensive applications as data typically ranges over infinite domains.
- Its applicability is subject to *decidability issues*; for infinite-state systems, or reasoning about abstract data types (that requires undecidable or semi-decidable logics), model checking is in general not effectively computable.
- It verifies a *system model*, and not the actual system (product or prototype) itself; any obtained result is thus as good as the system model. Complementary techniques such as testing are needed to find fabrication faults (for hardware) or coding errors (for software).

- It checks only *stated requirements*, i.e., there is no guarantee of completeness. The validity of properties that are not checked cannot be judged.
- It suffers from the *state-space explosion* problem, i.e., the number of states needed to model the system accurately may easily exceed the amount of available computer memory. Despite the development of several very effective methods to combat this problem (cf. Chapter 5), models of realistic systems may still be too large to fit in memory.
- Its usage requires some *expertise* in finding appropriate abstractions to obtain smaller system models and to state properties in the logical formalism used.
- It is not guaranteed to yield correct results: as any tool, a model checker may contain *software defects*.[2]
- It does not allow to check *generalizations*: in general, checking systems with an arbitrary number of components, or parameterized systems cannot be treated. Model checking can, however, suggest results for arbitrary parameters that may be verified using proof assistants.

We believe that one can never achieve absolute guaranteed correctness for systems of realistic size. Despite the above limitations we conclude that:

Model checking is an effective technique to expose potential design errors.

Thus, model checking can provide a significant increase in the level of confidence of a system design.

1.4.3 Integration in the Development Cycle

Model-checking hardware. With the notable exception of communication protocols, formal verification has been more successful for hardware than for software. There are several reasons for this. The high-quality standards in hardware design, together with the rather standard design levels (e.g., architecture, register transfer, gate, and transistor level) in its development cycle have paved the way to the smooth introduction of techniques such as model checking and theorem proving. Besides, the role of checking the correctness of circuits as part of the design process, together with the usage of finite-state models have been beneficial. Both theorem proving and model checking, and combinations thereof, have found their place in the hardware development process of companies like Cadence, Fujitsu, IBM, Intel and Motorola. Theorem proving is mostly

[2] Parts of the more advanced model-checking procedures have been formally proven correct using theorem provers to circumvent this.

used for checking data paths, signal processors and arithmetic units, whereas model checking is typically used for the control logic (one of the main sources of design flaws), controllers, and combinatorial circuits. Model checking is a widely accepted technique for the design phases that deal with circuits at the register transfer level and the gate level. At these levels, phenomena like nondeterminism, concurrency, and module composition – issues par excellence for model checking – play a prominent role. Recently, IBM reported the successful usage of model checking at multiple levels in their design trajectory, including the more abstract architecture level. Industrial experiments have provided evidence that model checking is no worse than random simulation in terms of time spent and that it is clearly superior in terms of coverage. The design of a memory bus adapter at IBM showed that 24% of all defects found were found with model checking, while 40% of these errors would most likely not have been found by simulation.

Model-checking software. Model checking has been successfully applied to a particular branch of software, namely the development of communication protocols. In such protocols, notions like atomicity, concurrency control and nondeterminism play a crucial role, and these phenomena can extremely well be captured by model checkers. Lucent Technologies, in earlier days known as AT&T, and IBM have played a prominent role in the practical development of (the first) model checkers. Several serious defects in communication protocols have been found using model checking. One of the most prominent example is perhaps the error that was adopted in the popular Needham-Schroeder encryption protocol that remained undetected for over 17 years.

As opposed to hardware design, software engineering has not exposed a "verification aware" discipline in the design process. Formal verification of (sequential) computer programs was begun in the late fourties by Turing and has emerged in the sixties with the pioneering works by Floyd and Hoare. Despite this early interest in correctness of software, these rigorous verification techniques have mainly been used by academia only. Although the rigid verification approach using axioms and proof rules never has become popular among software engineers, concepts like assertions, and, more importantly, pre- and postconditions have found their role in modern software engineering methods. In the popular "design by contract" software engineering philosophy, pre- and postconditions constitute the specification (i.e., the contract) to which the software under development should comply.

One of the main reasons for the conservative attitude of software engineers with respect to model checking has been the need for constructing a model of their software that is amenable to model checking. This obstacle has recently led to an increased interest by large companies and institutes such as Microsoft, NASA and Compaq to automatically generate compact models from programs written in programming languages such as `C`, `C++`, Java, or the like. First experiments with these techniques are very promising. It is expected that model-checking

techniques will rapidly be adopted on a wider scale by sofware engineers in the near future. According to Holzmann, the main developer of one of the leading model-checking tools SPIN, "within 5 years it (model checking) *will become* standard in most software development tools".

1.5 Bibliographic Notes

Model checking. Model checking originates from the independent work of two couples in the early eighties: of Clarke and Emerson [46] and Queille and Sifakis [155]. The term model checking was coined by Clarke and Emerson. The brute force examination of the entire state space in model checking can be considered as an extension of automated protocol validation techniques by Hajek [86] and West [187, 188]. While these earlier techiques were restricted to checking the absence of deadlocks or livelocks, model checking allows for the examination of broader classes of properties. Introductory papers on model checking can be found in [52, 48, 54, 138, 190]. The limitations of model checking were discussed by Apt and Kozen [15]. More information on model checking is available in the earlier books by Holzmann [96], McMillan [133] and Kurshan [115] and the recent works by Clarke, Peled and Grumberg [53], Huth and Ryan [102], and Bérard *et al.* [22]. The model-checking trajectory has recently been described by Brinksma and Ruys [159].

Software verification. Empirical data about software engineering is gathered by the Center for Empirically Based Software Engineering (`www.cebase.org`); their collected data about software defects has recently been summarised by Boehm and Basili [25]. The different characterisations of verification ("are we building the thing right?") and validation ("are we verifying the right thing?") originate from Boehm [26]. An overview of software testing is given by Whittaker [189]; books about software testing are by Myers [143] and Beizer [20]. Testing based on formal specifications has been studied extensively in the area of communication protocols. This has led to an international standard for conformance testing [103]. The use of software verification techniques by German software industry has been studied by Liggesmeyer *et al.* [126]. Books by Storey [172] and Leveson [122] describe techniques for developing safety-critical software and discuss the role of formal verification in this context. Rushby [158] addresses the role of formal methods for developing safety-critical software. The recent book of Peled [149] gives a detailed account on formal techniques for software reliability that includes testing, model checking and deductive methods.

Model-checking software. Model-checking communication protocols has become popular through the pioneering work by Holzmann [96, 97]. An interesting project at Bell Labs in which a model-checking team and a traditional design team worked on the design of part of the ISDN user part protocol has been reported by Holzmann [95]. In this large case study, 112 serious design flaws

were discovered while checking 145 formal properties in about 10,000 verification runs. Errors found by Clarke *et al.* [49] in the IEEE Futurebus+ standard (checking a model of more than 10^{30} states) has led to a substantial revision of the protocol by IEEE. Chan *et al.* [40] used model checking to verify the control software of a traffic control and alert system for airplanes. Recently, Staunstrup *et al.* [171] have reported the succesful model-checking of a train model consisting of 1,421 state machines comprising a state space of 10^{476} states. Lowe [127] discovered using model checking a flaw in the well-known Needham-Schroeder public key encryption algorithm. The usage of formal methods (that includes model checking) in the software development process of a safety-critical system within a Dutch software house is presented by Tretmans, Wijbrans and Chaudron [178]. The formal analysis of NASA's Mars Pathfinder and the Deep Space-1 space-craft are addressed by Havelund, Lowry and Penix [88], and Holzmann, Najm and Serhrouchini [98], respectively. The automated generation of abstract models amenable to model checking from programs written in programming languages such as `C`, `C++`, or Java has been pursued, for instance, by Godefroid [79], Dwyer, Hatcliff and co-workers [87], at Microsoft Research by Ball, Podelski and Rajamani [18] and at NASA Research by Havelund and Pressburger [89].

Model-checking hardware. Applying model checking to hardware originates from Browne *et al.* [33] analyzing some moderate size self-timed sequential circuits. Successful applications of (symbolic) model checking to large hardware systems have been first reported by Burch *et al.* [38] in the early nineties. They analyzed a synchronous pipeline circuit of approximately 10^{20} states. Overviews of formal hardware verification techniques can be found in works by Gupta [84], and the books by Yoeli [194] and Kropf [113]. The need for formal verification techniques for hardware verification has been advocated by, amongst others, Sangiovanni-Vincentelli, McGeer and Saldanha [161]. The integration of model checking techniques for error finding in the hardware development process at IBM has been recently described by Schlipf *et al.* [162] and Abarbanel-Vinov *et al.* [1]. They conclude that model checking is a powerful extension of the traditional verification process, and consider it as complementary to simulation/emulation.

Theorem proving. The Boyer-Moore proof assistant NQTHM (nowadays called ACL2) [29] for first-order logic has been used for hardware verification by, amongst others, Bronstein and Talcott [32] and Pierre [150]. Higher-order logics have recently become more popular for this purpose. Well-known higher-order logic proof assistants are Coq [101], HOL [137], Isabelle [148], Nuprl [57], and PVS [147]. A recent overview of checking proofs for distributed, concurrent systems has been provided by Groote, Monin and van de Pol [82]. The application of theorem proving to checking software systems is covered in the recent monograph by Schumann [**?**]. An interesting current trend is the application of proof assistants to the verification of smartcards, see, e.g., the recent work by Poll, van den Berg and Jacobs [153]. An impressive application of theorem proving is the correctness proof, consisting of about 28,000 proof obligations,

of the formal specification for the automatic train operatring system METEOR of the (first) driverless metro-line in Paris [19].

Chapter 2

Modelling Reactive Systems

2.0.1 Reactive Systems

Model checking is an appropriate technique for the verification of *reactive systems*. Reactive systems – this term was coined by Pnueli (1985) – are characterized by a continuous interaction with their environment. They typically continuously receive inputs (stimuli) from their environment and, usually within quite a short delay, react on these inputs (response). Reactive systems are usually technical, event/driven systems and are embedded in products of which they determine (part of) their functionality. Administrative systems are typically not reactive systems. Examples of reactive systems are operating systems, aircraft control systems, embedded systems, communication protocols, and process control software. For instance, a control program of a chemical plant regularly receives control signals, like indications about temperature and pressure, at several time instants. Based on this information, the program decides to turn on the heating elements, to switch off a pump, or the like. As soon as a dangerous situation is anticipated, e.g, the pressure in the tank exceeds certain thresholds, the control software needs to take appropriate action. Reactive systems tend to be complex: the nature of their interaction with the environment can be intricate and they typically have a distributed and concurrent nature. Aspects like non-determinism, distribution and concurrency play an important role in reactive systems.

Chapter 3

Linear Temporal Logic

This chapter discusses the appropriateness of temporal logic as a specification language for formalizing properties of reactive systems. This motivation takes place starting from the perspective of classical assertional verification of sequential programs. Propositional *linear* temporal logic is presented, covering both its theoretical underpinnings – syntax, semantics and axiomatization – and its practical use as a property specification language.

3.1 The Need for Temporal Logic

Reactive systems are characterized by a *continuous* interaction with the environment. For instance, an operating system or a coffee machine interacts with its environment (i.e., the user) and usually performs actions, such as fetching a file or producing coffee. A reactive system thus reacts to a stimulus from the environment. After receiving a stimulus and producing an accompanying reaction, a reactive system is – once more – able to interact. Typically, this repetitive behavior does not terminate. The continuous character of interaction in reactive systems differs from the traditional view of sequential programs. Sequential programs can be considered as functions mapping inputs onto outputs. After receiving input, a sequential program can generate output(s) and will eventually terminate after a finite number of computation steps, provided it does not end up in an endless loop. In the next section, we briefly explain assertional verification of sequential programs and argue that the different nature of reactive systems requires a modification of this approach. In particular, it does not suffice to use propositional (or predicate) logic to express the relevant properties but a logic is needed that is able to cope with the dynamic evolution between states during the program execution.

3.1.1 Propositional Logic

Properties of sequential programs can be expressed in terms of *pre- and postconditions*. A precondition describes the set of interesting start states, i.e., the allowed input(s), while a postcondition describes the set of desired final states, i.e., the required output(s). A pair of pre- and postcondition thus specifies the required input-output behavior of a sequential program. For instance, the simple program $x := x + 1\,;\; x := x + 1$ establishes the postcondition "x equals 2" once started in a state satisfying the precondition "x equals 0"; it thus transforms the value of x from 0 into 2. Similarly, the program **while** $x < 200$**do** $x := x + 1$**od** establishes the postcondition "x equals 200" when started in some state satisfying the precondition "x is at most 200".

The syntax of *propositional logic*, a language that constitutes the basis for specifying pre- and postconditions, is defined as follows. The basic elements are *atomic propositions*, i.e., statements that cannot be further broken down. Atomic propositions are the most elementary statements that can be made. That is to say, atomic propositions intuitively express atomic facts about the states of the system under consideration. Examples of atomic propositions are "x equals 0", or "x is smaller than 200" for some given integer variable x. Other examples are "it is raining" or "there are currently no customers in the shop". The (possibly empty) finite set of atomic propositions is referred to by AP, and elements of AP are typically denoted by p, q and r. We will not dwell upon a precise definition of AP here and simply postulate its existence.

Note that the choice of the set AP of atomic propositions is an important one; it determines the most basic statements that can be expressed about the program under investigation. Fixing the set AP can therefore be regarded as a first step of abstraction. If one, for instance, decides not to allow some program variables to be referred to in AP, then no property can be stated that refers to these variables, and consequently no such property can be checked.

Each atomic proposition ranges over the boolean types `tt` (true) and `ff` (false). In the setting of this book, it is assumed that for each state in the system under consideration, it is known which atomic propositions do hold and which ones do not. This is established by an interpretation function, or also called labelling. Let S be a set of states.

Definition 3.1. (**Interpretation function**)
Interpretation function $Label : S \longrightarrow 2^{AP}$ assigns to each state s the atomic propositions $Label(s)$ that are valid in s.

The function *Label* indicates which atomic propositions are valid for any state. If for state s we have $Label(s) = \varnothing$ it means that no proposition is valid in s. A state s for which the proposition p is valid, i.e., $p \in Label(s)$, is referred to

as a p-state. Alternatively, a mapping from $AP \longrightarrow 2^S$ can be defined which provides for each atomic proposition p, the set of p-states. Such function is typically called a valuation.

Note that no constraints are put on the labelling *Label* of states with atomic propositions. No further interpretation of propositions is thus given. For instance, if proposition "$x = 1$" belongs to *Label*(s), it does not mean that proposition "$x \neq 0$" also belongs to *Label*(s).

Definition 3.2. (**Syntax of propositional logic**)
Let p be an atomic proposition. Formulas in *propositional logic* satisfy the following rules:

1. p is a formula.
2. If Φ is a formula, then $\neg\,\Phi$ is a formula.
3. If Φ and Ψ are formulas, then $\Phi \vee \Psi$ is a formula.
4. Anything else is not a formula.

Here, $\neg$ denotes negation and $\vee$ denotes disjunction. The boolean connectives $\wedge$ (conjunction), $\Rightarrow$ (implication) and $\Leftrightarrow$ (equivalence) are defined by:

$$\begin{aligned}
\Phi \wedge \Psi &\equiv \neg(\neg\Phi \vee \neg\Psi) \\
\Phi \Rightarrow \Psi &\equiv \neg\Phi \vee \Psi \\
\Phi \Leftrightarrow \Psi &\equiv (\Phi \Rightarrow \Psi) \wedge (\Psi \Rightarrow \Phi) \\
\text{true} &\equiv \Phi \vee \neg\Phi \\
\text{false} &\equiv \neg\,\text{true}
\end{aligned}$$

The precedence of the above operators is as follows: $\neg$ binds the strongest, followed by $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$. Parentheses are omitted whenever appropriate. For example, $p \Rightarrow q \wedge r$ is interpreted as $p \Rightarrow (q \wedge r)$. We further assume that $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$ are all right-associative, e.g., $p \wedge (q \wedge r)$ may be written as $p \wedge q \wedge r$.

The meaning of formulas in propositional logic is defined by means of a *satisfaction relation* (denoted $\models$) between a state s and a formula Φ. $(s, \Phi) \in \models$ is denoted by the following infix notation: $s \models \Phi$. The concept is that $s \models \Phi$ if and only if Φ is valid (i.e., evaluates to `tt`) in state s.

Definition 3.3. (**Semantics of propositional logic**)
Let p be an atomic proposition, s a state and Φ, Ψ formulas in propositional

logic. The satisfaction relation $\models$ is defined by:

$$\begin{array}{ll} s \models p & \text{iff } p \in \mathit{Label}(s) \\ s \models \neg\, \Phi & \text{iff not } (s \models \Phi) \\ s \models \Phi \ \vee \ \Psi & \text{iff } (s \models \Phi) \text{ or } (s \models \Psi) \end{array}$$

Note that the logical operators on the right-hand side of a defining equation are denoted by "not" and "or" to distinguish them from the logical operators $\neg$ and $\vee$ in propositional logic. If $s \models \Phi$ we say that state s satisfies Φ.

The validity of propositional logic is static in the sense that the truth value of Φ is only determined by the labelling of the current state (such as propositions about the current value of program variables). The labelling and the truth value of Φ are regarded as *immutable*; there is no way in which they are related to the validity of Φ in other states, like the previous or the next state.

Tautologies are statements that are satisfied by all states, e.g., $\Phi \ \vee \ \neg\, \Phi$. Similarly, the formula $\neg(\Phi \ \vee \ \neg\, \Phi)$ is a *contradiction*, i.e., it is satisfied by no state. For state s with $p, q \notin \mathit{Label}(s)$ we have $s \models (p \ \Rightarrow \ q)$; $p \ \Rightarrow \ q$ is however, neither a tautology nor a contradiction.

3.1.2 Assertional Verification of Sequential Programs

Pre- and postconditions are usually expressed in *predicate logic*, basically an extension of propositional logic in which

(i) atomic propositions are refined into expressions built up from variables, constants, function and predicate symbols, and

(ii) existential ($\exists$) and universal $\forall$ quantification over variables is allowed.

For example, if $\mathsf{getElt}(a, i)$ selects the i-th element of array a, postcondition $(\forall 0 < i \leqslant K.\, \mathsf{getElt}(a, i) \geqslant 0)$ asserts that all elements of array a of length K are non-negative. As predicate logic does not play a further role in this book, we omit a formal treatment of its syntax and semantics.

For precondition Φ and postcondition Ψ, the correctness of sequential program statement S is denoted by:

$$\{\, \Phi \,\}\, S\, \{\, \Psi \,\} \tag{3.1}$$

This is also referred to as a Hoare triple [93]. There are two possible interpretations of Hoare triples:

- (3.1) is *partially correct* if any terminating computation of S that starts in a state satisfying Φ, terminates in a state satisfying Ψ.
- (3.1) is *totally correct* if any computation of S that starts in a state satisfying Φ, *terminates* and finishes in a state satisfying Ψ.

So, in the case of partial correctness no statements are made about computations of S that diverge, i.e., that do not terminate. In the course of our discussion here, it suffices to assume that each program terminates, i.e., we do not distinguish between partial and total correctness in this section.

To prove that a sequential program meets its pre- and postcondition, a *mathematical proof system* is used consisting of a set of axioms and proof rules. To illustrate the approach, a proof system for simple deterministic sequential programs will be given. These programs are constructed according to the BNF grammar:

$$S ::= \textbf{skip} \mid x := E \mid S\,;\, S \mid \textbf{if } B \textbf{ then } S \textbf{ else } S \textbf{ fi} \mid \textbf{while } B \textbf{ do } S \textbf{ od}$$

where **skip** stands for no operation, $x := E$ for the assignment of the value of expression E to variable x (where x and E are assumed to be equally typed), $S\,;\, S$ for the sequential composition of statements, and the latter two for alternative composition and iteration (where B denotes a boolean expression), respectively. For simplicity, we do not consider variable declarations.

Example 3.1. Consider the following program S:

$$p := 0;\ i := k;\ \textbf{while } i > 0 \textbf{ do } p := p + n;\ i := i - 1 \textbf{ od}$$

The Hoare triple $\{\, k \geqslant 0 \wedge n \geqslant 0 \,\}\ S\ \{\, p = k{\cdot}n \,\}$ asserts that S computes for any two non-negative integers n and k, the product $k{\cdot}n$. (End of example.)

A proof system for our example programming language is given in Table 3.1. The proof rules should be read as follows: if all conditions indicated above the straight line are valid, then the conclusion below the line is valid. For rules with a condition true only the conclusion is indicated; these proof rules are called axioms. Axioms are thus proof rules that always can be applied.

The proof rule for the skip statement states that under any condition, if formula Φ is valid before the statement, then it is valid afterwards. According to the axiom for assignment, one starts with the postcondition Φ and determines by substitution the precondition $\Phi[x := k]$. $\Phi[x := k]$ roughly means Φ where all occurrences of x are replaced by k. The rule for sequential composition uses an intermediate predicate Φ' that characterizes the final state of S and the starting state of S'. The rule for alternative composition uses the boolean B whose value

Axiom for **skip**	$\{\,\Phi\,\}\ \mathbf{skip}\ \{\,\Phi\,\}$
Axiom for assignment	$\{\,\Phi[x := k]\,\}\ x := k\ \{\,\Phi\,\}$
Sequential composition	$\dfrac{\{\,\Phi\,\}\ S\ \{\,\Phi'\,\},\ \ \{\,\Phi'\,\}\ S'\ \{\,\Psi\,\}}{\{\,\Phi\,\}\ S\,;\ S'\ \{\,\Psi\,\}}$
Alternative	$\dfrac{\{\,\Phi \wedge B\,\}\ S\ \{\,\Psi\,\},\ \ \{\,\Phi \wedge \neg B\,\}\ S'\ \{\,\Psi\,\}}{\{\,\Phi\,\}\ \mathbf{if}\ B\ \mathbf{then}\ S\ \mathbf{else}\ S'\ \mathbf{fi}\ \{\,\Psi\,\}}$
Iteration	$\dfrac{\{\,\Phi \wedge B\,\}\ S\ \{\,\Phi\,\}}{\{\,\Phi\,\}\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}\ \{\,\Phi \wedge \neg B\,\}}$
Consequence	$\dfrac{\Phi \Rightarrow \Phi',\ \{\,\Phi'\,\}\ S\ \{\,\Psi'\,\},\ \Psi' \Rightarrow \Psi}{\{\,\Phi\,\}\ S\ \{\,\Psi\,\}}$

Table 3.1: Proof system for partial correctness of sequential programs

determines whether S or S' is executed. The proof rule for iteration states that predicate Φ holds after the termination of $\mathbf{while}\,B\,\mathbf{do}\,S\,\mathbf{od}$, provided Φ remains valid during each execution of body S. Hence, Φ is called an *invariant*.

All rules discussed so far are *syntax-oriented*: a proof rule is associated to each syntactical construct. This differs from the consequence rule which establishes the connection between program verification and logics. The consequence rule allows the strengthening of preconditions and the weakening of postconditions. In this way, it may facilitate the application of other proof rules. It should be noted, though, that proving implications like $\Phi \Rightarrow \Phi'$ is in general undecidable.

The presented proof system allows to prove the correctness of composed programs with respect to their pre- and postcondition by considering program parts only. The procedure of starting the proof from a postcondition is usually applied successively to parts of the program such that finally the precondition of the entire program can be proven. For instance, the proof rule for sequential composition allows the correctness of the composed program $S\,;\,S'$ to be established by considering the pre- and postconditions of its components S and S'. Proof systems that exhibit this property are called *compositional*.

3.1.3 Assertional Verification of Parallel Programs

Let us now consider the extension to parallelism. For statements S and S' let the construct $S \parallel S'$ denote their parallel composition. Major aim of applying assertional verification to parallel programs is to obtain a proof rule such as:

$$\frac{\{\,\Phi\,\}\,S\,\{\,\Psi\,\},\ \ \{\,\Phi'\,\}\,S'\,\{\,\Psi'\,\}}{\{\,\Phi \wedge \Phi'\,\}\,S \parallel S'\,\{\,\Psi \wedge \Psi'\,\}}$$

This proof rule would allow the verification of parallel programs in a compositional way by considering the parts S and S' separately, i.e., to prove the correctness of $S \parallel S'$ it suffices to prove $\{\,\Phi\,\}\,S\,\{\,\Psi\,\}$ and $\{\,\Phi'\,\}\,S'\,\{\,\Psi'\,\}$. Due to interaction between S and S', albeit in the form of access to shared variables or by exchanging messages, this rule is unfortunately *not valid* in general. Much effort has been devoted to obtain proof rules of the above form. The development of a compositional proof system for parallel systems that interact (in some way) has turned out to be hard, and proof rules tend to become rather complex. There are several reasons for this.

Parallelism inherently leads to the introduction of *non-determinism*. This results in the fact that for parallel programs which interact using shared variables the input-output behavior strongly depends on the order in which the shared variables are accessed. For instance, for statements $S \equiv x := x{+}2$, and $S' \equiv x := x{+}1\,;\ x := x{+}1$, and $S'' \equiv x := 0$, the value of x at termination of $S \parallel S''$ can be either 0 or 2, whereas the value of x at termination of $S' \parallel S''$ can be 0, 1 or 2. The different outcomes for x depend on the order of execution of the statements in S and S'', or S' and S''. Hence, although the input-output behavior of S and S' is obviously the same – increasing x by 2 – there is no guarantee that this remains true in an identical parallel context (like S'').

Concurrent processes can potentially interact at any point of their execution, not only at the beginning or end of their computation. In order to infer how parallel programs interact, it is not sufficient to know properties of their initial and final states only. Instead, it is important to be able to make, in addition, statements about what happens *during* the computation. So, relevant properties should not only refer to start and end-states, but also to the evolution between states during executions.

The main problem of applying the classical approach to the verification of parallel and reactive systems is its sole focus on the idea that a program (system) computes a function from inputs to outputs. That is, given certain allowed input(s), certain desired output(s) are produced. For parallel systems the computation usually does not terminate, and correctness refers to the behavior of the system in time, not only to the final result of a computation (if a computation ends at all). Typically, the global property of a parallel program can often not be stated in terms of an input-output relationship only.

The main difficulties of applying classical assertional verification to parallel (and reactive) systems are:

- the proof rules adequately reflect the input-output behavior of programs, but do not refer to their (finite or infinite) executions.

- finding appropriate invariants is typically hard.
- the proof rules cannot handle fairness.
- the number of proof obligations becomes very large [1].

As a result of the last drawback, proofs are rather lengthy, tedious, and thus quite vulnerable to (human) errors. Besides, the organization of proofs of such complexity in a comprehensible form is difficult. To overcome these drawbacks, proof assistants and theorem provers may be used (see also Chapter 1). The first drawback stems from the fact that propositional (or predicate) logic is static, i.e., each formula represents the set of states that satisfy it. In order to capture the continuous behavior of reactive systems, the dynamic evolution between states during the execution needs to be addressed.

3.1.4 Temporal Logic

For reactive systems, correctness depends on the executions of the system – not only on the input-output relationship of a computation – and on fairness issues. Temporal logic is a formalism par excellence for treating these two aspects. It allows to express properties about the relation between states during an execution. This has first been recognized by Pnueli [151] in the late seventies. It is a well-accepted and commonly used specification technique for expressing properties of executions (of reactive systems) at a rather high level of abstraction.

Temporal logic is a member of the broader class of modal logics. Besides propositional (or predicate) logic these logics possess *modal* operators. A typical modality in temporal logic is "sometime Φ" which holds if formula Φ holds at some future moment. There do exist various temporal logics with different modal operators and with different interpretations thereof. The main distinguishing aspects are:

- *Propositional or first-order logic* as basis. In propositional temporal logic, atomic propositions express simple atomic facts about the state of the system. In first-order temporal logic, atomic propositions are allowed to contain expressions constructed from function and predicate symbols, and quantification is allowed.
- *The underlying nature of time.* This aspect refers to issues like:
  - are temporal operators evaluated as true or false of points of time, or are they evaluated over intervals of time?

[1] Due to possible interactions between parallel programs, $N \cdot M$ additional proof obligations have to be checked for parallel programs of length N and M using the classical approach of [146].

  - is time discrete where the present moment refers to the current state and the next moment corresponds to the immediate successor state, or is it continuous?
  - is time linear – at each moment in time there is a single successor moment – or does it have a branching, tree-like structure, where time may split into alternative courses?

- *Future or past modalities.* Future modal operators refer to events that happen in the future; past modalities refer to moments in the past.

In this book, we only consider propositional temporal logics for which any temporal operator is evaluated as true or false of points of time. This chapter considers temporal logic that is based on a linear interpretation of discrete time – linear temporal logic (PLTL). Chapters 6 and 9 both deal with branching temporal logics; Chapter 6 with a discrete time variant (CTL), and Chapter 9 with a continuous variant (TCTL). Table 3.2 summarizes the distinguishing features of these logics.

logic	linear	branching	discrete	continuous
PLTL	√		√	
CTL		√	√	
TCTL		√		√

Table 3.2: Classification of the temporal logics in this book

3.2 Syntax of Propositional Linear Temporal Logic

The set of formulas that can be stated in propositional linear temporal logic (PLTL, for short) is defined as follows.

Definition 3.4. (**Syntax of propositional linear temporal logic**)
Let p be an atomic proposition. Formulas in PLTL satisfy the following rules:

1. p is a formula.
2. If Φ is a formula, then $\neg\,\Phi$ is a formula.
3. If Φ and Ψ are formulas, then $\Phi \vee \Psi$ is a formula.
4. If Φ is a formula, then $\mathsf{X}\,\Phi$ is a formula.
5. If Φ and Ψ are formulas, then $\Phi\,\mathsf{U}\,\Psi$ is a formula.
6. Anything else is not a formula.

PLTL thus extends propositional logic with the temporal operators X (pronounced "neXt") and U (pronounced "Until"). The intuitive interpretation of these operators is as follows. Formula $\mathsf{X}\,\Phi$ holds at the current moment, if Φ holds at the next moment. Formula $\Phi\,\mathsf{U}\,\Psi$ holds at the current moment, if there is some future moment for which Ψ holds and Φ holds at all moments until that future moment.

The precedence order on the operators is as follows. The unary operators bind stronger than the binary ones. $\neg$ and X bind equally strong. The temporal operator U takes precedence over $\wedge$, $\vee$, and $\Rightarrow$. Parentheses are omitted whenever appropriate, e.g., we write $\neg\,\Phi\,\mathsf{U}\,\mathsf{X}\,\Psi$ instead of $(\neg\,\Phi)\,\mathsf{U}\,(\mathsf{X}\,\Psi)$. Operator U is right-associative, e.g., $p\,\mathsf{U}\,q\,\mathsf{U}\,r$ stands for $p\,\mathsf{U}\,(q\,\mathsf{U}\,r)$.

Example 3.2. Let $AP = \{\, x = 1, x < 2, x \geqslant 3 \,\}$ be the set of atomic propositions. Example PLTL-formulas over AP are: $\mathsf{X}\,(x = 1)$, $\neg\,(x < 2)$, $x < 2 \vee x = 1$, $(x < 2)\,\mathsf{U}\,(x \geqslant 3)$, and $\mathsf{X}\,(x = 1)$. The second and the third formula are also propositional logic formulas. An example of a PLTL-formula in which temporal operators are nested is $\mathsf{X}\,((x < 2)\,\mathsf{U}\,\mathsf{X}\,(x \geqslant 3))$. (End of example.)

3.3 Semantics of PLTL

3.3.1 Kripke Structures

The above definition provides us a recipe for the construction of PLTL-formulas, but it does not give an interpretation to these operators. The formal meaning of temporal logic formulas is defined using the notion of Kripke structures [2].

Definition 3.5. (**Kripke structure**)
A *Kripke structure* $\mathcal{K}$ is a tuple $(S, I, R, Label)$ where

- S is a countable set of states,
- $I \subseteq S$ is a set of initial states,
- $R \subseteq S \times S$ is a transition relation satisfying $\forall s \in S.\,(\exists s' \in S.\,(s, s') \in R)$
- $Label : S \longrightarrow 2^{AP}$ is an interpretation function on S.

Transition relation R associates to any state its set of successors $R(s) = \{\, s' \mid (s, s') \in R \,\}$ which is required to be non-empty. Each state thus has at least

[2] Kripke used similar structures to provide an interpretation to modal logics [112].

one successor, i.e., there are no states without any successor. In other words, R is a total relation. Note that a state may have no predecessors, i.e., $R^{-1}(s) = \{ s' \mid (s', s) \in R \}$ may be empty, and that an initial state is not required to have no predecessors.

Example 3.3. *The following Kripke structure models a* triple modular redun-

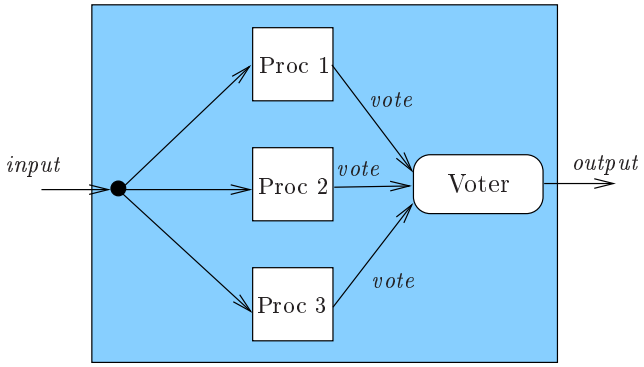


Figure 3.1: A triple modular redundant system

dant system *(TMR, cf. Figure 3.1), a fault-tolerant computer system consisting of three processors and a single (majority) voter. The processors generate results and the voter decides upon the correct value by taking a majority vote. Components may fail and after failure may be repaired. Initially all components are functioning correctly. For simplicity, it is assumed that one component can be repaired at a time. If the voter fails, the entire system is assumed to have failed, and after a repair the system is assumed to start "as good as new". Consider as atomic propositions the set* $AP = \{ \mathrm{up}_i \mid 0 \leqslant i < 4 \} \cup \{ \mathrm{down} \}$. *The components of the Kripke structure are:*

- $S = \{ s_{i,1} \mid 0 \leqslant i < 4 \} \cup \{ s_{0,0} \}$.
- $I = \{ s_{3,1} \}$
- $R = \{ (s_{i,1}, s_{0,0}) \mid 0 \leqslant i < 4 \} \cup \{ (s_{0,0}, s_{3,1}) \}$
  $\cup \{ (s_{i,1}, s_{i,1}) \mid 0 \leqslant i < 4 \} \cup \{ (s_{i,1}, s_{i+1,1}) \mid 0 \leqslant i < 3 \}$
  $\cup \{ (s_{i+1,1}, s_{i,1}) \mid 0 \leqslant i < 3 \}$
- $\mathrm{Label}(s_{0,0}) = \{ \mathrm{down} \}$ *and* $\mathrm{Label}(s_{i,1}) = \{ \mathrm{up}_i \}$ *for* $0 \leqslant i < 4$.

State $s_{i,j}$ *models that* i $(0 \leqslant i < 4)$ *processors and* j $(0 \leqslant j < 2)$ *voters are operational. The Kripke structure is depicted in Figure 3.2. Here states are depicted by circles, initial states have an incoming arrow with no source state, and the relation* R *is denoted by arrows, i.e., there is an arrow from* s *to* s' *if and only if* $(s, s') \in R$. *The labelling* $\mathrm{Label}(s)$ *is indicated beside the state* s, *where for simplicity, the set brackets are omitted from singleton sets, e.g.,* down *denotes* $\{ \mathrm{down} \}$. *(End of example.)*

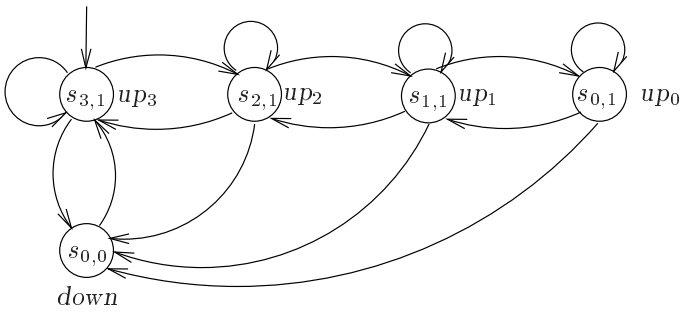


Figure 3.2: A Kripke structure of the TMR system

As each state in a Kripke structure is required to have at least one successor state, one may wonder how to model a system that contains a deadlock state, i.e., a state from which no progress can be made. This situation can be modeled, by adding to the Kripke structure a distinguishing state that is equipped with a self-loop. Each state from which no progress should take place is equipped with a transition to this new state.

Definition 3.6. (**Path**)
A *path* in $\mathcal{K}$ is an infinite sequence of states $s_0\, s_1\, s_2 \ldots$ such that $(s_i, s_{i+1}) \in R$ for all $i \geqslant 0$.

A path is thus an infinite sequence of states such that between successive states transitions do exist. For path $\sigma = s_0\, s_1\, s_2 \ldots$ and integer $i \geqslant 0$ we use $\sigma[i]$ to denote the $(i{+}1)$-th state of σ, i.e., $\sigma[i] = s_i$ and use σ^i to denote the suffix of σ obtained by removing its first i states, i.e., $\sigma^i = s_i\, s_{i+1}\, s_{i+2} \ldots$. Note that $\sigma^i[j] = \sigma[i{+}j]$; in particular, $\sigma^0 = \sigma$. For paths with a certain regularity, such as $\sigma = s_0\, s_0\, s_0 \ldots$ or $\widehat{\sigma} = s_0\, s_1\, s_0\, s_1 \ldots$ an alternative and more succinct notation is $\sigma = (s_0)^\omega$ and $\widehat{\sigma} = (s_0\, s_1)^\omega$. The set of paths that start in state s is denoted $Paths(s)$, i.e., $Paths(s) = \{\, \sigma \in S^\omega \mid \sigma[0] = s \,\}$.

Example 3.4. An example path in the Kripke structure modeling the TMR system is $\sigma = s_{3,1}\, s_{2,1}\, s_{0,0}\, s_{3,1}\, s_{2,1}\, s_{0,0} \ldots$. *That is,* $\sigma = (s_{3,1}\, s_{2,1}\, s_{0,0})^\omega$. *We have, for example,* $\sigma[2] = s_{0,0}$, $\sigma[3] = s_{3,1}$, $\sigma^2 = s_{0,0}\, s_{3,1}\, s_{2,1}\, s_{0,0} \ldots$, *i.e.,* $\sigma^2 = (s_{0,0}\, s_{3,1}\, s_{2,1})^\omega$.

Some alternative paths are $\widehat{\sigma} = s_{3,1}\, s_{2,1}\, s_{1,1}\, s_{0,1}\, s_{1,1}\, s_{0,1} \ldots$ *and* $\tilde{\sigma} = s_{3,1}\, s_{3,1} \ldots$, *i.e.,* $\widehat{\sigma} = s_{3,1}\, s_{2,1}\, (s_{1,1}\, s_{0,1})^\omega$ *and* $\tilde{\sigma} = (s_{3,1})^\omega$. *(End of example.)*

3.3.2 Semantics of PLTL

The meaning of formulas in logic is defined by means of a satisfaction relation (denoted by $\models$) between a path σ, and PLTL-formula Φ. The concept is that

$\sigma \models \Phi$ if and only if Φ is valid for σ.

Definition 3.7. (**Semantics of PLTL**)
Let $p \in AP$ be an atomic proposition, σ a path and Φ, Ψ PLTL-formulas. The satisfaction relation $\models$ is defined by:

$$
\begin{array}{ll}
\sigma \models p & \text{iff } p \in \mathit{Label}(\sigma[0]) \\
\sigma \models \neg\, \Phi & \text{iff not } (\sigma \models \Phi) \\
\sigma \models \Phi \ \vee\ \Psi & \text{iff } (\sigma \models \Phi) \text{ or } (\sigma \models \Psi) \\
\sigma \models \mathsf{X}\, \Phi & \text{iff } \sigma^1 \models \Phi \\
\sigma \models \Phi\, \mathsf{U}\, \Psi & \text{iff } \exists\, j \geqslant 0.\ \big(\sigma^j \models \Psi \text{ and } (\forall\, 0 \leqslant k < j.\, \sigma^k \models \Phi)\big)
\end{array}
$$

If $\sigma \models \Phi$ we say that path σ satisfies Φ. The first three clauses of the above definition coincide with the semantics of propositional logic, cf. Definition 3.3. The fourth clause states that $\sigma = s_0\, s_1\, s_2 \ldots$ satisfies $\mathsf{X}\, \Phi$ if and only if its suffix $s_1\, s_2 \ldots$ satisfies Φ. Note that the propositions in s_0 play no role here. Formula $\Phi\, \mathsf{U}\, \Psi$ is satisfied by σ if there is some suffix of σ, σ^j say, that satisfies Ψ, and all preceding suffixes satisfy Φ. Since the case $j = 0$ is allowed, it means that suffix σ^j may be equal to σ itself. In that case, the second conjunct of the fifth clause becomes vacuously true as there is no k such that $0 \leqslant k < j = 0$. We will discuss this issue more extensively in Section 3.5. Intuitively, $\sigma \models \Phi\, \mathsf{U}\, \Psi$ if and only if a Ψ-state will be reached at some moment in the future, and in all preceding states Φ is guaranteed to hold. The semantics of U is neither stating anything about the validity of Φ for suffix σ^j, nor does it state anything about the validity of Ψ for σ^k $(k < j)$, nor about the validity of Φ (and Ψ) in any suffix σ^m with $m > j$. The latter means that once a Ψ-state is reached, the validity of Φ (and Ψ) in any subsequent state is irrelevant.

Example 3.5. Consider the paths σ, $\widehat{\sigma}$ and $\tilde{\sigma}$ from the previous example. It follows, for instance, that $\sigma \models \mathsf{X}\, up_2$ and $\sigma \models \mathsf{X}\mathsf{X}\mathsf{X}\, \neg\, down$. In addition, we have $\sigma \models (up_2 \ \vee\ up_3)\, \mathsf{U}\, down$ and $\sigma \models up_0\, \mathsf{U}\, up_3$. The latter follows from the fact that for $j = 0$ suffix σ^0 satisfies proposition up_3; the fact that there is no state in σ for which up_0 holds is thus irrelevant. As a final example, we have that $\tilde{\sigma} \models \neg\, (true\, \mathsf{U}\, down)$, as a state for which proposition down holds is never reached along this path. *(End of example.)*

3.3.3 Auxiliary Temporal Operators

To ease the specification of relevant properties, four auxiliary temporal operators are now introduced. These operators are defined in terms of the operators introduced before, and thus, do not add any expressiveness to the language of PLTL. The temporal operators G (pronounced "always" or "Globally") and F

(pronounced "eventually" or "Future") are defined by:

$$\begin{aligned} \mathsf{F}\,\Phi &\equiv \mathrm{true}\,\mathsf{U}\,\Phi \\ \mathsf{G}\,\Phi &\equiv \neg\,\mathsf{F}\,\neg\,\Phi \end{aligned}$$

Since true is a tautology, i.e., it is valid in all states, $\mathsf{F}\,\Phi$ indeed denotes that Φ holds at some moment in the future. Suppose there is no moment in the future for which $\neg\,\Phi$ holds. Then, the counterpart of Φ holds at any moment. This explains the definition of $\mathsf{G}\,\Phi$. Alternative notations are: F may be denoted as $\Diamond$, G as $\Box$, and X as $\bigcirc$. In this book, we use the traditional notations F, G and X. The precedence order on the operators is extended in the following way: the operators $\neg$ and X bind equally strong and stronger than F and G that also bind equally strong.

To summarize, a formula without a temporal operator ($\mathsf{X}, \mathsf{F}, \mathsf{G}, \mathsf{U}$) at the "top level" refers to the first state of a path, the formula $\mathsf{X}\,\Phi$ to the next state, $\mathsf{G}\,\Phi$ to all (future) states, $\mathsf{F}\,\Phi$ to some future state, and U to all future states until a certain condition becomes valid.

The formal interpretation of G and F can be obtained from the above characterization and Definition 3.7 by straightforward calculation. We derive for the semantics of $\mathsf{F}\,\Phi$:

$$\begin{aligned} & \sigma \models \mathsf{F}\,\Phi \\ \Leftrightarrow\quad & \{\ \text{definition of } \mathsf{F}\ \} \\ & \sigma \models \mathrm{true}\,\mathsf{U}\,\Phi \\ \Leftrightarrow\quad & \{\ \text{semantics of } \mathsf{U}\ \} \\ & \exists\, j \geqslant 0.\ \big(\sigma^j \models \Phi\ \wedge\ (\forall\, 0 \leqslant k < j.\, \sigma^k \models \mathrm{true})\big) \\ \Leftrightarrow\quad & \{\ \text{calculus}\ \} \\ & \exists\, j \geqslant 0.\, \sigma^j \models \Phi \end{aligned}$$

Therefore, $\mathsf{F}\,\Phi$ is valid for path σ if and only if there is some (not necessarily direct) successor state of $\sigma[0]$, or $\sigma[0]$ itself where Φ is valid.

Using this result for F for the semantics of $\mathsf{G}\,\Phi$ we derive:

$$\begin{aligned} & \sigma \models \mathsf{G}\,\Phi \\ \Leftrightarrow\quad & \{\ \text{definition of } \mathsf{G}\ \} \\ & \sigma \models \neg\,\mathsf{F}\,\neg\,\Phi \\ \Leftrightarrow\quad & \{\ \text{using result of previous derivation}\ \} \\ & \neg\,\big(\exists\, j \geqslant 0.\, \sigma^j \models \neg\,\Phi\big) \\ \Leftrightarrow\quad & \{\ \text{semantics of } \neg\,\Phi\ \} \\ & \neg\,\big(\exists\, j \geqslant 0.\, \neg\,(\sigma^j \models \Phi)\big) \\ \Leftrightarrow\quad & \{\ \text{predicate calculus}\ \} \end{aligned}$$

$$\forall\, j \geqslant 0.\, \sigma^j \models \Phi.$$

Therefore, path σ satisfies the formula $\mathsf{G}\,\Phi$ if and only if all its suffixes (including σ itself) satisfy Φ. This explains the expression "always" Φ. Combinations of F and G frequently occur and have the following intuitive meaning: $\mathsf{F}\,\mathsf{G}\,\Phi$ is satisfied if from some future moment on Φ holds continuously, and $\mathsf{G}\,\mathsf{F}\,\Phi$ is satisfied if Φ holds "infinitely often".

Example 3.6. Consider again the paths of the TMR system. The simple path $(s_{3,1})^\omega$ *satisfies, amongst others, the formulas* $\mathsf{G}\,up_3$ *(as each state of the path is labelled with proposition* up_3*),* $\mathsf{G}\,\neg\, down$ *(as a down-state is never reached),* $\mathsf{F}\,\mathsf{G}\,up_3$ *(as the path itself satisfies* $\mathsf{G}\,up_3$*), and* $\mathsf{G}\,\mathsf{F}\,up_3$ *(as a state labelled with* up_3 *is visited infinitely often.)*

Path $(s_{3,1}\, s_{2,1}\, s_{0,0})^\omega$ *satisfies the formula* $\mathsf{F}\,down$ *as state* $s_{0,0}$ *is reachable. Besides it satisfies* $\mathsf{G}\,((up_3 \vee up_2)\,\mathsf{U}\,down)$*,* $\mathsf{G}\,\mathsf{F}\,down$ *and* $\mathsf{G}\,\mathsf{F}\,((up_3 \vee up_2)\,\mathsf{U}\,down)$*.*

Finally, path $s_{3,1}\, s_{2,1}(s_{1,1}\, s_{0,1})^\omega$ *satisfies the following formulas:* $\mathsf{F}\,up_2$*,* $\neg\,\mathsf{G}\,\mathsf{F}\,up_2$*, and* $\mathsf{G}\,(up_3 \;\Rightarrow\; \mathsf{X}\,\mathsf{G}\,\neg\, up_3)$*.* *(End of example.)*

The temporal operator U is sometimes called a *strong* until, as the formula $\Phi\,\mathsf{U}\,\Psi$ implicitly states that there exists some future state for which Ψ holds. That is, if a path satisfies $\Phi\,\mathsf{U}\,\Psi$, then it also satisfies $\mathsf{F}\,\Psi$. This is often too strong, as it is not always clear that indeed a Ψ-state will be reached. A weaker variant of until, the *unless* operator W, states that Φ holds continuously either until Ψ holds for the first time, or throughout the path. This operator is defined by:

$$\Phi\,\mathsf{W}\,\Psi \;\equiv\; \mathsf{G}\,\Phi \;\vee\; (\Phi\,\mathsf{U}\,\Psi)$$

or equivalently by:

$$\Phi\,\mathsf{W}\,\Psi \;\equiv\; \mathsf{F}\,\neg\,\Phi \;\Rightarrow\; \Phi\,\mathsf{U}\,\Psi$$

The final auxiliary operator that we treat here is the *release* operator, denoted R; it is defined as follows:

$$\Phi\,\mathsf{R}\,\Psi \;\equiv\; \neg\,(\neg\,\Phi\,\mathsf{U}\,\neg\,\Psi)$$

Its intuitive interpretation is as follows. Formula $\Phi\,\mathsf{R}\,\Psi$ holds for path σ if Ψ always holds, a requirement that is released as soon as Φ becomes valid. Thus, the formula $\mathrm{false}\,\mathsf{R}\,\Phi$ is valid for σ if Φ always holds, since the release condition (false) is a contradiction (i.e., not valid in any state). We thus have:

$$\sigma \models \mathrm{false}\,\mathsf{R}\,\Phi \;\text{ if and only if }\; \sigma \models \mathsf{G}\,\Phi$$

In a similar way, formula $\Phi \,\mathsf{R}\, \neg \Psi$ can be shown to be valid for a path if on this path it holds that if Ψ ever becomes true eventually, then it is strictly preceded by an occurrence of Φ.

3.3.4 Model Checking, Satisfiability and Validity

In Chapter 1, we gave an informal definition of the model-checking problem. Given the formal machinery developed so far we are now in a position to give a more precise characterization. We first define what it means for a Kripke structure to satisfy a PLTL-formula. [3]

Definition 3.8. (**Satisfaction of a formula by a Kripke structure**)
For Kripke structure $\mathcal{K} = (S, I, R, Label)$ and PLTL-formula Φ:

$$\mathcal{K} \models \Phi \text{ if and only if } \forall s \in I.\ (\forall \sigma \in Paths(s).\ \sigma \models \Phi)$$

Stated in words, Kripke structure $\mathcal{K}$ satisfies Φ if and only if all paths that start in some of its initial states satisfy Φ.

Example 3.7. The Kripke structure of the TMR system does not satisfy the formula $\mathsf{G} \neg down$, *as there exists a path starting from* $s_{3,1}$, *the only initial state, that reaches a* down*-state. For instance, path* $s_{3,1}\, s_{0,0} \ldots$ *is such path. Similarly, it does not satisfy* $\mathsf{G}\,\mathsf{F}\, down$, *since e.g., path* $(s_{3,1})^{\omega}$ *does never reach a* down*-state. The Kripke structure does, however, satisfy* $\mathsf{G}\,(down \Rightarrow \mathsf{X}\, up_3)$, *as being currently in the* down*-state, it means that immediately afterwards the system is repaired, i.e., in the* up_3*-state. This holds for all paths. (End of example.)*

The model-checking problem is now formalized as follows:

> *The model-checking problem is: given a Kripke structure* $\mathcal{K}$, *and a formula* Φ, *do we have* $\mathcal{K} \models \Phi$*?*

If $\mathcal{K} \models \Phi$ we say that $\mathcal{K}$ is a *model* of the formula Φ; this explains the term "model checking". The model-checking problem should not be confused with the more traditional satisfiability problem in logic.

The *satisfiability* problem is: given a formula Φ, does there exist a Kripke structure $\mathcal{K}$ such that $\mathcal{K} \models \Phi$? While for model checking the Kripke structure

[3] This relation is denoted by $\models$, the same symbol as we have used to defined the semantics of propositional logic and PLTL. This overloading of notation is, however, not a large problem as it is always clear whether $\models$ is used for a model (as below) or for a path (as in Definition 3.7).

$\mathcal{K}$ is given, this is not the case for the satisfiability problem. For PLTL, the satisfiability problem is decidable, for first-order temporal logic – where atomic propositions are richer – this problem is undecidable. A logic is said to be *decidable* if its satisfiability problem is decidable. For propositional logic, the satisfiability problem is NP-complete, i.e., the best known algorithm to check for satisfiability needs a time exponential in the length of the formula. This can be seen by considering the following naive procedure. Consider a propositional formula Φ of length n, i.e., Φ contains n propositions. In order to check for satisfiability of Φ each proposition is checked for all possible values, `tt` of `ff`. This requires 2^n combinations to be checked for n propositions.

The *validity* problem is: given a property Φ, do we have for all Kripke structures $\mathcal{K}$: $\mathcal{K} \models \Phi$? Validity thus amounts to check whether a formula is a tautology. The difference to the satisfiability problem is that to solve the validity problem one has to check whether $\mathcal{K} \models \Phi$ for all existing $\mathcal{K}$, rather than to determine the existence of one (or more) such $\mathcal{K}$. Logically speaking, Φ is valid if $\neg\,\Phi$ is unsatisfiable. In order to show that a formula is not a tautology it thus suffices to construct one Kripke structure that refutes it.

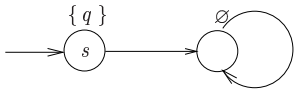
At first sight the validity problem seems to be undecidable – there does not seem to be an effective method for deciding whether formula Φ is valid in all Kripke structures, since these structures may have an infinite number of states. However, due to the so-called *finite-model property* for PLTL, it suffices to consider only all structures with a finite number of states. The finite-model property states that if a PLTL-formula is satisfiable, then it is satisfiable in a structure with a finite set of states.

Example 3.8. $\mathsf{G}\,p \;\Rightarrow\; \neg\,(\neg\,p \;\wedge\; \mathsf{X}\,\mathsf{G}\,\neg\,p)$ *is a tautology. The proof goes along the following lines:*

$$
\begin{array}{ll}
 & \sigma \models \neg\,\mathsf{G}\,p \;\vee\; (\neg\,(\neg\,p \;\wedge\; \mathsf{X}\,\mathsf{G}\,\neg\,p)) \\
\Leftrightarrow & \quad \{\ \textit{calculus; semantics}\ \neg\ \} \\
 & \quad \neg\,(\sigma \models \mathsf{G}\,p \;\wedge\; (\neg\,p \;\wedge\; \mathsf{X}\,\mathsf{G}\,\neg\,p)) \\
\Leftrightarrow & \quad \{\ \textit{semantics conjunction}\ \} \\
 & \quad \neg\,(\sigma \models \mathsf{G}\,p \;\wedge\; \sigma \models \neg\,p \;\wedge\; \mathsf{X}\,\mathsf{G}\,\neg\,p) \\
\Leftrightarrow & \quad \{\ \textit{semantics}\ \mathsf{G}\ \textit{(twice), conjunction and semantics}\ \mathsf{X}\ \} \\
 & \quad \neg\,((\forall\, j \geqslant 0.\, \sigma^j \models p) \;\wedge\; \sigma \models \neg\,p \;\wedge\; (\forall\, j \geqslant 0.\, (\sigma^1)^j \models \neg\,p)) \\
\Leftrightarrow & \quad \{\ \textit{calculus}\ \} \\
 & \quad \neg\,((\forall\, j \geqslant 0.\, \sigma^j \models p) \;\wedge\; (\forall\, j \geqslant 0.\, \sigma^j \models \neg\,p))
\end{array}
$$

Note that all steps in the derivations are equivalences, and hence the reverse implication is also valid. *(End of example.)*

Example 3.9. $\mathsf{G}\,p\,\mathsf{U}\,\mathsf{F}\,q \;\Rightarrow\; \mathsf{G}\,(p\,\mathsf{U}\,\mathsf{F}\,q)$ *is satisfiable – this can be shown by just constructing a model that satisfies* $\mathsf{G}\,(p\,\mathsf{U}\,\mathsf{F}\,q)$ *– but is not a tautology as there exists a Kripke structure such as:*



The only path starting in s satisfies formula $\mathsf{G}\,p\,\mathsf{U}\,\mathsf{F}\,q$ as the initial state of this path satisfies q and thus the path satisfies $\mathsf{F}\,q$. Thus, $\mathcal{K} \models \mathsf{G}\,p\,\mathsf{U}\,\mathsf{F}\,q$. However, $\mathsf{G}\,(p\,\mathsf{U}\,\mathsf{F}\,q)$ is not fulfilled as the successor of s does not satisfy $p\,\mathsf{U}\,\mathsf{F}\,q$. Thus, $\mathcal{K} \not\models \mathsf{G}\,(p\,\mathsf{U}\,\mathsf{F}\,q)$. *(End of example.)*

Satisfiability is relevant to assertional system verification using PLTL in the following sense. Consider a system specification and its implementation, both formalized as PLTL-formulas, Φ_{spec} and Φ_{impl}, say. Checking whether the implementation conforms to the system specification may be interpreted as: $\Phi_{impl} \Rightarrow \Phi_{spec}$. If a PLTL-formula is viewed as a characterization of the set of paths that satisfy it, then this implication should be read as: "all the paths that are allowed by the implementation are also allowed by the specification". If $\Phi_{impl} \Rightarrow \Phi_{spec}$ is not satisfiable, i.e., no Kripke structure exists for which this formula holds, then the relation between the implementation and specification cannot be realized in any structure. Thus, the implementation can never be a correct implementation of the specification.

3.4 Axiomatization

The validity of a PLTL-formula of the form $\Phi \Leftrightarrow \Psi$ can be derived using the semantics of Definition 3.7 by proving for all paths σ:

$$\sigma \models \Phi \text{ if and only if } \sigma \models \Psi$$

This is usually a rather cumbersome task, since we have to reason about the formal semantics that is defined in terms of the Kripke structure $\mathcal{K}$. Let us, for instance, try to deduce that:

$$\Phi\,\mathsf{U}\,\Psi \;\Leftrightarrow\; \Psi \;\vee\; (\Phi \;\wedge\; \mathsf{X}\,(\Phi\,\mathsf{U}\,\Psi))$$

Intuitively the implication $\Leftarrow$ is valid: if in the first state of a path Ψ holds, then obviously $\Phi\,\mathsf{U}\,\Psi$ holds (for arbitrary Φ), since Ψ can be reached via a path of length 0. Otherwise, if Φ holds in the first state and in the next state $\Phi\,\mathsf{U}\,\Psi$ holds, then $\Phi\,\mathsf{U}\,\Psi$ holds. For the implication in the other direction, a similar informal reasoning applies. The reader is encouraged to first prove the above fact without considering the derivation below. We formally derive:

$$\begin{array}{ll} & \sigma \models \Psi \;\vee\; (\Phi \;\wedge\; \mathsf{X}\,(\Phi\,\mathsf{U}\,\Psi)) \\ \Leftrightarrow & \{ \text{ semantics of } \wedge \text{ and } \vee \} \\ & (\sigma \models \Psi) \;\vee\; (\sigma \models \Phi \;\wedge\; \sigma \models \mathsf{X}\,(\Phi\,\mathsf{U}\,\Psi)) \end{array}$$

$$
\begin{array}{ll}
\Leftrightarrow & \{ \text{ semantics of } \mathsf{X} \ \} \\
& (\sigma \models \Psi) \ \vee \ (\sigma \models \Phi \ \wedge \ \sigma^1 \models \Phi \,\mathsf{U}\, \Psi) \\
\Leftrightarrow & \{ \text{ semantics of } \mathsf{U} \ \} \\
& (\sigma \models \Psi) \ \vee \ (\sigma \models \Phi \ \wedge \ (\exists\, j \geqslant 0.\, (\sigma^1)^j \models \Psi \ \wedge \ \forall 0 \leqslant k < j.\, (\sigma^1)^k \models \Phi)) \\
\Leftrightarrow & \{ \text{ calculus using } (\sigma^1)^j = \sigma^{j+1} \ \} \\
& (\sigma \models \Psi) \ \vee \ (\exists\, j \geqslant 0.\, \sigma^{j+1} \models \Psi \ \wedge \ \forall 0 \leqslant k < j.\, (\sigma^{k+1} \models \Phi \ \wedge \ \sigma \models \Phi)) \\
\Leftrightarrow & \{ \text{ calculus using } \sigma^0 = \sigma \ \} \\
& (\sigma \models \Psi) \ \vee \ (\exists\, j \geqslant 0.\, \sigma^{j+1} \models \Psi \ \wedge \ \forall 0 \leqslant k < j{+}1.\, \sigma^k \models \Phi) \\
\Leftrightarrow & \{ \text{ calculus using } \sigma^0 = \sigma \ \} \\
& \quad (\exists\, j = 0.\, \sigma^0 \models \Psi \ \wedge \ \forall 0 \leqslant k < j.\, \sigma^k \models \Phi) \\
& \vee \ (\exists\, j \geqslant 0.\, \sigma^{j+1} \models \Psi \ \wedge \ \forall 0 \leqslant k < j{+}1.\, \sigma^k \models \Phi) \\
\Leftrightarrow & \{ \text{ predicate calculus } \} \\
& (\exists\, j \geqslant 0.\, \sigma^j \models \Psi \ \wedge \ \forall 0 \leqslant k < j.\, \sigma^k \models \Phi) \\
\Leftrightarrow & \{ \text{ semantics of } \mathsf{U} \ \} \\
& \sigma \models \Phi \,\mathsf{U}\, \Psi.
\end{array}
$$

As we can learn from this calculation, formula manipulation is tedious and error-prone. A more effective way to check the validity of formulas is to use the syntax of the formulas rather than their semantics. The concept is to define a set of axioms that allow the replacement of PLTL-formulas by semantically equivalent PLTL-formulas for reasoning at a syntactic level. The set of such axioms obtained is called a *sound axiomatization.*

It is not our aim to cover all possible axioms for PLTL but rather to give the reader some elementary ones that are convenient. The axioms presented in Table 3.3 are grouped and each group has been given a name, for reference purposes. Using the idempotency and the absorption axioms any non-empty sequence of F and G can be reduced to either F, G, $\mathsf{F}\,\mathsf{G}$, or $\mathsf{G}\,\mathsf{F}$. The validity of these axioms can be established using the semantic interpretation as we have exemplified for the first expansion axiom. The difference is that we only have to perform these tedious proofs once; thereafter these axioms can be universally applied, i.e., using the principle of substitivity, each occurrence of $\Psi \ \vee \ (\Phi \ \wedge \ \mathsf{X}(\Phi \,\mathsf{U}\, \Psi))$ in a formula can be replaced by $\Phi \,\mathsf{U}\, \Psi$, and vice versa. Note that the validity of the expansion axioms for F and G follows directly from the validity of the expansion axiom for U, using the definition of F and G (the reader may check this).

An axiom is said to be *sound* if it is valid. Formally, the axiom $\Phi \equiv \Psi$ is called sound if and only if, for any path σ:

$$\sigma \models \Phi \text{ if and only if } \sigma \models \Psi$$

Applying sound axioms to a certain formula means that the validity of that formula is unchanged: the axioms do not change the semantics of the formula

Duality axioms:	$\neg \mathsf{G}\,\Phi$	$\equiv$	$\mathsf{F}\,\neg\Phi$
	$\neg \mathsf{F}\,\Phi$	$\equiv$	$\mathsf{G}\,\neg\Phi$
	$\neg \mathsf{X}\,\Phi$	$\equiv$	$\mathsf{X}\,\neg\Phi$
Idempotency axioms:	$\mathsf{G}\,\mathsf{G}\,\Phi$	$\equiv$	$\mathsf{G}\,\Phi$
	$\mathsf{F}\,\mathsf{F}\,\Phi$	$\equiv$	$\mathsf{F}\,\Phi$
	$\Phi\,\mathsf{U}\,(\Phi\,\mathsf{U}\,\Psi)$	$\equiv$	$\Phi\,\mathsf{U}\,\Psi$
	$(\Phi\,\mathsf{U}\,\Psi)\,\mathsf{U}\,\Psi$	$\equiv$	$\Phi\,\mathsf{U}\,\Psi$
Absorption axioms:	$\mathsf{F}\,\mathsf{G}\,\mathsf{F}\,\Phi$	$\equiv$	$\mathsf{G}\,\mathsf{F}\,\Phi$
	$\mathsf{G}\,\mathsf{F}\,\mathsf{G}\,\Phi$	$\equiv$	$\mathsf{F}\,\mathsf{G}\,\Phi$
Commutation axiom:	$\mathsf{X}\,(\Phi\,\mathsf{U}\,\Psi)$	$\equiv$	$(\mathsf{X}\,\Phi)\,\mathsf{U}\,(\mathsf{X}\,\Psi)$
Expansion axioms:	$\Phi\,\mathsf{U}\,\Psi$	$\equiv$	$\Psi \vee (\Phi \wedge \mathsf{X}\,(\Phi\,\mathsf{U}\,\Psi))$
	$\mathsf{F}\,\Phi$	$\equiv$	$\Phi \vee \mathsf{X}\,\mathsf{F}\,\Phi$
	$\mathsf{G}\,\Phi$	$\equiv$	$\Phi \wedge \mathsf{X}\,\mathsf{G}\,\Phi$

Table 3.3: Some sound axioms for PLTL

at hand. If for any semantically equivalent Φ and Ψ it is possible to derive this equivalence using axioms then the axiomatization is said to be *complete*. The list of axioms given before for PLTL is sound, but not complete. A sound and complete axiomatization for PLTL does exist [78], but falls outside the scope of this book. For first-order temporal logic, such complete axiomatization does not exist, as variables (in atomic propositions) may range over the integers, and no complete deductive system exists for reasoning about integers.

3.5 Variants of PLTL

Strict and non-strict interpretation. $\mathsf{G}\,p$ means that p holds in all suffixes of the path under consideration including the path itself. It is called a non-strict interpretation since it also refers to the current state, i.e., the first state of the path. In contrast, a strict interpretation does not refer to the current state. The strict version of G, denoted $\tilde{\mathsf{G}}$, can be defined by $\tilde{\mathsf{G}}\,\Phi \equiv \mathsf{X}\,\mathsf{G}\,\Phi$. That is, $\tilde{\mathsf{G}}\,p$ means that p holds at all successor states without stating anything about the current state. Similarly, we have the strict variants of F and U that are defined by $\tilde{\mathsf{F}}\,\Phi \equiv \mathsf{X}\,\mathsf{F}\,\Phi$ and $\Phi\,\tilde{\mathsf{U}}\,\Psi \equiv \mathsf{X}\,(\Phi\,\mathsf{U}\,\Psi)$. Notice that for X it does not make much sense to distinguish between a strict and a non-strict interpretation. These definitions show that the strict interpretation can be defined in terms of the non-strict interpretation. For the reverse direction we have:

$$\mathsf{G}\,\Phi \equiv \Phi \wedge \tilde{\mathsf{G}}\,\Phi$$
$$\mathsf{F}\,\Phi \equiv \Phi \vee \tilde{\mathsf{F}}\,\Phi$$

$$\Phi \,\mathsf{U}\, \Psi \quad \equiv \quad \Psi \,\vee\, (\Phi \,\wedge\, (\Phi \,\tilde{\mathsf{U}}\, \Psi)).$$

The first two equations are self-explanatory given the above definitions of $\tilde{\mathsf{G}}$ and $\tilde{\mathsf{F}}$. The third equation is justified by the expansion axiom of the previous section: if we substitute $\Phi \,\tilde{\mathsf{U}}\, \Psi \equiv \mathsf{X}(\Phi \,\mathsf{U}\, \Psi)$ in the third equation we indeed obtain the expansion axiom for U from Table 3.3. As a result, the strict and the non-strict interpretation are equivalent: for each non-strict formula there does exist a stric-formula that expresses the same, and vice versa. We will adopt the more common non-strict interpretation.

Past operators. All operators from PLTL refer to the future (including the current state). Consequently, operators are known as future operators. PLTL can, however, also be extended with *past* operators. This can be useful for specifying properties as some properties are more easily (and succinctly) expressed in terms of the past than in terms of the future. For instance, $\mathsf{G}^{-1}\, p$ ("always in the past") means – in the non-strict interpretation – that p is valid now and in any state in the past. $\mathsf{F}^{-1}\, p$ ("sometime in the past") means that either p is valid in the current state or in some state in the past and $\mathsf{X}^{-1}\, p$ means that p holds in the previous state, if such state exists. As for future operators, also for past operators a strict and a non-strict interpretation can be given. The main reason for introducing past operators is to simplify the property specification; the expressive power of PLTL is not increased by the addition of past operators [124] when a discrete notion of time is taken, as we do. Thus, for any property which contains one or more past operators, a PLTL-formula with only future temporal operators does exist expressing the same thing.

3.6 Specifying Properties in PLTL

In order to give the reader some idea of how to formulate informally stated properties in PLTL, we treat a couple of examples. The first example deals with a classical problem, a mutual exclusion algorithm. In the second example we formalize properties of a simple communication system. The third example deals with a (more involved) leader election protocol in a distributed system where processes can start the election at arbitrary, i.e., non-synchronized moments.

3.6.1 Mutual Exclusion

The following program is a mutual exclusion protocol for two processes due to Pnueli (taken from [62]). The only purpose of using this algorithm is its simplicity; the approach we use for formalizing its properties is equally well applicable to other (similar) mutual exclusion algorithms. In Pnueli's protocol there is a single shared variable s which is either 0 or 1, and initially equal to

1. Besides, each process has a local boolean variable y that initially equals 0. The program text for process P_i $(i = 0, 1)$ is as follows:

```
l0: while true do
l1:       Non-critical section
l2:       (y_i, s) := (1, i)
l3:       wait until ((y_{1-i} = 0) ∨ (s ≠ i))
l4:       Critical section
l5:       y_i := 0
l6: od.
```

Here, the statement $(y_i, s) := (1, i)$ is a multiple assignment in which the assignments $y_i := 1$ and $s := i$ are performed in a single, atomic step. Statement **wait until** B for boolean expression B can be seen as an abbreviation of **while** $\neg B$ **do skip od**.

The intuition behind this protocol is as follows. The variables y_0 and y_1 are used by each process to signal the other process of active interest in entering the critical section. On leaving the non-critical section, process P_i sets its own local variable y_i to 1. In a similar way this variable is reset to 0 once the critical section is left. The global variable s is used to resolve a tie situation between the processes. It serves as a log-book in which each process that sets its y variable to 1 signs at the same time. The test at line `l3` says that P_0 may enter its critical section if either y_1 equals 0 – implying that its competitor is not interested in entering its critical section – or if s differs from 0 – implying that the competitor process P_1 performed its assignment to y_1 after P_0 assigned 1 to y_0.

We formalize the following informal requirements on the mutual exclusion protocol. We use proposition $P_i@\mathtt{l}j$ to denote that the execution of process P_i $(i = 0, 1)$ is currently at line `l`j $(0 \leqslant j < 7)$ and propositions $s = b$ and $y_i = b$ to compare the program variables s and y_i with boolean b $(i = 0, 1)$.

- "Mutual exclusion is guaranteed", i.e., the processes cannot occupy their critical sections simultaneously. Using the fact that line `l4` indicates whether a process occupies its critical section, we obtain:

  $$\mathsf{G}\,\neg\,(P_0@\mathtt{l4} \wedge P_1@\mathtt{l4})$$

  Note that it does not suffice to state $\neg\,(P_0@\mathtt{l4} \wedge P_1@\mathtt{l4})$ as this would require mutual exclusion to hold only for the current state of the path, and does not state anything about the remaining part of the path.

- "Absence of unbounded overtaking", i.e., when a process wants to enter its critical section, it eventually will be able to do so. This property has the following form:

  $$\mathsf{G}\,((p_0 \;\Rightarrow\; \mathsf{F}\,q_0) \wedge (p_1 \;\Rightarrow\; \mathsf{F}\,q_1))$$

where proposition p_i stands for "process P_i wants to enter its critical section" and proposition q_i stands for "process P_i is currently in its critical section", i.e., $q_i \equiv P_i@\texttt{14}$. A process wants to enter its critical section is expressed by the fact that its local variable y is equal to 1, thus we obtain:

$$\mathsf{G}\,(((y_0 = 1) \;\Rightarrow\; \mathsf{F}\,P_0@\texttt{14}) \wedge ((y_1 = 1) \;\Rightarrow\; \mathsf{F}\,P_1@\texttt{14}))$$

Note that this is the same as stating

$$\mathsf{G}\,((y_0 = 1) \;\Rightarrow\; \mathsf{F}\,P_0@\texttt{14}) \wedge \mathsf{G}\,((y_1 = 1) \;\Rightarrow\; \mathsf{F}\,P_1@\texttt{14})$$

- "Each process will infinitely often occupy its critical section". This is expressed by:

  $$(\mathsf{G}\,\mathsf{F}\,P_0@\texttt{14}) \wedge (\mathsf{G}\,\mathsf{F}\,P_1@\texttt{14})$$

  It expresses that process P_0 is infinitely often in its critical section, and that the same applies to process P_1. This formula should not be confused with:

  $$\mathsf{G}\,\mathsf{F}\,(P_0@\texttt{14} \wedge P_1@\texttt{14})$$

  since this expresses that infinitely often P_0 and P_1 are occupying their critical section simultaneously, and this is not what we want to express. Similarly,

  $$\mathsf{G}\,\mathsf{F}\,(P_0@\texttt{14} \;\vee\; P_1@\texttt{14})$$

  is not the formula that we are looking for, as this expresses that infinitely often one of the two processes is occupying its critical section. The latter property would be satisfied by for instance, an extremely unfair mutual exclusion algorithm in which just one process always gets access to its critical section while the other process never gets access to its critical section. Such behavior would, however, violate our intended property.

3.6.2 A Communication Channel

Consider an unidirectional channel between two communicating processes: a sender S and a receiver R. Sender S is equipped with an output buffer $S.out$ and recipient R with an input buffer $R.in$. Both buffers have an infinite capacity. If sender S sends a message m to R it inserts the message into its output buffer $S.out$. The output buffer $S.out$ and the input buffer $R.in$ are connected via an unidirectional channel. The receiver R receives messages by deleting messages from its input buffer $R.in$. We also assume that all messages are uniquely identified, and let $AP = \{\, m \in S.out, m \in R.in \,\}$, where m denotes a message, be the set of atomic propositions. We assume that all properties are implicitly

stated for all messages m, i.e., universal quantification over m is assumed. This is for convenience and does not affect model checking if we assume that there are finitely many messages. In addition, we assume that the buffers $S.out$ and $R.in$ behave in a normal way, i.e., they do not disrupt or lose messages, and messages cannot stay in a buffer ad infinitum. A schematic view of the system under consideration is:

We formalize the following informal requirements:

- "A message cannot be in both buffers at the same time".

  $$\mathsf{G}\,\neg\,(m \in S.out \;\wedge\; m \in R.in)$$

- "The channel does not lose messages". This means that messages that are in $S.out$ will eventually be in $R.in$ [4]:

  $$\mathsf{G}\,(m \in S.out \;\Rightarrow\; \mathsf{F}\,(m \in R.in))$$

  If the first property is valid, we can equally well state:

  $$\mathsf{G}\,(m \in S.out \;\Rightarrow\; \mathsf{X}\,\mathsf{F}\,(m \in R.in))$$

  since m cannot be in $S.out$ and $R.in$ at the same time.

- "The channel is order-preserving". This means that if m is offered first by S to its output buffer $S.out$ and subsequently m', then m will be received by R before m':

  $$\begin{aligned}&\mathsf{G}\,(m \in S.out \;\wedge\; \neg\, m' \in S.out \;\wedge\; \mathsf{F}\,(m' \in S.out)\\ &\quad\Rightarrow\; \mathsf{F}\,(m \in R.in \;\wedge\; \neg\, m' \in R.in \;\wedge\; \mathsf{F}\,(m' \in R.in)))\end{aligned}$$

  Note that in the premise the conjunct $\neg\, m' \in S.out$ is needed in order to specify that m' is put in $S.out$ after m. $\mathsf{F}\,(m' \in S.out)$ on its own does not exclude that m' is already in the sender's buffer when message m is in $S.out$. (It is left to the reader to investigate what the meaning is when the first and third occurrence of F are replaced by X.)

[4] If we do not assume uniqueness of messages then this formula does not suffice to characterize the intended property. If, e.g., two copies of m are transmitted and only the last copy is eventually received this would satisfy the given formula. Uniqueness of messages is a prerequisite for the specification of requirements for message-passing systems in temporal logic, see, e.g., [110].

- "The channel does not spontaneously generate messages", i.e., for any m in $R.in$, it must have been previously sent by S. Using the past operator F^{-1} this can be formalized conveniently by:

  $$\mathsf{G}\,\big(m \in R.in \;\Rightarrow\; \mathsf{F}^{-1}\,(m \in S.out)\big)$$

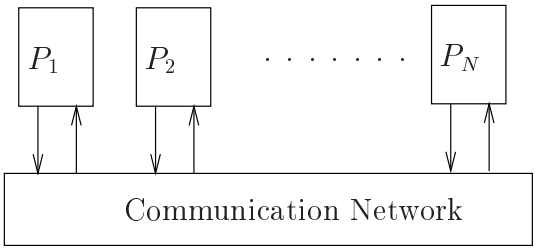
  In the absence of past operators this can be expressed by:

  $$\mathsf{G}\,((\neg\, m \in R.in)\,\mathsf{U}\,(m \in S.out))$$

3.6.3 Dynamic Leader Election

In current distributed systems several services are offered by some dedicated process(es) in the system. Consider, for example, address assignment and registration, query co-ordination in a distributed database system, clock distribution, token regeneration after token loss in a token ring network, initiation of topology updates in a mobile network, load balancing, and so forth. Usually many processes in the system are potentially capable of providing these services. However, for consistency reasons it is usually the case that at any time only one process is allowed to actually provide a given service. This process – called the "leader" – is in fact elected. Sometimes it suffices to elect an arbitrary process, but for other services it is important to elect the process with the best capabilities for performing that service. Here we abstract from specific capabilities and use ranking on basis of process identities. The idea is therefore that the higher the process' identity, the better its capabilities.

Assume we have a finite number $N > 0$ of processes connected via some communication means. The communication between processes is asynchronous, as in the previous example. Pictorially,



Each process has a unique identity, and it is assumed that a total ordering exists on these identities. Processes behave dynamically in the sense that they are initially inactive, i.e., not participating in the election, and may become active, i.e., participating in the election, at arbitrary moments. In order to have some progress we assume that a process cannot be inactive indefinitely; i.e., each process becomes active at some time. Once a process participates it continues to do so, i.e., it does not become inactive anymore. For a given set of active

processes a leader will be elected; if an inactive process becomes active, a new election takes place if this process has a higher identity than the current leader.

We will use i, j as process identities. Let the set of atomic propositions be $\{\, leader_i, active_i, i < j \mid 0 < i, j \leqslant N \,\}$, where $leader_i$ means that process i is a leader, $active_i$ means that process i is active, and $i < j$ is valid if the identity of i is smaller than the identity of j (according to the total order on identities). An inactive process cannot be a leader.

The formulations below (adopted from [34]) use universal and existential quantifications over the set of process identities. Strictly speaking, these quantifications are not part of PLTL. Since we deal with a finite number of processes the universal quantification $\forall i.\, P(i)$, where $P(i)$ is some proposition over process i, can be expanded into $P(1) \wedge \ldots \wedge P(N)$, and similarly we can expand $\exists i.\, P(i)$ into $P(1) \vee \ldots \vee P(N)$. The quantifications are thus simply abbreviations and are used for convenience only.

- "There is always one leader". This is formalized by:

  $$\mathsf{G}\ (\exists i.\, leader_i \ \wedge\ (\forall j \neq i.\ \neg\, leader_j))$$

  Although this formula expresses the informally stated property, it will not be satisfied by any realistic protocol. One reason is that processes may be initially inactive, and thus no leader is guaranteed to exist initially. Besides, in a distributed system with asynchronous communication switching from one leader to another can hardly be made atomic. So, it is more realistic to allow the temporary absence of a leader. As a first attempt to do so, one could modify the above formula into:

  $$\mathsf{G}\,\mathsf{F}\ (\exists i.\, leader_i \ \wedge\ (\forall j \neq i.\ \neg\, leader_j))$$

  Problematic, though, is that this allows there to be more than one leader at a time temporarily – it is only stated that infinitely often there should be exactly one leader, but no statement is made about the moments at which this is not the case. For consistency reasons this is not desired. We therefore consider the following two properties.

- "There must always be at most one leader":

  $$\mathsf{G}\ (leader_i \ \Rightarrow\ \forall j \neq i.\ \neg\, leader_j)$$

- "There will be enough leaders in due time":

  $$\mathsf{G}\,\mathsf{F}\ (\exists i.\, leader_i)$$

  This property does not imply that there will be infinitely many leaders. It only states that there are infinitely many states at which a leader exists.

This requirement avoids the construction of a leader election protocol that never elects a leader. Such a protocol would fulfill the previous requirement, but is not desired for obvious reasons.

- "In the presence of an active process with a higher identity the leader will resign at some time":

$$\mathsf{G}\,(\forall i, j.\,((leader_i \;\wedge\; i < j \;\wedge\; \neg\, leader_j \;\wedge\; active_j) \;\Rightarrow\; \mathsf{F}\,\neg\, leader_i))$$

For reasons of efficiency it is assumed not to be desirable that a leader eventually resigns in presence of an inactive process that may participate at some unknown time in the future. Therefore we require j to be an active process.

- "A new leader will be an improvement over the previous one". This property requires that successive leaders have an increasing identity.

$$\mathsf{G}\,(\forall i, j.\,(leader_i \;\wedge\; \neg\,\mathsf{X}\, leader_i \;\wedge\; \mathsf{X}\,\mathsf{F}\, leader_j) \;\Rightarrow\; i < j)$$

Note that this requirement implies that a process that resigns once, will not become a leader any more.

3.7 Fairness

3.7.1 On the Notion of Fairness

An important aspect of reactive systems is fairness. We illustrate the concept of fairness by means of a frequently encountered problem in concurrent systems.

Example 3.10. Consider N processes $P_1, \ldots, P_N$ which require a certain service. There is one server process Server *that is expected to provide services to these processes. A possible strategy that* Server *can realize is the following. Check the processes starting with P_1, then P_2, and so on, and serve the first thus encountered process that requires service. On finishing serving this process, repeat this selection procedure once again starting with checking P_1.*

Now suppose that P_1 is continously requesting service. Then this strategy will result in Server *always serving P_1. Since in this way another process has to wait infinitely long before being served, this is called an unfair strategy. In a fair serving strategy it is required that the server eventually responds to any request by any one of the processes. For instance, a round-robin scheduling strategy where each process is only served for a limited amount of time is a fair strategy: after having served one process, the next (in the round-robin order) is checked and, if needed, served. (End of example.)*

When verifying concurrent systems one is often only interested in paths in which enabled transitions (statements) are executed in some "fair" manner. In Section 3.6.1, for instance, we treated a mutual exclusion algorithm for two processes. In order to prove the absence of individual starvation, the situation in which a process that wants to enter its critical section has to wait infinitely long, we want to exclude those paths in which the competitor process is always being selected for execution. This type of fairness is also known as *process fairness*, since it concerns the fair scheduling of the execution of processes. If we were to consider unfair paths when proving the absence of individual starvation we would usually fail, since there always exists an unfair strategy according to which some process is always neglected, and thus can never make progress. One might argue that such unfair strategy is unrealistic and should be avoided.

Process fairness is a particular form of fairness. In general, fairness assumptions are needed to prove liveness properties, properties of the type "something good will eventually happen". This is of vital importance if the Kripke structure to be checked contains non-determinism. Fairness is then concerned with resolving non-determinism in such a way that it is not biased to consistently ignore a possible option. In the above example, the scheduling of processes is non-deterministic: the choice of the next process to be executed (if there are at least two processes that can be potentially selected) is arbitrary. Another example where non-determinism occurs is in sequential programs where constructs like:

do true $\longrightarrow S_1$**else** true $\longrightarrow S_2$ **od**

are allowed. Both statements S_1 and S_2 are always enabled. An unfair mechanism might always choose statement S_1 to be executed, and as a consequence, a property that is established by executing statement S_2 is never met. Another prominent example where fairness is used to "resolve" non-determinism is in modeling concurrent processes by means of interleaving. Interleaving is equivalent to modeling the concurrent execution of two independent processes by enumerating all the possible orders in which activities of the processes can be executed (cf. Chapter 2).

In general, a fair path (or: computation) is characterized by the fact that certain fairness constraints are fulfilled.

3.7.2 Fairness Expressed in PLTL

Fairness constraints are used to rule out computations that are considered to be unreasonable for the system under consideration. In PLTL, fairness can be expressed as part of the property specification. The general format of a

property specification is:

$$\textit{fairness constraint} \Rightarrow \textit{desired property}$$

We will treat three different forms of fairness. Let Ψ be the desired property, such as absence of individual starvation, and Φ be the fairness constraint under consideration, like a process has to have its turn "regularly". Then we distinguish between the following forms of fairness:

- *Unconditional fairness.* A path is unconditionally fair with respect to Ψ if it satisfies:

  $$\mathsf{G}\,\mathsf{F}\,\Psi$$

  For instance, if Ψ denotes " a process enters its critical section" or "a process gets its turn", then a path is uncondionally fair with respect to these properties if they hold infinitely often, i.e., either a process enters its critical section infinitely often, or, in the other case, a process gets its turn infinitely often. Note that the fairness constraint Φ here is vacuously true, e.g., no condition (such as "a process is enabled") is expressed under which circumstances a process gets its turn infinitely often. This becomes more clear by reformulating $\mathsf{G}\,\mathsf{F}\,\Psi$ by:

  $$\text{true} \Rightarrow \mathsf{G}\,\mathsf{F}\,\Psi$$

  Unconditional fairness is sometimes referred to as *impartiality.*

- *Weak fairness.* A path is weakly fair with respect to Ψ and fairness constraint Φ if it satisfies:

  $$\mathsf{F}\,\mathsf{G}\,\Phi \Rightarrow \mathsf{G}\,\mathsf{F}\,\Psi$$

  For instance, a typical weak fairness requirement is:

  $$\mathsf{F}\,\mathsf{G}\,\mathit{enabled}(a) \Rightarrow \mathsf{G}\,\mathsf{F}\,\mathit{executed}(a)$$

  Weak fairness means that if an activity such as a, e.g., a transition or an entire process, is *continuously* enabled ($\mathsf{F}\,\mathsf{G}\,\mathit{enabled}(a)$), then it has to be executed infinitely often ($\mathsf{G}\,\mathsf{F}\,\mathit{executed}(a)$). A computation is weakly fair with respect to activity a if it is not the case that a is always enabled beyond some point without being taken beyond this point. Weak fairness is sometimes referred to as *justice.*

- *Strong fairness.* A path is strongly fair with respect to Ψ and fairness constraint Φ if it satisfies:

  $$\mathsf{G}\,\mathsf{F}\,\Phi \Rightarrow \mathsf{G}\,\mathsf{F}\,\Psi$$

The difference to weak fairness is that the premise $\mathsf{F}\,\mathsf{G}\,\Phi$ is replaced by $\mathsf{G}\,\mathsf{F}\,\Phi$. Strong fairness means that if an activity is *infinitely often* enabled – but not necessarily always, i.e., there may be periods during which Φ is not valid – then it will be executed infinitely often. A path is strongly fair with respect to activity a if it is not the case that a is infinitely often enabled without being taken beyond a certain point. Strong fairness is sometimes referred to as *compassion*.

An important question now is: given a verification problem, which fairness notion to use? Unfortunately, there is no clear answer to this question. Different forms of fairness do exist – the above is just a small, though important, fragment of all possible fairness notions – and there is no single favorite notion. For verification purposes, fairness constraints are crucial, though. Recall that fairness rules out certain "unreasonable" computations. If the fairness constraint is too strong, relevant computations may not be considered. In case a formula is found to be satisfied (for a Kripke structure), it might well be the case that some reasonable computation that is not considered (as it is ruled out by the fairness constraint), refutes this formula. On the other hand, if the fairness constraint is too weak, we may fail to prove a certain property as some unreasonable computations (that are not ruled out) refute it.

Example 3.11. Consider the following two processes that run in parallel and share an integer variable x that initially has value 0:

$$\begin{aligned} \textbf{process}\ \mathsf{Inc} &= \textbf{while}\ \langle\, x \geqslant 0\ \textbf{do}\ x := x+1 \,\rangle\ \textbf{od} \\ \textbf{process}\ \mathsf{Reset} &= x := -1 \end{aligned}$$

Recall that the pair of brackets $\langle \ldots \rangle$ embraces an atomic section, i.e., process Inc *performs the check whether x is positive abd the increment of x (if the guard holds) as one atomic step. Does this parallel program terminate, i.e., does the model underlying this program satisfy the formula* $\mathsf{F}\,\mathit{terminate}$*? When no fairness constraints are imposed, it is possible that process* Inc *is permanently executing, i.e., process* Reset *never gets its turn and the assignment $x = -1$ will not be executed. In this case, termination is thus not guaranteed, and the property is refuted. If, however, we require unconditional process fairness, then we are able to establish that*

$$(\mathsf{G}\,\mathsf{F}\,\mathsf{Inc}.\mathit{running} \wedge \mathsf{G}\,\mathsf{F}\,\mathsf{Reset}.\mathit{running}) \Rightarrow \mathsf{F}\,\mathit{terminate}$$

where the atomic proposition P.running is valid for process p when it has its turn. *(End of example.)*

Logically speaking, the relationship between the above introduced fairness notions is as follows:

$$((\mathsf{G}\,\mathsf{F}\,\Psi) \Rightarrow (\mathsf{F}\,\mathsf{G}\,\Phi \Rightarrow \mathsf{G}\,\mathsf{F}\,\Psi)) \wedge ((\mathsf{F}\,\mathsf{G}\,\Phi \Rightarrow \mathsf{G}\,\mathsf{F}\,\Psi) \Rightarrow (\mathsf{G}\,\mathsf{F}\,\Phi \Rightarrow \mathsf{G}\,\mathsf{F}\,\Psi))$$

A stronger fairness constraint rules out more paths, and thus allows a subset of the paths allowed by a weaker fairness constraint. This hierarchy can be exploited to choose an appropriate form of fairness in the following sense. Try first to prove the property at hand under the weakest fairness constraint. If the property is found to be valid, it also holds for any stronger fairness constraint. Proving that a system satisfies a property under no fairness constraint – unconditional fairness – implies that it does so under any fairness constraint.

3.8 Practical Use of PLTL

This section concludes this chapter with a discussion of some important categories of properties, both from a theoretical as from a practical perspective. Several classifications of properties do exist; we adopt the following rather classical one.

A *reachability* property expresses that some particular situation can be reached. Example reachability properties are: "a process can enter its critical section", or "the coffee machine can produce hot chocolate", or "the system can always return to its initial state". Often a *negated reachability* property is of interest, expressing that some undesired situation (such as a deadlock) cannot be reached. Reachability properties are of the type "there exists a path such that some scenario can be reached". These properties are hard to formulate in PLTL, as satisfaction in PLTL by definition ranges over all paths (cf. Definition 3.8) rather than over a subset of paths. Consequently, negated reachability properties can be expressed easily, e.g., $\mathsf{G}\,\neg\,(P@cs)$ states that process P can never occupy its critical section. The typical syntax of a negated reachability property is $\mathsf{G}\,\neg\,\Phi$ where Φ characterizes the undesirable situation. Reachability properties like "the system can always return to its initial state" cannot be expressed in PLTL.

A *safety* property expresses that, under certain conditions, something never occurs. Typically, a safety property states that some *bad* situation may never occur. A negated reachability property is a safety property and the negation of a safety property is a reachability property. An example safety property is "the coffee machine will never provide tea if the user requests coffee", or "it is never the case that two processes occupy their critical section simultaneously". The operator G describes safety properties, e.g., $\mathsf{G}\,\neg\,(P_1@cs \wedge P_2@cs)$ expresses that processes P_1 and P_2 can never occupy their critical section simultaneously. For expressing conditional safety properties the unless operator W is of importance,

e.g., the property "as long as the user does not provide a 25 cents coin, the coffee machine won't offer coffee" is expressed by $\neg\, \mathit{coffee}\, \mathsf{W}\, \mathit{coin}$. This should be read as "the machine does not provide coffee unless a coin is inserted into the machine". Recall that this is equivalent to the formula $\mathsf{G}\, \neg\, \mathit{coffee} \vee (\mathit{coin}\, \mathsf{U}\, \mathit{coffee})$. Note that the formula $\neg\, \mathit{coffee}\, \mathsf{U}\, \mathit{coin}$ does not express our intention, as this formulation requires that a *coin* will be inserted some day, thus not permitting a behaviour in which never a coin is provided.

A *liveness* property expresses that, under certain conditions, something will ultimately occur. Liveness properties are also called progress properties.[5] Typically, a liveness property describes that some *good* situation will occur in the end. Example liveness properties are "the coffee-machine will eventually provide coffee, after issuing a coin" (note the difference with the above safety property), or "the traffic light will become green", or "once red, the traffic light will become green". The latter two properties are formulated in PLTL by: $\mathsf{F}\, \mathit{green}$ and $\mathsf{G}\, (\mathit{red} \Rightarrow \mathsf{F}\, \mathit{green})$. Whereas the unless operator is convenient to describe conditional safety properties, the until operator is used for conditional liveness properties, since $\Phi\, \mathsf{U}\, \Psi$ is valid if Ψ is will hold eventually. Here, Ψ characterizes the good situation while Φ describes the condition under which this good situation has to be reached. Note that we have:

$$\Phi\, \mathsf{U}\, \Psi \equiv \mathsf{F}\, \Psi \vee \Phi\, \mathsf{W}\, \Psi$$

where the property $\Phi\, \mathsf{U}\, \Psi$ can be decomposed into the unconditional liveness property $\mathsf{F}\, \Psi$ and the safety property $\Phi\, \mathsf{W}\, \Psi$.

A *deadlock-freeness* property expresses that never a situation can be reached in which no progress is possible. This is of particular interest to reactive systems, as these typically execute ad infinitum. One is tempted to classify deadlock-freedom as a negated reachability problem: the deadlocked states are not reachable. This, however, requires an explicit characterization of the set of deadlocked states, which is not always easy [22].

A *fairness* property expresses that, under certain conditions, an event will occur (or will fail to occur) infinitely often. Fairness has been more extensively discussed in the previous section.

A classification of properties is not only of academic interest, but is also of high practical use. First of all, it leads to better structured requirements specifications and is a vehicle to reduce the number of omissions by checking "what are the safety properties" and "what are liveness properties"? Secondly, they help the user to formulate their property in terms of PLTL (or any other logic), as the different categories suggest – but are not uniquely determined by – certain

[5] This is, however, sometimes confusing, as for instance the property "the program will deadlock" is a progress property, although it states that a state can be reached from which no progress can be made.

syntactic formats. The latter aspect can be further extended by considering *specification patterns*, i.e., a generalized description of a commonly occurring requirement on the admissable paths that is (i) parameterizable (the pattern contains holes to be replaced by PLTL-formulas), (ii) high-level (no detailed knowledge of PLTL is needed) and (iii) is formalism-independent (translation of patterns to a logical formula can be automated). Table 3.4 lists the most commonly used specification patterns for PLTL, describes their scope, property category and their empirically established importance (by considering several case studies).

pattern	*category*	*PLTL-formula*	*frequency*
response	liveness	$\mathsf{G}(\Phi \Rightarrow \mathsf{F}\,\Psi)$	43.4 %
universality	safety	$\mathsf{G}\,\Phi$	19.8 %
absence	negated reachability	$\mathsf{G}\,\neg\,\Phi$	7.4 %
precedence	liveness	$\mathsf{G}(\neg\,\Phi\,\mathsf{W}\,\Psi)$	4.5 %
absence		$\mathsf{G}\,((\Phi \wedge \neg\,\Psi \wedge \mathsf{F}\,\Psi) \Rightarrow (\neg\,\Phi'\,\mathsf{U}\,\Psi))$	3.2 %
absence	safety	$\mathsf{G}(\Psi \Rightarrow \mathsf{G}\,\neg\,\Phi)$	2.1 %
existence	liveness	$\mathsf{F}\,\Phi$	2.1 %
			≈ 80 %

Table 3.4: Most commonly used specification patterns for PLTL [66]

3.9 Bibliographic Notes

Temporal logic. Temporal logic belongs to the class of modal logics whose origins go back to the field of philosophy. Although it is hard to point out the original works, it is clear that the works by Lewis [123] and Prior [154] were one of the first to use modal logic to give a meaning to the modalities "sometime" and "everywhere". The semantics that is most commonly used nowadays is due to Kripke [112]. The until-operator was introduced in the late sixties by Kamp [107] who also showed that this operator is more expressive than F and G alone. The introduction of temporal logic into computer science is by Pnueli in the late seventies in his seminal paper [151]. Surveys of the use of temporal logic in computer science include the works by Manna and Pnueli [131], Emerson [67], and Gotzhein [80]. Besides the specification and verification of properties over computer systems, temporal logic is used in artificial intelligence to formally describe the knowledge of agents. This idea is due to Hintikka; a recommendable introduction can be found in [102, Chapter 5].

PLTL. Gabbay, Pnueli, Shelah and Stavi [78] presented a complete axiomatization for PLTL and a proof of the decidability of its satisfiability problem. The complexity of deciding the satisfiability problem for PLTL (and some variants thereof) has been reported by Sistla and Clarke [168]. Gabbay, Pnueli, Shelah and Stavi [77] showed that the inclusion of past operators – as in the original proposal by Kamp – does not enlarge the expressiveness of PLTL by

presenting an algorithm that translates any PLTL-formula with past operators into an equivalent (but larger) ordinary PLTL-formula. (This result carries over to CTL and μ-calculus.) Lichtenstein, Pnueli and Zuck [125] argued that past operators are helpful for specification convenience and modular reasoning. Recently, Laroussinie, Markay and Schnoebelen [120] have shown that PLTL with past operators is exponentially more succinct than ordinary PLTL. That is, there do exist properties that can be expressed as a past-PLTL formula of length n, for which all equivalent formulations in ordinary PLTL are of length 2^n.

Assertional verification. Establishing software correctness has been a significant research field since the early days of computing. The origins of a sound mathematical approach towards program correctness can be traced back to Turing in 1949 [179, 141]. It lasted until the mid 1960s before these initial ideas resulted in constructive methods for verifying flowcharts [74], a restrictive form of programs, and – due to the seminal work by Hoare [93] – for verifying while-programs. This work received a lot of attention, and many Hoare-style proof systems have been developed since then for various programming language constructs. Generalizations of this approach to parallel programs have initially been developed by Owicki and Gries [146] and Lamport [116], while work on assertional verification of distributed programs has taken place by several researchers since the early eighties, such as Apt, Francez and de Roever [14]. Overviews of Hoare-style proof systems can be found in the survey papers by Apt [12, 13]. The book by Apt and Olderog [16] gives an excellent treatment of assertional verification of sequential and parallel (and distributed) programs. The work by Dijkstra [63], Gries [81] and others have shown that the proof rules for a posteriori verification of sequential programs are also quite useful in systematic program development. The Unity-approach by Chandy and Misra [41] adopts this approach for parallel programs using a simple variant of temporal logic.

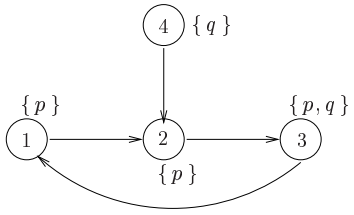
Assertional verification with LTL. Using a set of proof rules for temporal logic, assertional verification can be carried out in the same way as the verification of sequential programs using predicate logic. This was originally proposed by Pnueli [151], and has been taken up by, amongst others, Hailpern [85] to check concurrent programs. A tutorial introduction to assertional reasoning based on temporal logic is given by Udaya Shankar [164]. An extensive treatment of deductive systems for (first-order and propositional) LTL can be found in Manna and Pnueli [132]. The disadvantages of the proof verification method are similar to those for checking parallel systems: proofs are lengthy, detailed, and tedious, are thus quite labor-intensive, and require a substantial user involvement. An approach for reactive systems for which some tool support is available, is Lamport's TLA (Temporal Logic of Actions [118]). This approach allows one to specify requirements and system behavior in the same notation.

Property classification. The classification of properties into safety and liveness

properties is due to Lamport [116]. Pnueli pointed out (again in [151]) that in particular liveness properties require a specification language, such as temporal logic, to be formalized. The characterization of safety and liveness properties in temporal logic has received quite some attention in the literature; some initial work has been done by Sistla [166], and a full characterization has been given by Manna and Pnueli [130]. For an overview and a more detailed classification, consult Manna and Pnueli's book [131, Chapter 4] or the overview by Sistla [167]. Fairness has been extensively described by Francez [76]. The practical specification of different types of properties in PLTL (as well as in branching temporal logic) has recently been described by Bérard *et al.* [22]; we have largely adopted the description of property classes (cf. Section 3.8) from [22]. A pattern-based approach towards the specification properties in temporal logic has been proposed by Dwyer, Avrunin and Corbett [65]. These authors empirically established [66], by considering over 550 examples of property specifications taken from case studies, that indeed most of the properties fall into a (rather small) set of patterns. More information can be found at: `www.cis.ksu.edu/santos/spec-patterns/`

3.10 Exercises

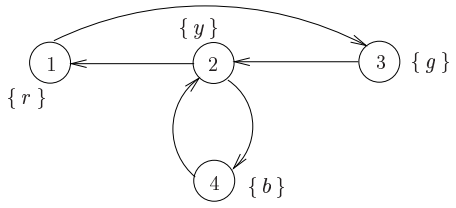
EXERCISE 3.1. Consider the following Kripke structure consisting of four states that are labelled with atomic propositions from the set $\{ p, q \}$.



Indicate for each of the following PLTL-formulas the set of states for which these formulas are valid:

1. $\mathsf{X}\, p$
2. $\mathsf{X}\,\mathsf{X}\,\mathsf{X}\, p$
3. $\mathsf{G}\, p$
4. $\mathsf{G}\,\mathsf{F}\, q$
5. $\mathsf{G}\,(p\,\mathsf{U}\, q)$
6. $\mathsf{F}\,(q\,\mathsf{U}\, p)$

EXERCISE 3.2. Consider the following Kripke structure that consists of four states. The following atomic propositions are used: r (red), y (yellow), g (green) and b (black). The model is intended to describe a traffic light that is able to blink yellow.



You are requested to indicate for each of the following PLTL-formulas the set of states for which these formulas are valid:

1. $\mathsf{X}\,\mathsf{X}\,y$
2. $\mathsf{X}\,\mathsf{X}\,r$
3. $\mathsf{F}\,y$
4. $\mathsf{G}\,\mathsf{F}\,y$
5. $\mathsf{F}\,g$
6. $\mathsf{G}\,\mathsf{F}\,g$
7. $\mathsf{G}\,\neg\,b$
8. $\mathsf{F}\,\neg\,b$
9. $\neg\,y\,\mathsf{U}\,y$
10. $\neg\,r\,\mathsf{U}\,g$
11. $\neg\,b\,\mathsf{U}\,b$
12. $b\,\mathsf{U}\,\neg\,b$

EXERCISE 3.3. Suppose we have two users, *Peter* and *Betsy*, and a single printer device *Printer*. Both users perform several tasks, and every now and then they want to print their results on the *Printer*. Since there is only a single printer, only one user can print a job at a time. Suppose we have the following atomic propositions for *Peter* at our disposal:

- *Peter.request* ::= indicates that *Peter* requests usage of the printer;
- *Peter.use* ::= indicates that *Peter* uses the printer;
- *Peter.release* ::= indicates that *Peter* releases the printer.

For *Betsy* similar predicates are defined. Specify in PLTL the following properties:

1. Mutual exclusion, i.e., only one user at a time can use the printer.
2. Finite time of usage, i.e., a user can print only for a finite amount of time.
3. Absence of individual starvation, i.e., if a user wants to print something, he/she eventually is able to do so.
4. Absence of blocking, i.e., a user can always request to use the printer
5. (More involved) Alternating access, i.e., users must strictly alternate in printing.

EXERCISE 3.4. Check for the following PLTL-formula whether they are (i) satisfiable, and/or (ii) valid:

1. $\mathsf{X}\,\mathsf{X}\,p \;\Rightarrow\; \mathsf{X}\,p$
2. $\mathsf{X}\,(p \vee \mathsf{F}\,p) \;\Rightarrow\; \mathsf{F}\,p$
3. $\mathsf{G}\,p \;\Rightarrow\; \neg\,\mathsf{X}\,(\neg\,p \wedge \mathsf{G}\,\neg\,p)$
4. $\mathsf{G}\,p\,\mathsf{U}\,\mathsf{F}\,q \;\Rightarrow\; \mathsf{G}\,(p\,\mathsf{U}\,\mathsf{F}\,q)$

5. $\mathsf{F}\, q \;\Rightarrow\; (p\,\mathsf{U}\, q)$

EXERCISE 3.5. Prove or disprove whether $\mathsf{G}\,(\Phi \;\Rightarrow\; \mathsf{X}\,\Phi)$ equals $\mathsf{G}\,(\Phi \;\Rightarrow\; \mathsf{G}\,\Phi)$. Denote in each step of your proof which result or definition you are using, or provide a counterexample.

EXERCISE 3.6. Check whether the following pairs of PLTL-formula are equivalent (p and q are atomic propositions). If not, check in which direction the implication holds. If one of the implications is invalid, provide an example Kripke structure that demonstrates this.

1. $\mathsf{X}\,\mathsf{F}\, p \Leftrightarrow \mathsf{F}\,\mathsf{X}\, p$
2. $(\mathsf{F}\,\mathsf{G}\, p) \wedge (\mathsf{F}\,\mathsf{G}\, q) \Leftrightarrow \mathsf{F}\,(\mathsf{G}\, p \wedge \mathsf{G}\, q)$
3. $(p\,\mathsf{U}\, q)\,\mathsf{U}\, q \Leftrightarrow p\,\mathsf{U}\, q$
4. $(p\,\mathsf{U}\, q) \wedge (q\,\mathsf{U}\, r) \Leftrightarrow (p\,\mathsf{U}\, r)$
5. $\mathsf{G}\, p \;\vee\; \mathsf{F}\, q \Leftrightarrow (\mathsf{G}\, p) \;\vee\; (p\,\mathsf{U}\, q)$

EXERCISE 3.7. Check whether the following pairs of PLTL-formula are equivalent (p, q and r are atomic propositions). If not, check in which direction the implication holds. If one of the implications is invalid, provide an example Kripke structure that demonstrates this.

1. $\mathsf{F}\, p \Leftrightarrow \mathsf{X}\, p$
2. $\mathsf{F}\,\mathsf{G}\, q \Leftrightarrow \mathsf{G}\,\mathsf{F}\, q$
3. $\mathsf{X}\,\mathsf{G}\, p \Leftrightarrow \mathsf{G}\,\mathsf{X}\, p$
4. $(p\;\mathsf{U}\, q \wedge p\;\mathsf{U}\, r) \Leftrightarrow p\;\mathsf{U}\,(q \wedge r)$
5. $(\mathsf{G}\,\mathsf{F}\, p \wedge \mathsf{G}\,\mathsf{F}\, q) \Leftrightarrow \mathsf{G}\,\mathsf{F}\,(p \wedge q)$

EXERCISE 3.8. Consider a lift system that services $N > 0$ floors numbered 0 through $N-1$. There is a lift door at each floor with a call-button and an indicator light that signals whether or not the lift has been called. In the lift cabin there are N send-buttons (one per floor) and N indicator lights that inform to which floor(s) is sent. For simplicity consider $N = 4$. Present a set of atomic propositions – try to minimize the number of propositions – that are needed to describe the following properties of the lift system as PLTL-formulas and give the corresponding PLTL-formulas:

1. The doors are "safe", i.e., a floor door is never open if the cabin is not present at the given floor.
2. A requested floor will be served sometime.
3. Again and again the lift returns to floor 0.

4. When the top floor is requested, the lift serves it immediately and does not stop on the way there.
5. The cabin is motionless unless there is some request.

EXERCISE 3.9. Let Φ and Ψ be PLTL-formulas. Consider the following new operators for PLTL:

1. "At next" $\Phi \mathsf{AX}\, \Psi$: at the next time where Ψ holds, also Φ does.
2. "While" $\Phi \,\mathsf{W}\, \Psi$: Φ holds as least as long as Ψ does.
3. "Before" $\Phi \,\mathsf{B}\, \Psi$: if Ψ holds sometime, Φ does so before.

Show that adding these operators to PLTL does not increase the expressivity of PLTL, i.e., find for each of the above formulae an equivalent (ordinary) PLTL-formula.

Chapter 4

Automata

An important strategy to model check PLTL is based on finite-state automata that accept infinite words. This chapter introduces automata on finite words, their relationship to regular languages and some elementary results on such automata. The main part of the chapter is devoted to Büchi automata, automata on infinite words, and some algorithms on these automata that are relevant to model-checking PLTL.

4.1 Automata on Finite Words

Automata are often used in computer science as models to describe the behaviour of systems. They consist of a set of *states* that are connected to each other by *transitions*. A state describes some information about a system at a certain moment of its behaviour. For instance, a state of a traffic light indicates the current colour of the light. Similarly, a state of a sequential computer program indicates the current values of all program variables together with the current value of the program counter that indicates the next statement to be executed. Transitions specify how the system can evolve from one state to another. In case of the traffic light a transition may indicate a switch from one colour to another, whereas for the sequential program a transition typically corresponds to the execution of a statement.

4.1.1 Regular Languages

Let Σ be an *alphabet*, i.e., a finite set of symbols ranged over by a, b, c, and so forth. Finite *words* (or strings) are finite sequences over an alphabet, e.g., some words over $\Sigma = \{ a, b \}$ are ε (the empty word), ab and $aaabb$. A set of words

over an alphabet is called a *language*, and is ranged over by $\mathcal{L}$ (and primed and subscripted versions thereof).

Important operations on words are concatenation and repetition. *Concatenation* takes two words and "glues them together" to construct a new word. It is denoted by juxtaposition. For instance, the concatenation of the words ba and aab yields the word $baaab$. The concatenation of a word with itself is denoted by squaring, e.g., $(ab)^2$ equals $abab$; this is generalised in a straightforward manner to arbitrary numbers of concatenations, e.g., $(ab)^5$ equals $ababababab$, i.e., the word ab 5 times in a row. *Repetition* (denoted by the Kleene star *) of a word yields a language, i.e., a set of words which contains zero or more (but finite) repetitions of it. Formally, for word σ we have $\sigma^* = \bigcup_{i=0}^{\infty} \sigma^i$. For instance, $(ab)^*$ denotes the set $\{ \varepsilon, ab, abab, ababab, \dots \}$. Note that the empty word ε is included. This is precisely the difference with the slight variant of the Kleene star, denoted $^+$, defined by $\sigma^+ = \bigcup_{i=1}^{\infty} \sigma^i$, or, equivalently, $\sigma^+ = \sigma^* - \{ \varepsilon \}$. For instance, $(ab)^+$ denotes the set $\{ ab, abab, ababab, \dots \}$.

Concatenation is lifted to languages in a natural way as a point-wise extension of concatenation on words. The same applies to repetition. For languages $\mathcal{L}$ and $\mathcal{L}'$ we have $\mathcal{L}.\mathcal{L}' = \{ \sigma\sigma' \mid \sigma \in \mathcal{L}, \sigma' \in \mathcal{L}' \}$ and $\mathcal{L}^* = \bigcup_{i=0}^{\infty} \mathcal{L}^i$ where $\mathcal{L}^i$ denotes concatenation of i times $\mathcal{L}$. Thus, for instance, for $\mathcal{L} = \{ a, ab \}$ and $\mathcal{L}' = \{ \varepsilon, bbb \}$ we have that $\mathcal{L}.\mathcal{L}'$ equals the language $\{ a, ab, abbb, abbbb \}$ whereas $\mathcal{L}^2 = \{ aa, aab, abab, aba \}$.

Regular languages are an important class of languages. They can be constructed inductively as follows. A regular language over an alphabet can be built from the empty set, the set containing the empty sequence, and the sets containing a single symbol of the alphabet using the operations of set union, concatenation and repetition (Kleene star). Formally, regular languages over alphabet Σ satisfy the following rules:

1. $\varnothing$ and $\{ \varepsilon \}$ are regular languages over Σ
2. $\{ a \}$ is a regular language over Σ, for any $a \in \Sigma$
3. if $\mathcal{L}$ and $\mathcal{L}'$ are regular languages over Σ, then so are $\mathcal{L}.\mathcal{L}'$, $\mathcal{L} \cup \mathcal{L}'$ and $\mathcal{L}^*$
4. anything else is not a regular language.

For instance, $\mathcal{L} = \{ a, b \}^*.\{ b \}.\{ b \}.\{ a, b \}^*$ is a regular language over $\Sigma = \{ a, b \}$ that consists of all words that contain the subword bb. Regular languages are usually described by *regular expressions*, e.g., $\mathcal{L}$ is described by the regular expression $(a|b)^*bb(a|b)^*$ where $|$ denotes a choice. It states that any word that belongs to $\mathcal{L}$ is constructed as the concatenation of (i) a (possibly empty) word of as and bs in arbitrary order, (ii) the word bb, and (iii) a (possibly empty) word of as and bs in arbitrary order.

4.1.2 Finite-State Automata

In the rest of this book, we will often use regular expressions to describe regular languages. An alternative way to describe these languages is to use finite-state automata.

Definition 4.1. (**Finite-state automaton**)
A finite-state automaton (FSA) A is a tuple $(\Sigma, S, I, \longrightarrow, F)$ where:

- Σ is an alphabet
- S is a finite set of states
- $I \subseteq S$ is a set of initial states
- $\longrightarrow \subseteq S \times \Sigma \times S$ is a labelled transition relation, and
- $F \subseteq S$ is a set of *accept* states.

Σ defines the symbols on which the automaton is defined. The (possibly empty) set I defines the states in which the automaton may start. Notation: we write $s \xrightarrow{a} s'$ if and only if $(s, a, s') \in \longrightarrow$. The basic concept is that the automaton starts in one of the states in I, and then is feeded with a sequence of symbols (each called an input symbol) from the alphabet Σ. After reading an input symbol, the automaton changes state according to the transition relation $\longrightarrow$. Intuitively, $s \xrightarrow{a} s'$ denotes that the automaton moves from state s to state s' on input symbol a. When all input has been read (and is correct), the automaton is in one of the states in the set F, the accept states.

Example 4.1. An example FSA is depicted in Figure 4.1. Here we have $\Sigma = \{ a, b, c \}$, $S = \{ s_1, s_2, s_3 \}$, $I = \{ s_1 \}$, *transition function* $s_1 \xrightarrow{a} s_2$, $s_1 \xrightarrow{c} s_1$, $s_2 \xrightarrow{b} s_3$, $s_3 \xrightarrow{b} s_1$ *and* $s_3 \xrightarrow{b} s_3$ *and set of accept states* $F = \{ s_3 \}$. *The drawing conventions for FSA are the same as for labelled transition systems. Accept states are distinguished from other states by drawing them with two circles.* *(End of example.)*

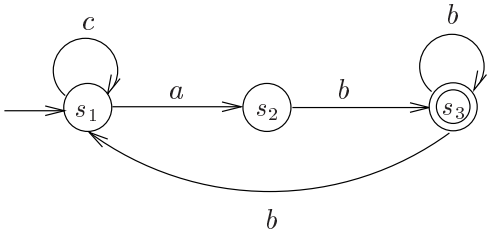


Figure 4.1: An example FSA

Definition 4.2. (**Run and word of an FSA**)
A *run* of FSA $A = (\Sigma, S, I, \longrightarrow, F)$ is a finite sequence of states $\sigma = s_0\, s_1\, \ldots\, s_n$ such that $s_0 \in I$ and $s_i \xrightarrow{a_i} s_{i+1}$ for all $0 \leqslant i < n$ for some $a_i \in \Sigma$. Run σ is called *accepted* by A if and only if $s_n \in F$.

A finite word $w = a_0\, a_1\, \ldots\, a_{n-1} \in \Sigma^*$ is *accepted* by A if and only if there exists an accepting run $\sigma = s_0\, s_1\, \ldots\, s_n$ such that $s_i \xrightarrow{a_i} s_{i+1}$ for $0 \leqslant i < n$.

The language accepted by A, denoted $\mathcal{L}(A)$, is the set of finite words accepted by A, i.e., $\mathcal{L}(A) = \{\, w \in \Sigma^* \mid w \text{ is accepted by } A \,\}$.

That is, a run is a finite sequence of states, starting from an initial state such that each state in the sequence can be reached via the transition relation $\longrightarrow$ from its predecessor in the sequence. A run is accepting if it ends in an accept state. If $F = \varnothing$ there are no accepting runs, and thus $\mathcal{L}(A) = \varnothing$ in this case. An FSA can be considered as an acceptor for the language that contains the set of words induced by its accepting runs. Note that by definition an FSA cannot perform any transition when its current state does not have an outgoing transition that is labelled with the current input symbol. That is to say, if an automaton is feeded with an input symbol that does not match a label of any of the outgoing transitions of the current state, the automaton is stuck and no further progress can be made. This situation can be made explicit by adding an extra state ("stuck") to the FSA, and equipping each state with a transition to that state for each input symbol that it cannot recognise.

Example 4.2. Example runs of the automaton in Figure 4.1 are s_1, $s_1\, s_2$, $s_1\, s_1\, s_1$, $s_1\, s_2\, s_3$, *and* $s_1\, s_2\, s_3\, s_1\, s_1\, s_1\, s_2\, s_3$. *Accepting runs are runs that finish in the accept state* s_3, *for instance,* $s_1\, s_2\, s_3$, $s_1\, s_1\, s_2\, s_3$, *and* $s_1\, s_2\, s_3\, s_1\, s_1\, s_1\, s_2\, s_3$. *The accepted words that correspond to these accepting runs are respectively, ab, cab, and abbccab. The word cca is, for instance, not accepting, since there is not an accepting run that generates cca. (End of example.)*

Each accepted word by our example automaton in Figure 4.1 consists of zero or more times a symbol c, then an a, and one or more bs (to reach the accept state). This pattern of cs, a, and bs can be repeated a finite number of times after having consumed at least two bs. Formally, the language accepted by this automaton is characterized by the regular expression $c^*ab \mid (c^*abb^+)^+$.

It is not by accident that the language accepted by this automaton turns out to be a regular language, since Kleene showed that:

Theorem 4.1.
Language $\mathcal{L}$ *is regular if and only if there exists an FSA* A *such that* $\mathcal{L} = \mathcal{L}(A)$.

Thus for every regular expression (which describes a regular language), there exists a finite-state automaton that only accepts this regular expression and nothing else. Given a regular expression, this automaton can be constructed in an automated manner. Vice versa, algorithms do exist that given a finite-state automaton construct the regular language it accepts. An automaton may have several accept states each of which exhibits the acceptance of the set of words that end in that state. The regular language is determined by simply taking the union of these sets of words.

As a consequence of the above theorem, languages accepted by finite-state automata are equally expressive as regular languages. This result also shows the limitation of FSA, as it entails that for a non-regular language such as, e.g., $\{ a^n b^n \mid n \geqslant 0 \}$, there does not exist an FSA that accepts it. Intuitively, this is not surprising, since in order to be able to recognise this language, one need to be able to count the number of as so as to be able to determine the number of bs that need to follow. (This non-regular language is a so-called context-free language, whose expressiveness coincides with push-down automata, an extension of FSA where states are equipped with a stack of input symbols.)

4.1.3 Deterministic Automata

An FSA may be non-deterministic, i.e., it may have several initial states allowing A to start differently, and the transition function may specify various possible successor states for a given state and a given input symbol. For instance, the example automaton in Figure 4.1 is non-deterministic since the successor state of state s_3 is not uniquely defined if the input symbol is b. In that situation, the automaton may autonomously decide to either stay in state s_3 (and accept the corresponding word), or to loop back to its initial state. Let $\mathit{Succ}(s, a)$ denote the set of successor states of state s on input symbol a, i.e., $\mathit{Succ}(s, a) = \{ s' \mid s \xrightarrow{a} s' \}$.

Definition 4.3. (**Deterministic FSA**)
FSA $A = (\Sigma, S, I, \longrightarrow, F)$ is *deterministic* if and only if $|I| = 1$ and $|\mathit{Succ}(s, a)| \leqslant 1$ for all states $s \in S$ and all symbols $a \in \Sigma$. Otherwise, A is called *non-deterministic*.

Stated in words, a FSA is deterministic if it has a single initial state and if for each symbol the successor state of each state is uniquely defined. For each accepted word by a deterministic FSA there is exactly one run that corresponds to this word. At first sight, this might restrict the expressiveness of deterministic automata, but this is not the case: for any regular language there exists a deterministic automaton that accepts it.

Regular languages exhibit some interesting closure properties, e.g., the union of

two regular languages is a regular language. The same applies to intersection, concatenation and Kleene star (repetition). Later on in this chapter we will present the operation on automata that corresponds to the intersection of the languages they accept. Interestingly enough, regular languages are also closed under complementation, i.e., if $\mathcal{L}$ is regular over alphabet Σ, then $\overline{\mathcal{L}} = \Sigma^* - \mathcal{L}$ is also regular. Complementation of regular languages corresponds to the following operation on deterministic automata. For deterministic FSA $A = (\Sigma, S, I, \longrightarrow, F)$, the FSA $\overline{A} = (\Sigma, S, I, \longrightarrow, S - F)$ accepts the regular language $\overline{\mathcal{L}(A)}$. Thus, in order to complement a deterministic automaton we just have to complement the acceptance condition. As A is deterministic to each input word exactly one accepting run corresponds. By construction such word will not be accepted by $\overline{A}$. Note that this does not work for non-deterministic automata A: as such automaton might have several runs that accept a given input word, it does not suffice that some runs of $\overline{A}$ to reject (i.e., not accept) the word, but this should hold for all possible runs. It turns out that in order to complement a non-deterministic automaton, one first needs to make it deterministic.

Example 4.3. Consider the deterministic FSA depicted in Figure 4.2(a). Its

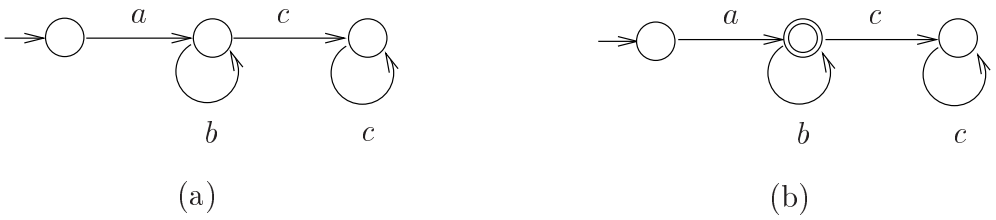


Figure 4.2: Complementing a deterministic FSA

language equals ab^*c^+. For instance, for word abbc there is a single corresponding accepting run, namely $s_1\, s_2\, s_2\, s_3$. The complement of this automaton is given in Figure 4.2(b). Note that abbc is not accepted by the complemented automaton. If we apply the same mechanism to the simple non-deterministic FSA of Figure 4.3(a) we obtain the automaton in Figure 4.3(b). This construction does, however, not correspond to the complement as, for instance, both automata accept the word ab. (End of example.)

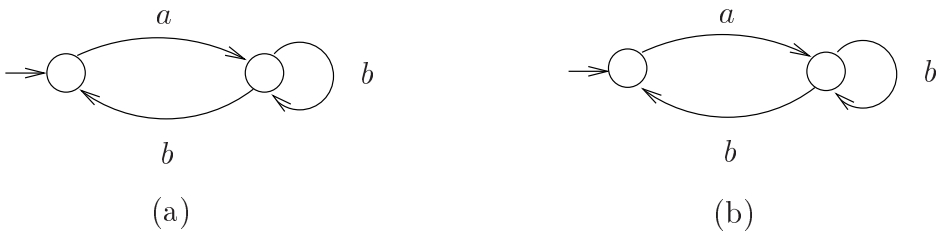


Figure 4.3: Naively complementing a non-deterministic FSA

4.2 Algorithms for Automata on Finite Words

This section presents several algorithms for finite-state automata. As we will see in Chapter 5, modifications of some of these algorithms for automata on infinite words are instrumental to model-checking PLTL. Algorithms to determinize non-deterministic FSA, to determine the synchronous product of two FSA, to carry out a depth-first search, and to check for emptiness are covered.

4.2.1 Determinization

Although deterministic and non-deterministic automata seem at first sight to be rather different – a non-deterministic FSA can have several runs on a given input word whereas a deterministic one can have maximally one – it turns out that there is a very strong relationship between them. Before giving a precise characterisation of this relationship we need the following concept. Two automata are called equivalent if they accept the same language:

Definition 4.4. (**Equivalence of FSA**)
FSA A and A' are *equivalent*, denoted $A \equiv A'$, if and only if $\mathcal{L}(A) = \mathcal{L}(A')$.

This notion of equivalence allows to express the relationship between deterministic and non-deterministic automata in the following way:

Theorem 4.2.
For each non-deterministic FSA A there exists a deterministic FSA A' (on the same alphabet as A) such that $A \equiv A'$.

Example 4.4. Consider again the example FSA from Figure 4.1. This automaton is non-deterministic as state s_3 has two possible successors on input symbol b. The FSA depicted in Figure 4.4 is equivalent to our example automaton, but

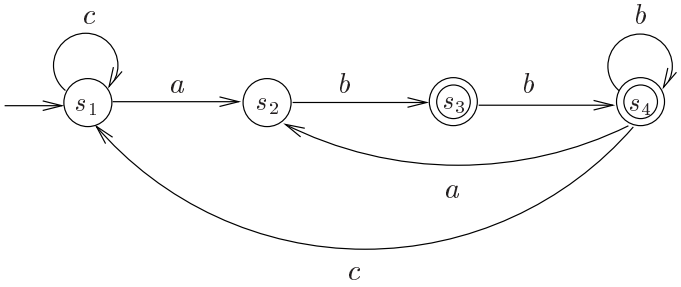


Figure 4.4: Deterministic FSA that is equivalent to the FSA of Figure 4.1

```
function nfsa2dfsa(A : FSA) : FSA;
(* pre: A is the FSA (Σ, S, I, —→, F) *)
begin var S', I', F', R : set of (set of States),
          —→' : set of Transitions,
          U, V : set of States;
      S', I', R, F', —→' := {I}, {I}, {I}, ∅, ∅;
      while R ≠ ∅
      do let U in R;                          (* pick an unexplored new state *)
         R := R − U;
         for each a ∈ Σ                       (* determine the a-successors of U *)
         do V := ∅;
            for each s ∈ U do V := V ∪ Succ(s, a) od;
            if V ∉ S' then S', R := S' ∪ V, R ∪ V fi;        (* V is new *)
            —→' := —→' ∪ {U —a→ V};                          (* add transition *)
         od;
      od;
      F' := {U ∈ S' | F ∩ U ≠ ∅};             (* determine new accept states *)
return (Σ, S', I', —→', F');
(* post: A' = (Σ, S', I', —→', F') is a deterministic FSA such that A ≡ A' *)
end
```

Table 4.1: Subset construction algorithm

is deterministic. It is constructed from the FSA of Figure 4.1 by introducing a new state (s_4) that has the same capabilities as the former states s_3 and s_1 together. In this way, the former transitions $s_3 \xrightarrow{b} s_3$ and $s_3 \xrightarrow{b} s_1$ are now represented by a single transition $s_3 \xrightarrow{b} s_4$. It is left to the reader to check that the two automata indeed accept the same language. (End of example.)

The crux of the transformation of our example FSA into an equivalent deterministic one is the grouping of target states that can be reached from a state by the same input symbol. In this way, states of the new, deterministic automaton can be considered as subsets of states of the original, non-deterministic automaton. The construction – originally due to Rabin and Scott – is therefore also known as the *subset construction algorithm*. The algorithm starts with the state I. In each iteration of the algorithm, a set is taken that has not been considered yet, and all its successors for a given symbol (from the alphabet Σ) are determined. These successors constitute a single state in the new automaton. That is to say, for set of states $\{ s_1, \ldots, s_n \}$ and symbol a, the set of a-successors is the union of the set of a-successors of s_i in the original non-deterministic automaton ($0 < i \leqslant n$). As in principle any subset of states of the orginal non-deterministic automaton (with n states, say) may yield a state in the new deterministic automaton, in worst case the number of states in the resulting deterministic FSA can be exponential: $\mathcal{O}(2^n)$. In practice, however, it turns out that this upperbound rarely occurs.

Regular languages and deterministic FSA have the same expressive power, i.e., for each regular language a deterministic FSA can be given that accepts pre-

cisely it, and, reversely, each accepting language by a deterministic FSA is regular. This does not imply that there exists a unique deterministic FSA that accepts a given regular language. In general, there may be several equivalent deterministic FSA. However, for each regular language there exists a unique *minimal* deterministic FSA (upto isomorphism) that accepts it. The algorithm to minimise a given deterministic FSA with n states and m transitions into its equivalent minimal deterministic FSA is based on partition-refinement and takes $\mathcal{O}(m \cdot \log n)$ time in worst case. The details of the algorithm are outside the scope of this chapter, but are very similar to the bisimulation minimisation algorithm that is covered in Chapter 14.

4.2.2 Synchronous Product

As stated before, regular languages are closed under intersection. Intersection of regular languages corresponds to a product construction of automata, i.e., a parallel composition of FSA in which both automata need to synchronise on all symbols. Both FSA are thus fully synchronised. This is formally defined by:

Definition 4.5. (**Synchronous product of FSA**)
For FSA $A_i = (\Sigma, S_i, I_i, \longrightarrow_i, F_i)$, with i=1, 2, the *product automaton* $A_1 \times A_2 = (\Sigma, S_1 \times S_2, I_1 \times I_2, \longrightarrow, F_1 \times F_2)$ where $\longrightarrow$ is the smallest relation defined by:

$$(s_1, s_2) \xrightarrow{a} (s_1', s_2') \text{ if and only if } s_1 \xrightarrow{a}_1 s_1' \text{ and } s_2 \xrightarrow{a}_2 s_2'.$$

It follows that this product construction of automata corresponds indeed to the intersection of their accepting languages, i.e., $\mathcal{L}(A_1 \times A_2) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$. This can be seen as follows. First consider $\mathcal{L}(A_1 \times A_2) \subseteq \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$. Let $(s_0, s_0')\,(s_1, s_1') \ldots (s_n, s_n')$ be an accepting run of $A_1 \times A_2$. Thus, state $(s_n, s_n') \in F_1 \times F_2$. In order to reach (s_n, s_n') it must be possible to reach s_n in A_1 via the run $s_0\, s_1 \ldots s_n$ which is an accepting run of A_1 since $s_n \in F_1$. Similarly, $s_0'\, s_1' \ldots s_n'$ is an accepting run of A_2 with $s_n' \in F_2$. Thus, any accepting run of $A_1 \times A_2$ is an accepting run of A_i when projected on a run of A_i for i=1, 2. For the reverse direction, i.e., $\mathcal{L}(A_1) \cap \mathcal{L}(A_2) \subseteq \mathcal{L}(A_1 \times A_2)$ the reasoning is straightforward. Let $s_0\, s_1 \ldots s_n$ be an accepting run of A_1 and $s_0'\, s_1' \ldots s_n'$ be an accepting run of A_2 such that the corresponding accepted words coincide. By Definition 4.5 it then follows that $(s_0, s_0')\,(s_1, s_1') \ldots (s_n, s_n')$ is an accepting run of $A_1 \times A_2$ that induces the same accepted word.

4.2.3 Depth-First Search

For many applications, such as detecting whether a given automaton contains a cycle or computing its accept states, a systematic way of visiting all states

in the automaton is necessary. Such algorithms are called *traversal* algorithms. For the purpose of traversal algorithms it is convenient to consider an FSA $A = (\Sigma, S, I, \longrightarrow, F)$ as a directed graph with S as the set of vertices and $\longrightarrow$ as the edge relation, i.e., there is an edge from s to s' if and only if there is a transition $s \xrightarrow{a} s'$ in A for some input symbol $a \in \Sigma$.

An important characteristic of traversal algorithms is the order in which states are visited. A frequently encountered visit order is *depth-first*; the corresponding search algorithm is called depth-first search. The basic depth-first search strategy is to start with an initial state and searching the entire graph by visiting states as far away from the selected initial state as quickly as possible. When we represent the visit order by means of a tree, then the states are visited according to a preorder traversal of the tree. The following example shows how a depth-first search works.

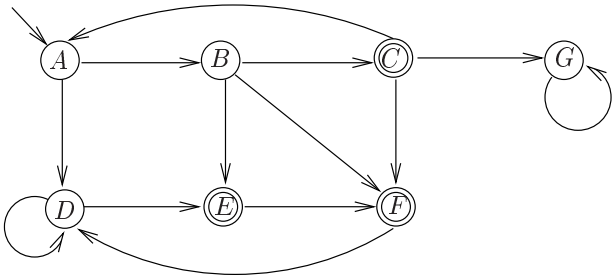


Figure 4.5: An example finite-state automaton

Example 4.5. Consider the graph of the FSA as depicted in Figure 4.5 and suppose we want to determine its accept states. Assume that the visiting order of searching the automaton equals $ADEFBCG$, i.e., first state A is visited, then state D, and so on. The order of depth-first search exploration is $FEDGCBA$. This is the order in which the depth-first search is finished in a state. Thus, the successor states of F are all explored first, then those of E, and so on. Given that C, E and F are accept states, the sequence of found accept states equals FEC, the order of depth-first exploration projected on the accept states. (End of example.)

The pseudo-code for the depth-first search algorithm is given in Table 4.2. As long as *CurPath* is non-empty the first state of that sequence is selected (as state s), and it is checked whether all its successors $Succ(s) = \bigcup_{a \in \Sigma} Succ(s, a) = \{ s' \mid \exists a.\, s \xrightarrow{a} s' \}$ have been visited before (i.e., are included in the sequence *DfsPath*). In that case, state s is removed from *CurPath*, and the traversal is continued. Otherwise, the newly encountered state is put in front of both *CurPath* and *DfsPath*, and the traversal is continued. Note that in this case the selected state s is not removed from *CurPath*.

```
function DepthFirstSearch(s_0 : Vertex) : seq of Vertex;
(* pre: true *)
begin var CurPath : seq of Vertex, (* path from s_0 to current vertex *)
          DfsPath : seq of Vertex;      (* visited vertices in dfs order *)
      CurPath, DfsPath := ⟨s_0⟩, ⟨s_0⟩;
      while CurPath ≠ ⟨⟩
      do s := head(CurPath);
         if Succ(s) ⊆ DfsPath then CurPath := tail(CurPath);
         else let s' in Succ(s) − DfsPath;
              CurPath, DfsPath := ⟨s'⟩⌢CurPath, ⟨s'⟩⌢DfsPath
         fi;
      od;
return DfsPath;
(* post: (1) s < s' (in DfsPath) if and only if s' is visited before s, and
         (2) sequence DfsPath contains all states reachable from s_0 *)
end
```

Table 4.2: Depth-first search traversal of state space

4.2.4 Checking for Emptiness

A fundamental issue in finite-state automata theory is to decide for a given FSA A whether it is *empty*, i.e., whether $\mathcal{L}(A) = \varnothing$? This is known as the *emptiness problem*. From the acceptance condition, it follows directly that A is non-empty if there is at least one run that ends in some accept state. FSA A is non-empty if and only if it has an accept state which is reachable from some initial state. Let us be more precise about the concept of reachability: state s' is *reachable* from s if there is a sequence $s_0 \ldots s_k$ such that $s_0 = s$ and $s_k = s'$ and $s_i \xrightarrow{a_i} s_{i+1}$ for $0 \leqslant i < k$ and input symbol a_i. Stated differently, s' is reachable from s if $s' \in \mathit{Succ}^*(s)$, where Succ^* denotes the reflexive and transitive closure of Succ.

The emptiness problem thus reduces to a graph reachability problem. Algorithmically this means that in order to decide whether FSA A is empty, we compute the set of states that are reachable from some initial state (for instance by means of a depth-first search algorithm), and check that no accept state (in F) occurs in this set.

Theorem 4.3.
For FSA $A = (\Sigma, S, I, \longrightarrow, F)$, $\mathcal{L}(A) \neq \varnothing$ if and only if there exists $s \in I$ and $s' \in F$ such that s' is reachable from s.

4.3 Automata on Infinite Words

Finite-state automata accept finite words, i.e., sequences of symbols of finite length. This section considers a variant of FSA that are suitable to accept words of infinite length.

4.3.1 ω-Regular Languages

Let Σ be an alphabet, i.e., a finite set of symbols (as before). The operations concatenation and (finite) repetition on words are now extended by *infinite repetition*. The infinite repetition of the finite word ab, for instance, equals $ababababab\ldots$ (ad infinitum). It is constructed from concatenating the word ab infinitely many times. We denote this by $(ab)^\omega$ (where ω denotes the first infinite ordinal). Note that the finite repetition of a word results in a language, i.e., a set of words, whereas infinite repetition results in a word. Sets of ω-words are called ω-languages. Infinite repetition can be lifted to languages as follows. For language $\mathcal{L}$ we let $\mathcal{L}^\omega = \{\, \sigma^\omega \mid \sigma \in \mathcal{L} \,\}$, i.e., the point-wise extension of $^\omega$ to sets of words. The result is thus an ω-language. For ω-word σ, $\mathit{inf}(\sigma)$ denotes the set of symbols that occurs infinitely often in σ. For instance, for $\sigma = bbba^\omega$ we have $\mathit{inf}(\sigma) = \{\, a \,\}$ and for $\sigma = bc(ac)^\omega$ we have $\mathit{inf}(\sigma) = \{\, a, c \,\}$. Note that for any infinite word σ we have $\mathit{inf}(\sigma) \neq \varnothing$: some input symbol has to occur infinitely often in σ, since the alphabet from which σ is constructed contains only finitely many symbols.

An important class of ω-languages is the set of *regular* ω-languages. A regular ω-language $\mathcal{L}$ is expressed as the finite union of $\mathcal{L}_i.(\mathcal{L}'_i)^\omega$ where $\mathcal{L}_i$ and $\mathcal{L}'_i$ are regular languages (for all i). Formally, $\mathcal{L} = \bigcup_{i=0}^{n} \mathcal{L}_i.(\mathcal{L}'_i)^\omega$. For instance, $\{\, aab^* \,\}.\{\, baa \,\}^\omega$ is a regular ω-language. Like regular languages, ω-regular languages are described by ω-regular expressions.

ω-regular languages possess a number of interesting properties. For instance, if $\mathcal{L} \subseteq \Sigma^*$ is regular then $\mathcal{L}^\omega$ is ω-regular. If in addition, $\mathcal{L}' \subseteq \Sigma^\omega$ is ω-regular, then the concatenation $\mathcal{L}.\mathcal{L}'$ is ω-regular. Furthermore, ω-regular languages possess several closure properties: they are closed under union, intersection and complementation. (The proof of the latter fact is non-trivial.) Further on in this chapter we will deal with the intersection of ω-regular languages and its automata-counterpart. Complementation will be dealt with briefly in Chapter 6 as its complexity (on Büchi automata) forms an essential motivation for the model-checking procedure for PLTL.

4.3.2 Büchi Automata

Automata play a special role in model-checking PLTL. Since we are interested in proving properties of infinite behavior, accepting runs are not considered to be finite, but rather as infinite, while cycling infinitely many times through accept states. The fact that for model checking we consider infinite behavior should not surprise the reader as reactive systems typically do not terminate. Automata with this alternative characterization of accepting run are called *Büchi automata* or ω-automata. A Büchi automaton (BA, for short) is an FSA that accepts infinite words rather than finite words. Note that the automaton itself is still *finite*: a BA contains only a finite number of states and finitely many transitions. An BA thus has the same components as an FSA, and only has a different acceptance condition, the so-called Büchi acceptance condition. What does it mean precisely to accept infinite words according to Büchi? This is defined as follows:

Definition 4.6. (**Run and word of an BA**)
A *run* of BA $A = (\Sigma, S, I, \longrightarrow, F)$ is an infinite sequence of states $\sigma = s_0\, s_1\, \ldots$ such that $s_0 \in I$ and $s_i \xrightarrow{a_i} s_{i+1}$ for all $0 \leqslant i$ for some $a_i \in \Sigma$. Run σ is *accepting* if and only if $\mathit{inf}(\sigma) \cap F \neq \varnothing$.

The infinite word $w = a_0\, a_1\, \ldots \in \Sigma^\omega$ is *accepted* by A if and only if there exists an accepting run $\sigma = s_0\, s_1\, \ldots$ such that $s_i \xrightarrow{a_i} s_{i+1}$ for $0 \leqslant i$.

The ω-language accepted by A, denoted $\mathcal{L}_\omega(A)$, is the set of infinite words accepted by A, i.e., $\mathcal{L}_\omega(A) = \{\, w \in \Sigma^\omega \mid w \text{ is accepted by } A \,\}$.

Since a run of a BA is infinite we cannot define acceptance by the fact that the final state of a run is an accept state or not. Such final state does not exist. According to Büchi's acceptance criterion, a run is accepting if some accept state is visited infinitely often. Notice that there always exists some state that is visited by a run infinitely often as we have finitely many states. If $F = \varnothing$ there are no accept states, no accepting runs, and thus $\mathcal{L}_\omega(A) = \varnothing$ in this case.

Example 4.6. *Consider the BA of Figure 4.6. Note that this automaton*

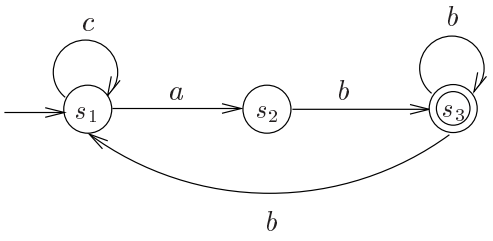


Figure 4.6: An example Büchi automaton

is equal to the FSA depicted in Figure 4.1. An example run of this BA is $s_1\,s_1\,s_1\,s_1\,\ldots$, *or shortly,* s_1^ω. *Some other runs are* $(s_1\,s_2\,s_3)^*\,s_1^\omega$, $s_1\,s_2\,s_3^\omega$ *and* $(s_1\,s_1\,s_2\,s_3)^\omega$. *The runs that go infinitely often through the accept state* s_3 *are accepting. For instance,* $s_1\,s_2\,s_3^\omega$ *and* $(s_1\,s_1\,s_2\,s_3)^\omega$ *are accepting runs.* s_1^ω *is not an accepting run as it never visits the accept state, while* $(s_1\,s_2\,s_3)^*\,s_1^\omega$ *is not accepting as it visits the accept state only finitely many times. The accepting words that correspond to these accepting runs are* ab^ω *and* $(ccab)^\omega$, *respectively. The language accepted by this BA is* $c^*a\,(b\,(c^*a \mid \varepsilon))^\omega$. *So, the BA accepts those infinite words that after starting with* c^*a, *have an infinite number of* b*'s such that in between any two successive* b*'s the sequence* c^*a *might appear but this does not need to be.* *(End of example.)*

The infinitary language accepted by this example BA is ω-regular. Like the strong relationship between FSA and regular languages, there is a relationship between BAs and ω-regular languages, as the following result by Büchi shows:

Theorem 4.4.
Language $\mathcal{L}$ *is* ω*-regular if and only if there exists a BA* A *such that* $\mathcal{L} = \mathcal{L}_\omega(A)$.

Thus, for every ω-regular expression (which describes an ω-regular language) there exists a Büchi automaton that only accepts this ω-regular expression and nothing else.

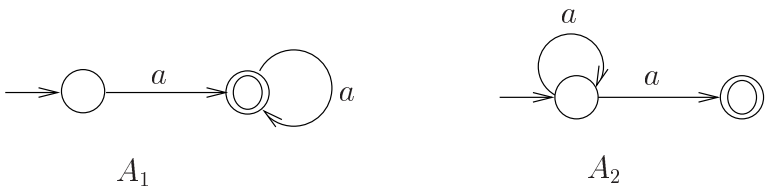
Two Büchi automata are ω-equivalent if they accept the same ω-language:

Definition 4.7. (**Equivalence of Büchi automaton**)
BA A and A' are ω*-equivalent*, denoted $A \equiv_\omega A'$, if and only if $\mathcal{L}_\omega(A) = \mathcal{L}_\omega(A')$.

Before continuing with Büchi automata, it is important to realize their subtle differences with finite-state automata. For instance, it is interesting to consider more carefully the relationship between $\mathcal{L}(A)$, the set of finite words accepted by the automaton A, and $\mathcal{L}_\omega(A)$, the set of infinite words accepted by A according to the Büchi acceptance condition. This is done in the following example.

Example 4.7. In this example we consider some differences between FSA and BA. Let A_1 *and* A_2 *be two automata. Let* $\mathcal{L}(A_i)$ *denote the language accepted by* A_i $(i = 1, 2)$, *and* $\mathcal{L}_\omega(A_i)$ *its* ω*-language.*

1. *If* A_1 *and* A_2 *accept the same finite words, then this does not mean that they also accept the same infinite words. The following two example automata show this:*



A_1 A_2

We have $\mathcal{L}(A_1) = \mathcal{L}(A_2) = \{\, a^n \mid n \geqslant 1 \,\}$, *but* $\mathcal{L}_\omega(A_1) = \{\, a^\omega \,\}$ *and* $\mathcal{L}_\omega(A_2) = \varnothing$. *Thus,* $A_1 \equiv A_2$ *but* $A_1 \not\equiv_\omega A_2$.

2. *If* A_1 *and* A_2 *accept the same infinite words, then one might expect that they would also accept the same finite words. This also turns out not to be true. The following example shows this:*

A_1 A_2

We have $\mathcal{L}_\omega(A_1) = \mathcal{L}_\omega(A_2) = \{\, a^\omega \,\}$, *but* $\mathcal{L}(A_1) = \{\, a^{2n} \mid n \geqslant 0 \,\}$ *and* $\mathcal{L}(A_2) = \{\, a^{2n+1} \mid n \geqslant 0 \,\}$. *Thus,* $A_1 \equiv_\omega A_2$ *but* $A_1 \not\equiv A_2$.

3. *If* A_1 *and* A_2 *are both deterministic, then* $A_1 \equiv A_2 \Rightarrow A_1 \equiv_\omega A_2$. *The reverse is, however, not true, as illustrated by the previous example.*

(End of example.)

4.3.3 Deterministic Büchi Automata

An important difference between finite-state automata and Büchi automata is the expressive power of deterministic and non-deterministic automata. As we have seen before, non-deterministic FSA are as expressive as deterministic FSA. However, *non-deterministic BA are more expressive than deterministic BA*. That is, there do exist non-deterministic BA for which there does not exist an equivalent deterministic BA. Stated this differently, we have that any ω-language accepted by a deterministic BA is ω-regular but the reverse does hot hold: there do exist ω-regular languages for which there does not exist a deterministic BA accepting it. An example of such ω-regular language is $(a \mid b)^* b^\omega$.

We could, for instance, try to transform the non-deterministic BA that accepts this language – by Theorem 4.4 such automaton should exist – into an equivalent deterministic BA using the subset construction for FSA. A non-deterministic

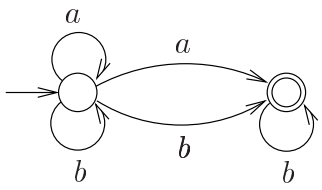


Figure 4.7: A non-deterministic BA for which there does not exist an ω-equivalent deterministic BA

BA A that accepts $(a \mid b)^* b^\omega$ is depicted in Figure 4.7. Note that in the initial state on input symbols a and b there is either the possibility to move to the accept state, or to remain in the current state. Naively applying the subset construction algorithm to this BA yields the BA, A' say, depicted in Figure 4.8. Although A' is deterministic, A and A' are not ω-equivalent. For instance, the infinite word $(ab)^\omega$ is accepted by A', but not by A. To be more precise, we have that $\mathcal{L}_\omega(A') = (a \mid b)^\omega$, a superset of $\mathcal{L}_\omega(A)$. The reason for their difference is that A' is always able to recognize a-symbols. Instead, the BA A allows to decide after an arbitrary (finite) number of as or bs to move to the accept state, and start to not recognize any a-symbols any further.

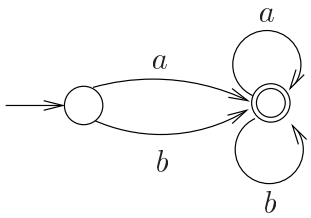


Figure 4.8: Deterministic BA obtained by applying the subset construction algorithm to Figure 4.7

It should not surprise the reader that the subset construction does not work as non-deterministic and deterministic BA differ in expressiveness. The set of languages accepted by a deterministic BA is just a subset of the set of ω-regular languages. The fact that there does not exist a deterministic BA that accepts $(a \mid b)^* b^\omega$ can intuitively be seen as follows. The reasoning is by contradiction. Suppose that such deterministic BA does exist and assume that the finite sequence $(a \mid b)^n$ for some arbitrary natural n has been recognised and that the next input symbol is b. As the BA is deterministic there are just two possibilities. It may either decide that after this b no as will follow anymore, and move to a corresponding (accept) state in which it will not be able to recognize any a anymore. This is, however, not appropriate, as this automaton will not be able to recognise the input word $(a \mid b)^n ba \ldots$. Alternatively, the BA could decide to still be able to allow as to follow. But then – as the BA is deterministic – the same argument can be repeated for $(a \mid b)^{n+1}$, $(a \mid b)^{n+2}$, and so on. The resulting automaton, however, would then not be able to accept, for instance, the infinite word b^ω.

Basically, in order for the BA to decide whether after a b no as will follow, it has to look arbitrarily far ahead in the input word. As this is not possible, there is only one way to get around this, and this is by using non-determinism. By changing the acceptance condition of Büchi, a class of deterministic automata on infinite words can be defined that characterizes the ω-regular languages. Two kinds of such automata, Muller and Rabin automata, are some of the variants of BA covered below.

Example 4.8. (This illuminating example is taken from [180].) Consider

Figure 4.9: The BA (a) A_1 and (b) A_2 with $A_1 \not\equiv_\omega A_2$

BA A_1 depicted in Figure 4.9(a). As there is no run that can visit its accept state infinitely often, it is clear that this BA does not accept any word, i.e., $\mathcal{L}_\omega(A_1) = \varnothing$. On the contrary, BA A_2 (cf. Figure 4.9(b)) accepts the infinite word a^ω, i.e., $\mathcal{L}_\omega(A_2) = \{ a^\omega \}$. Thus, A_1 and A_2 are not ω-equivalent. If we apply the subset construction algorithm to both automata we obtain in both cases the same BA! Thus, the subset construction algorithm is not able to distinguish between the two inequivalent BA A_1 and A_2. (End of example.)

4.3.4 Other Automata on Infinite Words

The Büchi acceptance condition on infinite runs is one out of several possibilities to define when a run is ought to be accepting. Although Büchi automata are instrumental for model-checking PLTL, we briefly mention here some of the other acceptance conditions for automata on infinite words. Due to the various ways in which the acceptance of infinite words can be defined, different variants of automata over infinite words exist. These automata are christened according to the scientist that proposed the acceptance criterion: Rabin and Muller. The acceptance characteristic of the most prominent types of automata over infinite words are listed in Table 4.3. In the sequel of this section we briefly discuss these automata. As generalised Büchi automata will play a central role in model-checking PLTL we treat these automata more extensively.

Automaton	Accept sets	Accept condition
Büchi	$F \subseteq S$	$\mathit{inf}(\sigma) \cap F \neq \varnothing$
Generalised Büchi	$\mathcal{F} = \{ F_1, \ldots, F_k \}$ where $F_i \subseteq S$	$\forall\, i.\ \mathit{inf}(\sigma) \cap F_i \neq \varnothing$
Muller	idem	$\exists\, i.\ \mathit{inf}(\sigma) \cap F_i = F_i$
Rabin	$\mathcal{F} = \{ (F_1, G_1), \ldots, (F_k, G_k) \}$ where $F_i \subseteq S$, $G_i \subseteq S$	$\exists\, i.\ \mathit{inf}(\sigma) \cap F_i = \varnothing$ $\wedge$ $\mathit{inf}(\sigma) \cap G_i \neq \varnothing$

Table 4.3: Major types of automata on infinite words

Generalised Büchi Automata

Definition 4.8. (**Generalised Büchi automaton**)
A *generalized Büchi automaton* (GBA) A is a tuple $(\Sigma, S, I, \longrightarrow, \mathcal{F})$ where the first four components are the same as for a BA and $\mathcal{F} \subseteq 2^S$.

A GBA is like a BA except that $\mathcal{F}$ is a *set* of accept sets $\{ F_1, \ldots, F_k \}$ for $k \geqslant 0$ with $F_i \subseteq S$ rather than a single set of accept states. Run $\sigma = s_0\, s_1 \ldots$ is accepting for GLBA A if and only if:

$$\mathit{inf}(\sigma) \cap F_i \neq \varnothing \text{ for all } 0 < i \leqslant k.$$

Thus, in an accepting run should go infinitely often through at least one state of each acceptance set $F_i \in \mathcal{F}$. Note that if $\mathcal{F} = \varnothing$ all runs go infinitely often through all accept sets in $\mathcal{F}$, so any run is accepting in that case.

Example 4.9. Consider the GBA depicted in Figure 4.10. We illustrate the relevance of the choice of $\mathcal{F}$ by discussing two alternatives. First, let $\mathcal{F}$ consist of a single set $\{ s_1, s_2 \}$, i.e. there is a single acceptance set containing s_1 and s_2. An accepting run has to go infinitely often through some state in $\mathcal{F}$. Since s_2 cannot be visited infinitely often, the only candidate to be visited infinitely often by an accepting run is state s_1. Thus, the language accepted by this automaton equals $a\,(b\,(a \mid \varepsilon))^\omega$.

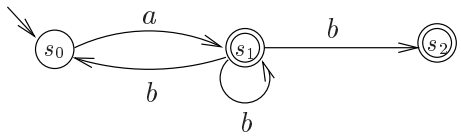


Figure 4.10: An example generalized Büchi automaton

If we now let $\mathcal{F}$ consist of $\{ s_1 \}$ and $\{ s_2 \}$, i.e., two accepting sets consisting of s_1 and s_2, respectively, the GBA has no accepting run, and thus its language is $\varnothing$. Since any accepting run has to go infinitely often through each acceptance

set, such run has to visit s_2 infinitely often, which is impossible. (End of example.)

A generalized BA can be transformed into an ω-equivalent BA. To transform a generalized BA A with k acceptance sets into a BA A', k copies of A are made, one copy per accept set. Each state s in the BA thus becomes a pair (s, i) with $0 < i \leqslant k$. Automaton A' can start in some initial state (s_0, i) where s_0 is an initial state of A. In each copy the transitions are as in A, with the only – but essential – exception that when an accept state in F_i is reached in the i-th copy, then the automaton switches to the $(i{+}1)$-th copy.

Definition 4.9. (**From a GBA to an BA**)
For GBA $A = (\Sigma, S, I, \longrightarrow, \mathcal{F})$ with $\mathcal{F} = \{ F_1, \ldots, F_k \}$, the BA $gba2ba(A) = (\Sigma, S', I', \longrightarrow', F')$ is defined as follows:

- $S' = S \times \{ i \mid 0 < i \leqslant k \}$
- $I' = I \times \{ i \}$ for some $0 < i \leqslant k$
- $\longrightarrow'$ is the smallest relation defined by:
  - for $s \notin F_i$: $(s, i) \xrightarrow{a}' (s', i)$ if and only if $s \xrightarrow{a} s'$
  - for $s \in F_i$: $(s, i) \xrightarrow{a}' (s', (i{+}1) \textbf{ mod } k)$ if and only if $s \xrightarrow{a} s'$
- $F' = F_i \times \{ i \}$ for some $0 < i \leqslant k$.

Since for the definition of the initial and accept states of A' an arbitrary (and even different) i can be chosen, the automaton A' is not uniquely determined.

It follows that GBA A and BA $A' = gba2ba(A)$ are ω-equivalent. This can be seen as follows. For a run of A' to be accepting it has to visit some state (s, i) infinitely often, where s is an accept state in F_i in the GBA A. As soon as a run reaches this state (s, i), the BA A' moves to the $(i{+}1)$-th copy. From the $(i{+}1)$-th copy the next copy can be reached by visiting $(s', i{+}1)$ with $s' \in F_{i+1}$. A' can only return to (s, i) if it goes through all k copies. This is only possible if it reaches an accept state in each copy since that is the only opportunity to move to the next copy. So, for a run to visit (s, i) infinitely often it has to visit some accept state in each copy infinitely often. Given this concept it is not hard to see that $A \equiv_\omega gba2ba(A)$.

Example 4.10. Consider the generalized BA in Figure 4.11(a). It has two acceptance sets $F_1 = \{ s_1 \}$ and $F_2 = \{ s_2 \}$. According to Definition 4.9 the states

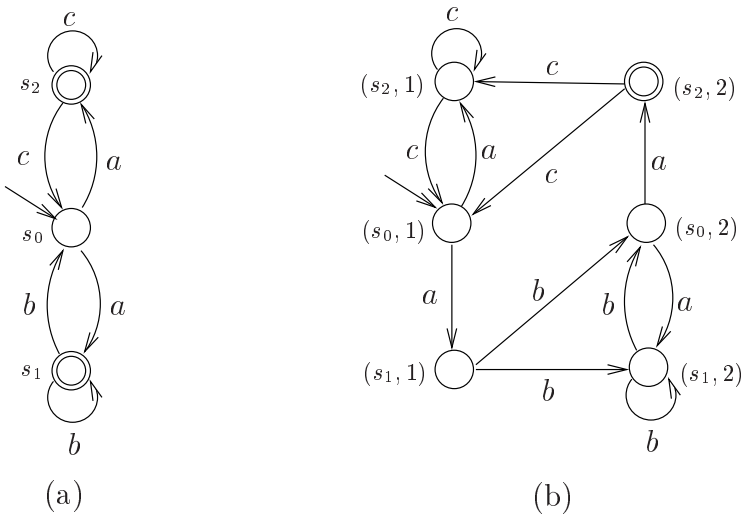


Figure 4.11: A generalized BA with and (one of) its corresponding BA

of the corresponding BA are $\{ s_0, s_1, s_2 \} \times \{ 1, 2 \}$. *Some example transitions in the corresponding BA are:*

$$
\begin{array}{lll}
(s_0,1) \xrightarrow{a} (s_1,1) & \textit{since} & s_0 \xrightarrow{a} s_1 \textit{ and } s_0 \notin F_1 \\
(s_0,1) \xrightarrow{a} (s_2,1) & \textit{since} & s_0 \xrightarrow{a} s_2 \textit{ and } s_0 \notin F_1 \\
(s_1,1) \xrightarrow{b} (s_0,2) & \textit{since} & s_1 \xrightarrow{b} s_0 \textit{ and } s_1 \in F_1 \\
(s_1,2) \xrightarrow{b} (s_1,2) & \textit{since} & s_1 \xrightarrow{b} s_1 \textit{ and } s_1 \notin F_2 \\
(s_2,2) \xrightarrow{c} (s_2,1) & \textit{since} & s_2 \xrightarrow{c} s_2 \textit{ and } s_2 \in F_2
\end{array}
$$

The justification of the other transitions is left to the reader. A possible resulting BA is depicted in Figure 4.11(b) where the set of initial states is chosen to be $\{ (s_0, 1) \}$ *and the set of accept states is chosen to be* $\{ (s_2, 2) \}$. *Any accepting run of the BA must visit* $(s_2, 2)$ *infinitely often* ($s_2 \in F_2$). *In order to do so it also has to visit a state labelled with* $s_1 \in F_1$ *infinitely often. Thus, an accepting run of the resulting BA visits some state of* F_1 *and some state of* F_2 *infinitely often.* *(End of example.)*

Muller Automata

A *Muller automaton* is defined like a generalised Büchi automaton except that it accepts a run if and only if it goes infinitely often through *all* states of $F_i \in \mathcal{F}$, for some i. Muller automata and BA have the same expressive power: they both characterize ω-regular languages. An interesting property is that deterministic Muller automata characterise ω-regular languages, i.e., an ω-language is regular if and only if it is recognisable by a deterministic Muller automaton. Recall that deterministic BA do not have this expressive power as there are ω-regular languages that cannot be recognized by any deterministic BA. The deterministic

BA $(\Sigma, S, I, \longrightarrow, F)$ is ω-equivalent to the Muller automaton $(\Sigma, S, I, \longrightarrow, \mathcal{F})$ where $\mathcal{F}$ contains all subsets $R \subseteq S$ with $R \cap F \neq \varnothing$. The complement of a Muller automaton $(\Sigma, S, I, \longrightarrow, \mathcal{F})$ is the Muller automaton $(\Sigma, S, I, \longrightarrow, 2^S - \mathcal{F})$. Unlike Büchi automata, complementation of Muller automata is thus rather straightforward.

Rabin Automata

A *Rabin automaton* has a set of pairs of sets of states as acceptance sets: it consists of Σ, S, I, and $\longrightarrow$ (as before) together with a set $(F_1, G_1), \ldots, (F_k, G_k)$ for some natural k. A run is accepted if for some i $(0 < i \leqslant k)$ we have that states in F_i are visited only finitely often, while there should be some state in G_i that is visited infinitely often. Rabin automata have the same expressiveness properties as Muller automata: any ω-regular language can be recognized by either a (possibly non-deterministic) Rabin automaton or by a deterministic one. For any non-deterministic BA with n states there is an ω-equivalent Rabin automaton with $2^{\mathcal{O}(n \log n)}$ states and n accepting pairs (F_i, G_i).

Example 4.11. Recall that there is no deterministic BA that accepts the language $(a \mid b)^* b^\omega$. However, the deterministic Rabin automaton in Figure 4.12 with the accept set consisting of the single pair $(\{ s_a \}, \{ s_b \})$ accepts this language. The Rabin automaton starts in state s_a and then remembers the last symbol it has read, where in state s_a the symbol a and in state s_b symbol b has just been read. As any accepting run of this automaton – according to Rabin's criterion – goes finitely often through s_a and infinitely often through s_b it follows immediately that its language equals $(a \mid b)^* b^\omega$. (End of example.)

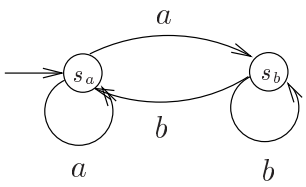


Figure 4.12: A deterministic Rabin automaton that accepts $(a \mid b)^* b^\omega$

4.4 Algorithms for Büchi Automata

This section presents several algorithms for Büchi automata that are relevant for model-checking PLTL. Algorithms to determine the synchronous product of two BA, to transform a GBA into an equivalent BA, and to check for emptiness are covered.

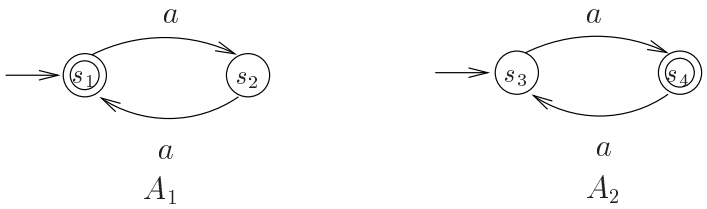
4.4.1 Synchronous Product

ω-regular languages are closed under intersection, i.e., if $\mathcal{L}$ and $\mathcal{L}'$ are ω-regular, then so is $\mathcal{L} \cap \mathcal{L}'$. The corresponding operation on BA is synchronous product. For Büchi automata the procedure for finite-state automata (cf. Definition 4.5) is, however, not appropriate. In the product construction for FSA, the set of accept states equals the product of acceptance sets F_1 and F_2. For BA A_1 and A_2 this means that $A_1 \times A_2$ accepts an infinite word w if there are accepting runs of A_1 and A_2 on w, where both runs go infinitely often and *simultaneously* through accept states. This requirement is too strong and leads in general to

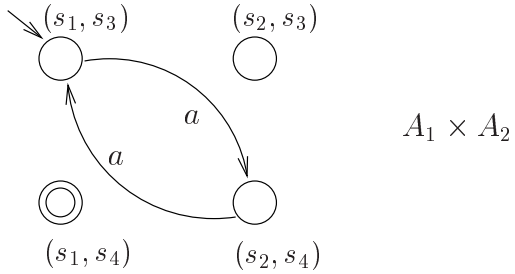
$$\mathcal{L}_\omega(A_1 \times A_2) \subseteq \mathcal{L}_\omega(A_1) \cap \mathcal{L}_\omega(A_2)$$

which is not the desired result. This is illustrated by the following example.

Example 4.12. Consider the two Büchi automata



The language accepted by these BA is: $\mathcal{L}_\omega(A_1) = \mathcal{L}_\omega(A_2) = \{\, a^\omega \,\}$. *Following the construction of Definition 4.5 we obtain the BA* $A_1 \times A_2$*:*



which has no accepting run. So, $\mathcal{L}_\omega(A_1 \times A_2) = \varnothing \neq \mathcal{L}_\omega(A_1) \cap \mathcal{L}_\omega(A_2)$. *The point is that the product automaton assumes that* A_1 *and* A_2 *go simultaneously through an accept state, which is never the case in this example since* A_1 *and* A_2 *are "out of phase" from the beginning on: if* A_1 *is in an accept state,* A_2 *is not, and vice versa.*

Notice that when considered as automata on finite words, then $\mathcal{L}(A_1) = \{\, a^{2n} \mid n \geqslant 0 \,\}$ *and* $\mathcal{L}(A_2) = \{\, a^{2n+1} \mid n \geqslant 0 \,\}$ *and* $\mathcal{L}(A_1 \times A_2) = \varnothing = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$. *(End of example.)*

There is a modification of the product automaton construction on Büchi automata that corresponds to intersection of ω-regular languages.

Definition 4.10. (**Synchronous product of Büchi automata**)
Let $A_i = (\Sigma, S_i, I_i, \longrightarrow_i, F_i)$ for i=1, 2 be two BA. The *product automaton* $A_1 \otimes A_2 = (\Sigma, S, I, \longrightarrow, F)$ is defined as follows:

- $S = S_1 \times S_2 \times \{ 1, 2 \}$
- $I = I_1 \times I_2 \times \{ 1 \}$
- if $s_1 \xrightarrow{a}_1 s_1'$ and $s_2 \xrightarrow{a}_2 s_2'$ then
  - (i) if $s_1 \in F_1$ then $(s_1, s_2, 1) \xrightarrow{a} (s_1', s_2', 2)$
  - (ii) if $s_2 \in F_2$ then $(s_1, s_2, 2) \xrightarrow{a} (s_1', s_2', 1)$
  - (iii) in all other cases $(s_1, s_2, i) \xrightarrow{a} (s_1', s_2', i)$ for i=1, 2
- $F = F_1 \times S_2 \times \{ 1 \}$

The intuition behind this construction is as follows. The automaton $A = A_1 \otimes A_2$ runs both A_1 and A_2 on the input word. Thus the automaton can be considered to have two "tracks", one for A_1 and one for A_2. In addition to remembering the state of each track (the first two components of a state), A also has a pointer that points to one of the tracks (the third component of a state). Whenever a track goes through an accept state, the pointer moves to another track (rules (i) and (ii)). More precisely, if it goes through an accept state of A_i while pointing to track i, it changes to track (i+1) modulo 2.

The acceptance condition guarantees that both tracks visit accept states infinitely often, since a run is accepted if and only if it goes through $F_1 \times S_2 \times \{ 1 \}$ infinitely often. This means that the first track visits infinitely often an accept state with the pointer pointing to the first track. Whenever the first track visits an accept state (of A_1) with the pointer pointing to the first track, the track is changed (i.e., the pointer is moved to the other track). The pointer only returns to the first track if the second track visits an accept state (of A_2). Thus in order to visit an accept state of $A_1 \otimes A_2$, both A_1 and A_2 have to visit an accept state infinitely often and we have $\mathcal{L}_\omega(A_1 \otimes A_2) = \mathcal{L}_\omega(A_1) \cap \mathcal{L}_\omega(A_2)$.

Example 4.13. Consider the Büchi automata A_1 and A_2 from the previous example. The BA $A_1 \otimes A_2$ is constructed as follows. The set of states equals

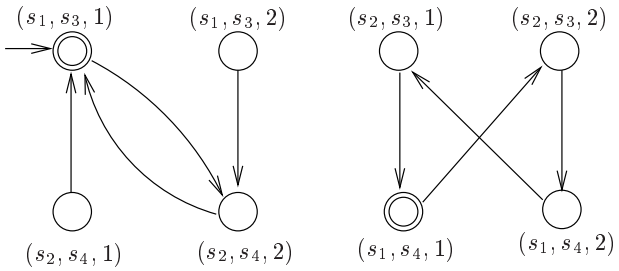
$$\{ s_1, s_2 \} \times \{ s_3, s_4 \} \times \{ 1, 2 \}$$

which yields $2^3 = 8$ states. The initial state is $(s_1, s_3, 1)$. The accept states are $(s_1, s_3, 1)$ and $(s_1, s_4, 1)$. The following example transitions can be derived from

the above definition:

$$
\begin{array}{lll}
(s_1, s_3, 1) \xrightarrow{a} (s_2, s_4, 2) & \textit{since} & s_1 \xrightarrow{a}_1 s_2 \textit{ and } s_3 \xrightarrow{a}_2 s_4 \textit{ and } s_1 \in F_1 \\
(s_1, s_4, 2) \xrightarrow{a} (s_2, s_3, 1) & \textit{since} & s_1 \xrightarrow{a}_1 s_2 \textit{ and } s_4 \xrightarrow{a}_2 s_3 \textit{ and } s_4 \in F_2 \\
(s_1, s_3, 2) \xrightarrow{a} (s_2, s_4, 2) & \textit{since} & s_1 \xrightarrow{a}_1 s_2 \textit{ and } s_3 \xrightarrow{a}_2 s_4 \textit{ and } s_3 \notin F_2
\end{array}
$$

The first transition follows from rule (i), the second from rule (ii), and the third rule follows from rule (iii). The justification of the other transitions is left to the interested reader. The resulting product automaton $A_1 \otimes A_2$ is now:



where all transitions are labeled with a. Clearly, the ω-language accepted by this product automaton is a^ω which equals $\mathcal{L}_\omega(A_1) \cap \mathcal{L}_\omega(A_2)$. (End of example.)

4.4.2 Checking for Emptiness

The second problem on BA that we consider here is: given a BA A how does one determine whether A is *empty*, i.e., whether $\mathcal{L}_\omega(A) = \varnothing$? This is known as the *emptiness problem*. From the acceptance condition, it follows directly that A is non-empty if there is at least one run that goes infinitely often through some accept state. A BA A is thus non-empty if and only if it has an accept state which is (i) reachable from some initial state and (ii) reachable from itself (in one or more steps). Stated in graph-theoretic terms it means that A contains a cycle reachable from an initial state such that the cycle contains some accept state. Recall that s' is reachable from s if there is a sequence $s_0 \ldots s_k$ such that $s_0 = s$ and $s_k = s'$ and $s_i \xrightarrow{a_i} s_{i+1}$ for $0 \leqslant i < k$ and input symbol a_i.

Theorem 4.5.
Let $A = (\Sigma, S, I, \longrightarrow, F)$ be a BA. $\mathcal{L}_\omega(A) \neq \varnothing$ if and only if there exists $s_0 \in I$ and $s' \in F$ such that s' is reachable from s_0 and s' is reachable from s'.

This result can be explained as follows. If A is non-empty it is not difficult to see that there must be a reachable cycle that contains an accept state. In the other direction the argument is slightly more involved. Suppose there are states $s_0 \in I$ and $s' \in F$ such that s' is reachable from s_0 and s' is reachable from itself. Since s' is reachable from s_0 there is a sequence of states $s_0\, s_1 \ldots s_k$, $k \geqslant 0$,

```
function ReachAccept (s_0 : Vertex) : seq of Vertex;
(* pre: true *)
begin var CurPath : seq of Vertex,          (* path to current vertex *)
          AccReach : seq of Vertex,     (* reachable accepting states *)
          Visit : set of Vertex;                (* visited vertices *)
      CurPath, AccReach, Visit := ⟨s_0⟩, ⟨⟩, ∅;
      while CurPath ≠ ⟨⟩
      do s := head(CurPath);
         if Succ(s) ⊆ Visit
         then CurPath := tail(CurPath);
              if s ∈ F then AccReach := AccReach⌢⟨s⟩ fi
         else let s' in Succ(s) − Visit;
              CurPath, Visit := ⟨s'⟩⌢CurPath, Visit ∪ { s' }
         fi;
      od;
return AccReach;
(* post: AccReach contains accept states reachable from s_0 in dfs-order *)
end
```

Table 4.4: Algorithm for determining reachable accept states

and a sequence of symbols $a_0 \ldots a_{k-1}$ such that $s_k = s'$ (i.e., the sequence ends in s'), $s_i \xrightarrow{a_i} s_{i+1}$ $(0 \leqslant i < k)$ for all i. Similarly, since s' is reachable from itself there is a sequence $s'_0\, s'_1 \ldots s'_n$ and a sequence of symbols $b_0 \ldots b_{n-1}$ such that $s'_0 = s'_n = s'$, $n > 0$, and $s'_i \xrightarrow{b_i} s'_{i+1}$ $(0 \leqslant i < n)$. But, since $s' \in F$ then $(s_0 \ldots s_{k-1})(s'_0 \ldots s'_n)^\omega$ is an accepting run of A on the input word $(a_0 \ldots a_{k-1})(b_0 \ldots b_n)^\omega$. Thus $\mathcal{L}_\omega(A)$ contains at least one accepting word, and hence, is non-empty.

So, *to determine whether a given A is non-empty, it suffices to check whether A has a reachable cycle that contains an accept state.* Such a reachable cycle is also called a – non-trivial, since it must contain at least one edge – maximal strongly connected component of A. The algorithm that checks whether A has a reachable cycle that contains an accept state consists of two steps. For convenience, assume that A has a single initial state s_0, i.e., $I = \{ s_0 \}$.

1. In the first step all accept states that are reachable from the initial state are determined. For convenience, the accept states are ordered. This is performed by the function *ReachAccept* which is listed in Table 4.4.
2. In the second step it is checked whether an accept state computed in the first step belongs to some cycle. This is performed by the function *DetectCycle* which is listed in Table 4.5.

The main program now consists of **return** *DetectCycle*(*ReachAccept*(s_0)).

In a nutshell, the algorithm *ReachAccept* works as follows. The graph representing the Büchi automaton is traversed in a depth-first search order. Starting

```
function DetectCycle(AccReach : seq of Vertex) : Bool;
(* pre: true *)
begin var CurPath, Accept : seq of Vertex,
          Visit : set of Vertex,
          CycleFound : Bool;
      CurPath, Accept, Visit, CycleFound := ⟨⟩, AccReach, ∅, false;
      while Accept ≠ ⟨⟩ ∧ ¬ CycleFound
      do s := head(Accept);
         Accept, CurPath := tail(Accept), ⟨s⟩⌢ CurPath;
         while CurPath ≠ ⟨⟩ ∧ ¬ CycleFound
         do s' := head(CurPath);
            CycleFound := (s ∈ Succ(s'));
            if Succ(s') ⊆ Visit then CurPath := tail(CurPath)
            else let s'' in Succ(s') − Visit;
                 CurPath, Visit := ⟨s''⟩⌢ CurPath, Visit ∪ { s'' }
            fi;
         od;
      od;
return CycleFound;
(* post: CycleFound ≡ some state in AccReach is member of a cycle *)
end
```

Table 4.5: Algorithm for determining the existence of an accept cycle

from the initial state s_0, in each step a new state is selected (if such state exists) that is directly reachable from the currently explored state s. When all paths starting from s have been explored (i.e., if $Succ(s) \subseteq Visit$), s is removed from *CurPath*, and if s is accepting it is appended to *AccReach*, the sequence of accept states reachable from s_0. The algorithm terminates when all possible paths starting from s_0 have been explored, i.e., when all states reachable from s_0 have been visited. Notice that all operations on *CurPath* are at its front; it is therefore usually implemented as a stack.

The algorithm *DetectCycle* carries out a depth-first search starting from an accept state reachable from s_0. It terminates if all accept states in *AccReach* have been checked or if some accept state has been found that is on a cycle. Note that elements of *AccReach* are removed from its front, whereas they were inserted by function *ReachAccept* at the back; *AccReach* is usually implemented as a first-in first-out queue.

4.4.3 Nested Depth-First Search

Rather than computing *ReachAccept* and *DetectCycle* as two separate phases, it is possible to check whether an accept state is a member of a cycle on-the-fly, that is while determining all accept states reachable from s_0. We thus obtain a *nested depth-first search*: in the "outermost" search an accept state is identified which is reachable from s_0, whereas in the "innermost" search it is determined

```
function ReachAcceptandDetectCycle(s_0 : Vertex) : Bool;
(* pre: true *)
begin var PathToAcc, Cycle : seq of Vertex,
          Visit, Visit' : set of Vertex,
          AccCycleFound : Bool;
          PathToAcc, Visit, AccCycleFound := ⟨s_0⟩, ∅, false;
      while PathToAcc ≠ ⟨⟩ ∧ ¬ AccCycleFound
      do s := head(PathToAcc);
          if Succ(s) ⊆ Visit
          then PathToAcc := tail(PathToAcc);
                  if s ∈ F
                  then Cycle, Visit' := ⟨s⟩, ∅;
                          while Cycle ≠ ⟨⟩ ∧ ¬ AccCycleFound
                          do s' := head(Cycle);
                              AccCycleFound := (s ∈ Succ(s'));
                              if Succ(s') ⊆ Visit' then Cycle := tail(Cycle)
                              else let s'' in Succ(s') − Visit';
                                      Cycle, Visit' := ⟨s''⟩⌢Cycle, Visit' ∪ { s'' }
                              fi;
                          od;
                  fi;
          else let s'' in Succ(s) − Visit;
                  PathToAcc, Visit := ⟨s''⟩⌢PathToAcc, Visit ∪ { s'' }
          fi;
      od;
return AccCycleFound;
(* post: AccCycleFound ≡ an accept cycle is reachable from state s_0 *)
end
```

Table 4.6: Algorithm for checking the existence of an accept cycle

whether such a state is a member of a cycle. The resulting program is shown in Table 4.6.

Now assume we apply this algorithm to the automaton A. An interesting aspect of this program is that if a cycle is determined in that contains an accept state s, then a path to that state can be computed easily: sequence *PathToAcc* contains the path from the start state s_0 to s (in reversed order), while sequence *Cycle* contains the cycle from s to s (in reversed order). Reversing each sequence and concatenating them yields the desired path.

The time complexity of the nested depth-first search algorithm of Table 4.6 is proportional to the number of states plus the number of transitions in the BA under consideration, i.e., $\mathcal{O}(\mid S \mid + \mid \longrightarrow \mid)$.

4.5 Summary

- a run is accepted by a finite-state automaton (FSA) if it ends in an accept state.
- FSA accept regular languages, and for each regular language a FSA exists that accepts it.
- for any non-deterministic FSA there exists an equivalent deterministic FSA; the algorithm is the subset construction algorithm and may yield an exponential blow-up in the number of states.
- the emptiness problem for FSA is reducible to graph reachability.
- a run is accepted by a Büchi automaton if it goed through an accept state infinitely often.
- BA accept ω-regular languages, and for each ω-regular language a BA exists that accepts it.
- BA are different from FSA in many respects:
  - deterministic BA are less expressive than non-deterministic ones; there exist ω-regular languages for which no deterministic BA exists that accepts it.
  - neither language equivalence implies ω-equivalence, nor the reverse.
  - checking for emptiness amounts to nested graph reachability.
- a run is accepted by a generalized BA if it goes through all its sets of accept states infinitely often
- for each generalized BA there exists an ω-equivalent BA.
- Muller and Rabin automata are variants of BA for which deterministic and non-deterministic automata are equally expressive; complementation of these automata is rather straightforward (as opposed to BA).

4.6 Bibliographic Notes

Finite-state automata and regular languages. The equivalence of regular languages and accepted languages of finite-state automata was established by Kleene [109]. Various algorithms exist to construct an FSA starting from a regular language. The one by Thompson [177] can be found in most compiler books and produces a non-deterministic FSA; the earlier algorithm by McNaughton and Yamada [135] generates a deterministic FSA. For a given regular expression of length n, the number of states of its corresponding non-deterministic FSA is maximally $\mathcal{O}(n)$. This automaton decides whether a word of length k

is accepted in time $\mathcal{O}(n{\cdot}k)$. For a deterministic automaton these complexity figures are $\mathcal{O}(2^n)$ and $\mathcal{O}(k)$, respectively. The deterministic automaton is larger than its non-deterministic counterpart, but recognizes a given input word faster. Concerning the resulting number of transitions in the non-deterministic FSA, Hromkovič, Seibert and Wilke [100] have recently proven that there does not exist a linear-size conversion; their algorithm generates $\mathcal{O}(n{\cdot}(\log n)^2)$ transitions in $\mathcal{O}(n^2{\cdot}\log n)$ time.

Algorithms on finite-state automata. The subset construction algorithm that converts a non-deterministic automaton into an equivalent deterministic one originates by Rabin and Scott [156] from 1959. The closure properties for FSA (under union, intersection and complementation of regular languages) also originate from this seminal paper. Meyer and Fischer [139] showed that the exponential blow-up in the number of states under the subset construction method cannot be improved, i.e., determinization cannot be done without an exponential growth of the number of states. Depth-first search algorithms and various other graph algorithms have been studied by Tarjan [173]. The reduction of the emptiness problem for FSA to graph reachability is due to Rabin and Scott [156]. This result showed that emptiness is decidable in linear time. Jones [106] showed that graph reachability – and thus FSA emptiness – is NLOGSPACE complete. The related problem of checking nonuniversality of an FSA A, i.e., decide whether $\mathcal{L}(A) \neq \Sigma^*$, or equivalently, $\mathcal{L}(\overline{A}) \neq \varnothing$ has been studied by Meyer and Stockmeyer [140]. It has an exponential time complexity and is PSPACE-complete.

Büchi automata. The product construction for BA is due to Choueka [42]. For a survey of automata on infinite words we recommend the overview by Thomas [175]. An overview of model checking using BA (and variants thereof) has been given by Vardi [180]. The characterization of ω-regular languages by Muller automata is due to McNaughton [134]. The algorithm to transform a non-deterministic BA with n states into a deterministic Muller automaton with $2^{\mathcal{O}(n \log n)}$ states and n accept pairs of sets of states is due to Safra [160]. The decidability of the emptiness problem for BA is due to Emerson and Lei [72] who showed that it is reducible to graph reachability. Vardi and Wolper [182] showed that checking for emptiness is NLOGSPACE-complete. The nonuniversality problem for BA has been studied by Sistla, Vardi and Wolper [169]. It has an exponential time complexity and is PSPACE-complete. The nested depth-first algorithm in Table 4.6 originates from Courcoubetis, Vardi, Wolper and Yannakakis [60] and its implementation in the model checker SPIN has been reported by Holzmann Peled and Yannakakis [99]. An alternative algorithm would be to first compute the (maximal) strongly connected components (SCCs) of the (graph of the) BA, then check for each of these SCCs whether they contain an accept state, and finally check for reachabilty (from the state under consideration).

4.7 Exercises

EXERCISE 4.1. Consider the language that consists of all words over the alphabet $\{ a, b \}$ such that all finite words have a b-symbol on position n ($n > 0$) from the right. Assume that the last symbol is one position from the right. For instance, for n=3 we have that the word *abbaabab* is in the language.

1. Construct a non-deterministic FSA that accepts this language with at most n+1 states.
2. Determize this FSA using the subset construction algorithm. Justify the necessity of the (high) number of states in the resulting deterministic FSA.

EXERCISE 4.2. Construct a deterministic BA that accepts the complement of the language $(a \mid b)^* b^\omega$, i.e., the set of ω-words in which b occurs infinitely often.

EXERCISE 4.3. Let the alphabet $\Sigma = \{ a, b \}$. Construct a BA A that accepts infinite words w over Σ such that a occurs infinitely many times in w and between any two successive as an odd number of bs occur.

EXERCISE 4.4. Let the alphabet $\Sigma = \{ a, b, c \}$.

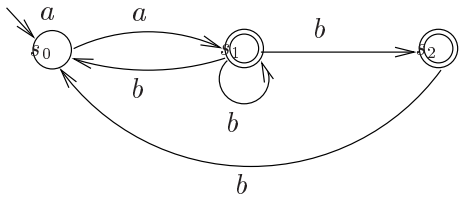
1. Construct a BA A that accepts infinite words w over Σ such that a occurs infinitely many times in w and between any two successive as an odd number of bs or an odd number of cs occur. Note that between any two successive as either only bs or only cs are allowed.
2. Repeat the previous exercise, such that any accepting word contains only finitely many c symbols.
3. Change your automaton from the first exercise such that between any two successive as an odd number of either b or c symbols may occur.
4. Same exercise, except that now an odd number of bc and an odd number of cs must occur between any two successive a symbols.

EXERCISE 4.5. The synchronous product construction of Büchi automata can be simplified considerably if we assume that in one of the two BA involved all states are accept states, i.e., $F = S$. Define this specialized product construction and compare the worst-case number of states that are generated by your construction with that of Definition 4.10.

EXERCISE 4.6. An alternative definition of the synchronous product of two BA is to define the result as a generalized BA. Define such construction and prove that your definition is ω-equivalent to the construction in Definition 4.10.

EXERCISE 4.7. Consider the generalized BA A with $\Sigma = \{ a, b \}$, $S = \{ s_1, s_2 \}$, $I = \{ s_1 \}$, $s_1 \xrightarrow{a} s_1$, $s_1 \xrightarrow{a} s_2$, $s_2 \xrightarrow{b} s_2$ and $s_2 \xrightarrow{b} s_1$ and $F_1 = \{ s_1 \}$ and $F_2 = \{ s_2 \}$. Construct a corresponding BA, i.e., $gba2ba(A)$. Justify why the resulting BA is ω-equivalent to A.

EXERCISE 4.8. Consider the following generalized BA:



with two accept sets $F_1 = \{ s_1 \}$ and $F_2 = \{ s_2 \}$. Construct the BA that corresponds to this GBA, i.e., $gba2ba(A)$ and justify why the resulting BA is ω-equivalent to A.

EXERCISE 4.9. Construct a deterministic Muller automaton that accepts the language $(a \mid b)^* b^\omega$. Justify your answer: why is your Muller automaton correct?

Chapter 5

Automata-based Model Checking of PLTL

This chapter provides the bridge between the logic PLTL introduced in Chapter 3, and Büchi automata as presented in the previous chapter. It explains the relationship between these two – at first sight, rather different – formalisms and details the basic Büchi automata-based model-checking algorithm for PLTL.

5.1 Linking Büchi automata and PLTL

What do Büchi automata have to do with PLTL-formulas? In order to understand this, suppose that we label each transition of a BA with propositional formulas over some given set AP of atomic propositions. For instance, for $AP = \{ p, q \}$, the transition $s \xrightarrow{p \wedge \neg q} s'$ means that the automaton can move from state s to state s' on the input expression $p \ \wedge \ \neg q$, i.e., when p holds and q does not, while transition $s \xrightarrow{p \vee \neg p} s'$ means that on any input ($p \vee \neg p \equiv$ true) the automaton moves from s to s'. For the moment we assume that we can technically establish such labelling by chosing the right alphabet Σ on which the BA is considered; the precise details will be given later on. For propositional formulas α_i over AP, we have – according to Büchi's acceptance criterion – that the word $\alpha_0 \, \alpha_1 \ldots$ is accepted if and only if there is an accepting run $s_0 \, s_1 \ldots$ such that $s_i \xrightarrow{\alpha_i} s_{i+1}$, for any $i \geqslant 0$. Thus, words are now infinite sequences of propositional formulas.

The key concept is: associate to PLTL-formula Φ (defined over AP) a BA that accepts all infinite words, i.e., sequences of propositional formulas, that satisfy Φ. For instance, the word $(p \ \wedge \ \neg q)^\omega$ corresponds to a computation in which it is always the case that p holds and q does not. This word thus satisfies the formula $\mathsf{G}\,(p \ \wedge \ \neg q)$. Similarly, the word $(\neg p)^* \, p \, (p \vee \neg p)^\omega$ corresponds to a

computation in which p holds at some point. This word satisfies the formula $\mathsf{F}\,p$. The following examples illustrate this idea.

Example 5.1. *Let the set of atomic propositions $AP = \{p\}$ and consider*

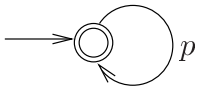


Figure 5.1: A BA that accepts sequences satisfying the formula $\mathsf{G}\,p$

the BA depicted in Figure 5.1. As any accepting run of this BA goes infinitely often through the initial state, the transition labelled with p is taken infinitely often. The language of this automaton is p^ω. This coincides with the infinite sequences of propositional formulas for which the PLTL-formula $\mathsf{G}\,p$ holds, since by definition of the semantics we have $p^\omega \models \mathsf{G}\,p$. *(End of example.)*

Example 5.2. *Consider again the simple set of atomic propositions $AP = \{p\}$*

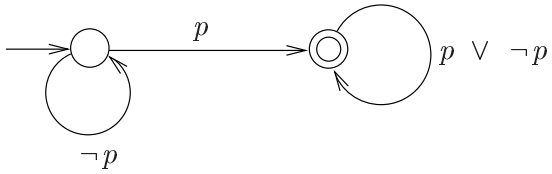


Figure 5.2: A BA that accepts sequences sastifying the formula $\mathsf{F}\,p$

together now with the BA of Figure 5.2. The initial state has two outgoing transitions. On input p the BA moves to the accept state, in which it stays permanently. On input $\neg p$, the automaton remains in its initial state. In the accept state any input is recognized without a state change. Accepted words of this automaton are of the form $(\neg p)^*\,p\,(p \vee \neg p)^\omega$. Thus this BA accepts any run that at some arbitrary point satisfies p, and anything else afterwards. This corresponds to the formula $\mathsf{F}\,p$. *(End of example.)*

Example 5.3. *As a third, and somewhat more complicated example, con-*

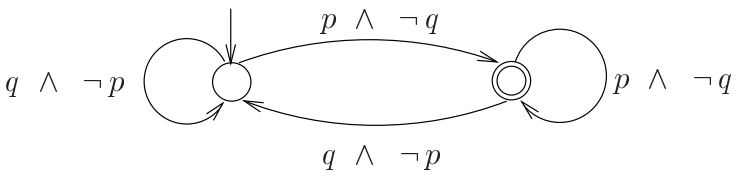


Figure 5.3: A BA that accepts sequences satisfying $\mathsf{G}\,((q \wedge \neg p)\,\mathsf{U}\,(p \wedge \neg q))$

sider $AP = \{p, q\}$ and the BA depicted in Figure 5.3. Any accepting word of this automaton contains infinitely many occurrences of the propositional formula $p \wedge \neg q$, while between any two successive visits to the accept state a

transition labelled $q \wedge \neg p$ *may be traversed finitely many times. The accepted language is thus* $((q \wedge \neg p)^*(p \wedge \neg q))^\omega$ *whose sequences satisfy the formula* $\mathsf{G}((q \wedge \neg p)\,\mathsf{U}\,(p \wedge \neg q))$. *(End of example.)*

These examples show that there exist BA that accept sequences satisfying, for instance, $\mathsf{G}p$ or $\mathsf{F}p$. It turns out that for each PLTL-formula Φ a BA exists that accepts exactly the infinite sequences that satisfy Φ. In the sequel, we will see what this result means for the model-checking procedure.

Prior to this we will detail the way in which transition labels can be propositional formulas over AP. Formally, the alphabet $\Sigma = 2^{AP}$. Thus, transitions are labelled with (sets of) sets of atomic propositions.

From these examples we infer that for $AP = \{p\}$, $\{\varnothing\}$ stands for the propositional formula $\neg p$, $\{\{p\}\}$ stands for the propositional formula p and $\{\varnothing, \{p\}\}$ stands for $\neg p \vee p$ (which equals true). Sets of sets of atomic propositions thus encode propositional formulas. More precisely, sets of subsets of a given set AP of atomic propositions encode propositional formulas over AP. Formally, if $AP_1, \ldots, AP_n \subset AP$, then the set AP_i, for $0 < i \leqslant n$ encodes the formula

$$\bigwedge_{p \in AP_i} p \;\wedge \bigwedge_{p \notin AP_i} \neg p$$

which we abbreviate by $[\![AP_i]\!]$ for the sake of convenience. The set $\{AP_{k_1}, \ldots AP_{k_m}\}$ for $m \geqslant 1$ with $0 < k_j \leqslant n$ now encodes the propositional formula

$$\bigvee_{0<j\leqslant m} [\![AP_{k_j}]\!] = [\![AP_{k_1}]\!] \vee \ldots \vee [\![AP_{k_m}]\!]$$

Note that sets of sets of atomic propositions must be non-empty.

Example 5.4. For $AP = \{p, q, r\}$ *the formula* $p \wedge \neg q$ *denotes the set of subsets* $\{p\}$ *and* $\{p, r\}$*, i.e., all subsets of* AP *in which* p *occurs and* q *does not. Proposition* r *may or may not be included. A transition* $s \xrightarrow{p \wedge \neg q} s'$ *thus means that the automaton can move from state stos'oneither*{ p }*oron*{ *p,r* }. *(Endofexample.)*

In the sequel we will label transitions of BA with propositional formulas over a given set AP of atomic propositions.

It turns out that for each PLTL-formula (on atomic propositions AP) one can find a corresponding Büchi automaton.

Theorem 5.1.
For any PLTL-formula Φ *a Büchi automaton* A *can be constructed on the alpha-*

bet $\Sigma = 2^{AP}$ *such that* $\mathcal{L}_\omega(A)$ *equals the sequences of sets of atomic propositions satisfying* Φ.

This result is due to Wolper, Vardi and Sistla (1983). The BA corresponding to Φ is denoted by A_Φ. Given this key result we can present the basic scheme for model-checking PLTL-formulas. A naive recipe for checking whether the PLTL-property Φ holds for a given model is given in Table 5.1. Büchi automata are

1.	*construct the Büchi automaton for* Φ, A_Φ
2.	*construct the Büchi automaton for the model of the system,* A_{sys}
3.	*check whether* $\mathcal{L}_\omega(A_{sys}) \subseteq \mathcal{L}_\omega(A_\Phi)$.

Table 5.1: Naive recipe for model checking PLTL

constructed for the desired property Φ and the model *sys* of the system. The accepting runs of A_{sys} correspond to the possible behaviour of the model, while the accepting runs of A_Φ correspond to the desired behaviour of the model. If all possible behaviour is desirable, i.e. when $\mathcal{L}_\omega(A_{sys}) \subseteq \mathcal{L}_\omega(A_\Phi)$, then we can conclude that the model satisfies Φ.

5.2 From PLTL to Büchi automata

5.2.1 Normal-form Formulas

In order to construct a BA for a given PLTL-formula Φ, Φ is first transformed into *normal form*. This means that Φ does neither contain the operator F nor G, and that all negations in Φ are adjacent to atomic propositions. One can easily eliminate the occurrences of F and G by using the definitions $\mathsf{F}\,\Psi \equiv \text{true}\,\mathsf{U}\,\Psi$ and $\mathsf{G}\,\Psi \equiv \neg\,\mathsf{F}\,\neg\,\Psi$. To keep the presentation simple we also assume that true and false are replaced by their definitions. In order to make it possible to transform a negated until-formula into normal form, the auxiliary temporal operator R ("release") is used. From Chapter 3 we recall that this operator is defined by

$$(\neg\,\Phi)\,\mathsf{R}\,(\neg\,\Psi) \equiv \neg\,(\Phi\,\mathsf{U}\,\Psi).$$

Its intuitive interpretation is as follows. Formula $\Phi\,\mathsf{R}\,\Psi$ holds for path σ if Ψ always holds, a requirement that is *released* as soon as Φ becomes valid. Thus, for instance, the formula $\text{false}\,\mathsf{R}\,\Phi$ is valid for σ if Φ always holds, since the release condition (false) is a contradiction.

Definition 5.1. (**Normal-form PLTL-formulas**)
For atomic proposition $p \in AP$, the set of PLTL-formulas in normal form is

defined by:

$$\Phi ::= p \mid \neg p \mid \Phi \vee \Phi \mid \Phi \wedge \Phi \mid \mathsf{X}\Phi \mid \Phi \,\mathsf{U}\, \Phi \mid \Phi \,\mathsf{R}\, \Phi.$$

The following equations are used to transform a PLTL-formula into normal form:

$$\begin{aligned}
\neg(\Phi \vee \Psi) &\equiv (\neg\Phi) \wedge (\neg\Psi) \\
\neg(\Phi \wedge \Psi) &\equiv (\neg\Phi) \vee (\neg\Psi) \\
\neg\mathsf{X}\Phi &\equiv \mathsf{X}(\neg\Phi) \\
\neg(\Phi \,\mathsf{U}\, \Psi) &\equiv (\neg\Phi) \,\mathsf{R}\, (\neg\Psi) \\
\neg(\Phi \,\mathsf{R}\, \Psi) &\equiv (\neg\Phi) \,\mathsf{U}\, (\neg\Psi)
\end{aligned}$$

Note that in each of these equations, reading them from left to right, the outermost negation is "pushed" inside the formula. Applying these rules recursively, allows one to push negations until they are adjacent to atomic propositions.

Example 5.5. *As an example we derive the normal-form of the formula* $\neg\mathsf{X}(r \Rightarrow p \,\mathsf{U}\, q)$.

$$\begin{aligned}
& \neg\mathsf{X}(r \Rightarrow p \,\mathsf{U}\, q) \\
\Leftrightarrow\ & \{ \textit{ definition of } \Rightarrow \} \\
& \neg\mathsf{X}(\neg r \vee (p \,\mathsf{U}\, q)) \\
\Leftrightarrow\ & \{ \neg\mathsf{X}\Phi \Leftrightarrow \mathsf{X}(\neg\Phi) \} \\
& \mathsf{X}(\neg(\neg r \vee (p \,\mathsf{U}\, q))) \\
\Leftrightarrow\ & \{ \textit{ predicate calculus } \} \\
& \mathsf{X}(r \wedge \neg(p \,\mathsf{U}\, q)) \\
\Leftrightarrow\ & \{ \textit{ definition of } \mathsf{R} \} \\
& \mathsf{X}(r \wedge (\neg p) \,\mathsf{R}\, (\neg q)).
\end{aligned}$$

As a next example, the interested reader may check that the normal form of $\mathsf{G}\,p$ *is* $(q \vee \neg q) \,\mathsf{R}\, p$. *(End of example.)*

It is not hard to see that the worst-case time complexity of transforming Φ into normal form is linear in the length of Φ, denoted by $\mid \Phi \mid$. The length of Φ can easily be defined by induction on the structure of Φ, e.g. $\mid p \mid = 1$, $\mid \mathsf{X}\Phi \mid = 1 + \mid \Phi \mid$ and $\mid \Phi \,\mathsf{U}\, \Psi \mid = 1 + \mid \Phi \mid + \mid \Psi \mid$. For the sake of brevity we do not give the full definition here.

5.2.2

Definition 5.2. (**Closure**)
For PLTL-formula Φ, the *closure* of Φ, denoted $closure(\Phi)$ is defined as the set of formulas satisfying:

- $\Phi \in closure(\Phi)$
- $\Psi \wedge \Psi' \in closure(\Phi) \Rightarrow \Psi, \Psi' \in closure(\Phi)$
- $\Psi \vee \Psi' \in closure(\Phi) \Rightarrow \Psi, \Psi' \in closure(\Phi)$
- $\mathsf{X}\,\Psi \in closure(\Phi) \Rightarrow \Psi \in closure(\Phi)$
- $\Psi\,\mathsf{U}\,\Psi' \in closure(\Phi) \Rightarrow \Psi, \Psi' \in closure(\Phi)$
- $\Psi\,\mathsf{R}\,\Psi' \in closure(\Phi) \Rightarrow \Psi, \Psi' \in closure(\Phi)$.

Example 5.6.

(End of example.)

5.2.3 Labelling Sequences

5.2.4 Defining the Automaton

5.2.5 Possible Optimisations

5.3 Model-checking PLTL

This seems to be a plausible approach. However, the problem to decide language inclusion of Büchi automata is PSPACE-complete, i.e. it is a difficult type of NP-problem.[1]

Observe that

$$\mathcal{L}_\omega(A_{sys}) \subseteq \mathcal{L}_\omega(A_\Phi) \Leftrightarrow (\mathcal{L}_\omega(A_{sys}) \cap \mathcal{L}_\omega(\overline{A_\Phi}) = \varnothing)$$

[1] PSPACE-complete problems belong to the category of problems that can still be solved in polynomial space, i.e. in a memory space that is polynomial in the size of the problem (the number of states in the Büchi automaton, for instance). Therefore, for these problems it is very unlikely to find algorithms that do not have an exponential time complexity.

where $\overline{A}$ is the complement of A that accepts $\Sigma^\omega \setminus \mathcal{L}_\omega(A)$ as a language. The construction of $\overline{A}$ for BA A, is however, quadratically exponential: if A has n states then $\overline{A}$ has c^{n^2} states, for some constant $c > 1$. This means that the resulting automaton is very large. (For deterministic BAs an algorithm exists that is polynomial in the size of A, but since deterministic BAs are strictly less expressive than non-deterministic BAs, this does not interest us here.)

Using the observation that the complement automaton of A_Φ is equal to the automaton for the negation of Φ:

$$\mathcal{L}_\omega(\overline{A_\Phi}) = \mathcal{L}_\omega(A_{\neg\,\Phi})$$

we arrive at the following more efficient method for model checking PLTL which is usually more efficient. This is shown in Table 5.2. The idea is to construct

1.	*construct the Büchi automaton for* $\neg\,\Phi$, $A_{\neg\,\Phi}$
2.	*construct the Büchi automaton for the model of the system,* A_{sys}
3.	*check whether* $\mathcal{L}_\omega(A_{sys}) \cap \mathcal{L}_\omega(A_{\neg\,\Phi}) = \varnothing$.

Table 5.2: Basic recipe for model checking PLTL

an BA for the negation of the desired property Φ. Thus the automaton $A_{\neg\,\Phi}$ models the undesired computations of the model that we want to check. If A_{sys} has a certain accepting run that is also an accepting run of $A_{\neg\,\Phi}$ then this is an example run that violates Φ, so we conclude that Φ is not a property of A_{sys}. If there is no such common run, then Φ is satisfied. This explains step 3 of the method. Emerson and Lei (1985) have proven that the third step is decidable in linear time, and thus falls outside the class of NP-problems.

In the sequel we will assume that the automaton A_{sys} is given. The step from a programming or specification language to an BA depends very much on the language considered, and is usually also not that difficult. In the next section we provide an algorithm for step 1 of the method. Later in this chapter we also explain in detail how step 3 is performed.

5.3.1 Basic Model-Checking Procedure

Definition 5.3. (**From Kripke structure to Büchi automaton**)
For Kripke structure $K = (S, I, \longrightarrow, \mathit{Label})$ with $\mathit{Label} : S \longrightarrow 2^{AP}$, the BA $\mathit{ks2ba}(K)$ is defined as $(\Sigma, S', I', \longrightarrow', F)$ where $\Sigma = 2^{AP}$, $S' = S \cup \{\, s \,\}$ with $s \notin S$, $I' = \{\, s \,\}$, $F = S'$ and $\longrightarrow'$ the smallest relation satisfying:

- $s \xrightarrow{\alpha} s'$ if and only if $s' \in I$ and $\alpha = \mathit{Label}(s')$

- $s' \xrightarrow{\alpha} s''$ if and only if $s' \longrightarrow s''$ and $\alpha = Label(s'')$.

The basic idea behind this construction is as follows. As indicated earlier, the alphabet of the BA is the powerset of the set of atomic proposition under consideration. A new state s is introduced ($s \notin S$) that becomes the initial state of the BA. From this state, transitions are emanating to all initial states of the Kripke structure. The other transitions are identical to the transitions in K. All transitions are labelled with the atomic propositions of the target state of the transition. All states in $ks2ba(K)$ are accept states.

5.3.2

An overview of the different steps of model checking PLTL is shown in Figure 5.4. The model of the system sys is transformed into a Büchi automaton A_{sys} in which all states are accepting. The property to be checked is specified in PLTL — yielding Φ —, negated, and subsequently transformed into a second Büchi automaton $A_{\neg\Phi}$. The product of these two automata represents all possible computations that violate Φ. By checking the emptiness of this automaton, it is thus determined whether Φ is satisfied by the system-model sys or not.

Although the various steps in the algorithm are presented in a strict sequential order, that is, indicating that the next step cannot be performed, until the previous ones are completely finished, some steps can be done *on-the-fly*. For instance, constructing the graph for a normal-form formula can be performed while checking the emptiness of the product automaton $A_{sys} \otimes A_{\neg\Phi}$. In this way the graph is constructed "on demand": only a new vertex is considered if no accepting cycle has been found yet in the partially constructed product automaton. When the successors of a vertex in the graph are constructed, one chooses the successors that match the current state of the automaton A_{sys} rather than all possible successors. Thus it is possible that an accepting cycle is found (i.e. a violation of Φ with corresponding counter-example) without the need for generating the entire graph $\mathcal{G}_\Phi$. In a similar way, the automaton A_{sys} does not need to be totally available before starting checking non-emptiness of the product automaton. This is usually the most beneficial step of on-the-fly model checking, since this automaton is typically rather large, and avoiding the entire consideration of this automaton may reduce the state space requirements significantly.

We conclude by discussing the worst-case time complexity of the model-checking recipe for PLTL-formulas. Let Φ be the formula to be checked and sys the model of the system under consideration. The crucial step is the transformation of a normal form PLTL-formula into a graph. Since each vertex in the graph is labelled with a set of sub-formulas of Φ, the number of vertices in the graph

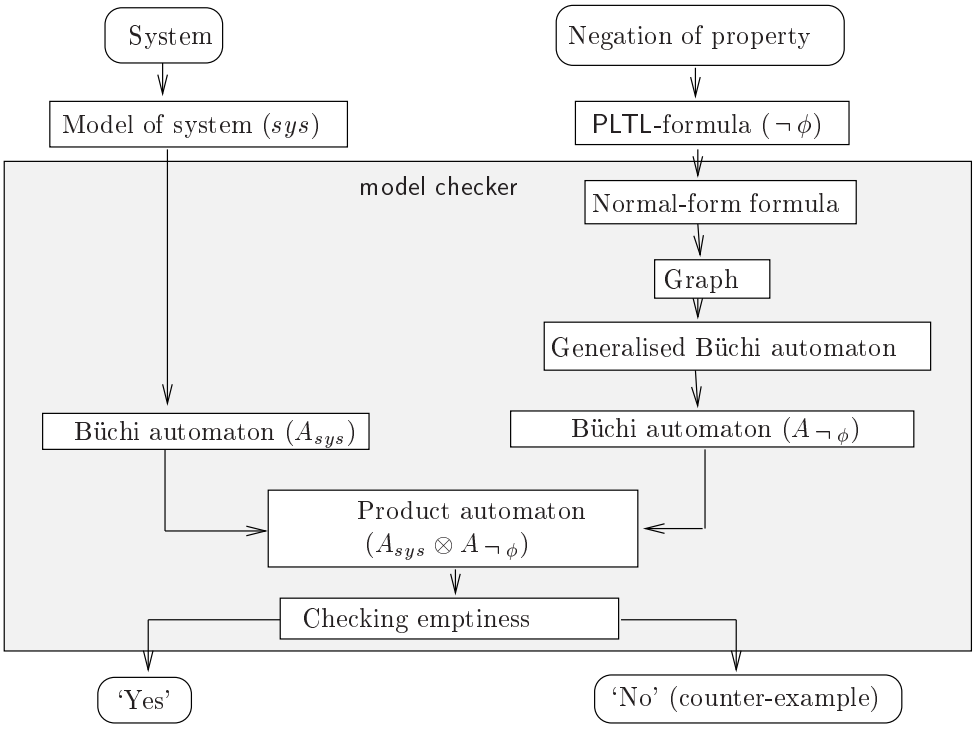


Figure 5.4: Overview of model-checking PLTL

is proportional to the number of sets of sub-formulas of Φ, that is $\mathcal{O}(2^{|\Phi|})$. Since the other steps of the transformation of Φ into an BA do not affect this worst-case complexity, the resulting BA has a state space of $\mathcal{O}(2^{|\Phi|})$. The worst-case space complexity of the product automaton $A_{sys} \otimes A_{\neg \Phi}$ is therefore $\mathcal{O}(\mid S_{sys} \mid \times 2^{|\Phi|})$ where S_{sys} denotes the set of states in the BA A_{sys}. Since the time complexity of checking the emptiness of an BA is linear in the number of states and transitions we obtain that

> *The worst-case time complexity of checking whether system-model sys satisfies the PLTL-formula Φ is* $\mathcal{O}(\mid S_{sys} \mid^2 \times 2^{|\Phi|})$

since in worst case we have $\mid S_{sys} \mid^2$ transitions. In the literature it is common to say that the time complexity of model checking PLTL is linear in the size of the model (rather than quadratic) and exponential in the size of the formula to be checked.

The fact that the time complexity is exponential in the length of the PLTL-formula seems to be a major obstacle for the application of this algorithm to practical systems. Experiments have shown that this dependency is not significant, since the length of the property to be checked is usually rather short. This is also justified by several industrial case studies. (Holzmann, for instance, declares that "PLTL-formulas have rarely more than 2 or 3 operators".)

We conclude by briefly indicating the space complexity of model checking PLTL. The model checking problem for PLTL is PSPACE-complete, i.e. a state space that is polynomial in the size of the model and the formula is needed (Sistla and Clarke, 1985).

5.4 Summary

5.5 Bibliographic Notes

5.6 Exercises

Chapter 6

Computation Tree Logic

This chapter introduces Computation Tree Logic (CTL), a prominent branching temporal logic for specifying relevant system properties. In particular, propositional CTL is presented, covering its syntax, semantics and (briefly) axiomatization, a comparison to propositional LTL, a discussion about fairness and its practical usage as a property specification language.

6.1 Introduction

Pnueli [151] has introduced linear temporal logic to the computer science community for the specification and verification of reactive systems. In Chapter 3, we have treated one important kind of linear temporal logic, namely propositional LTL (PLTL). This temporal logic is called linear, because the – qualitative notion of – time viewed to be linear: at each moment of time there is only one possible successor state and thus each time moment has a unique possible future. Technically speaking, this follows from the fact that the interpretation of linear temporal logic-formulas (by the satisfaction relation $\models$) is defined in terms of computations, i.e., sequences of states. Due to this basis on sequences, the temporal operators X, U, F and G in fact describe the ordering of events along a time path, i.e., a *single* computation of a system.

Paths themselves, though, are obtained from a Kripke structure that has branching: in such automaton a state may have several, distinct direct successor states, and thus several computations may start in a state. The interpretation of PLTL-formulae is lifted to the notion of a state by requiring that a formula φ holds in state s if *all* possible computations that start in s satisfy φ. The universal quantification over all computations that start in s can also be made explicit

in the formula, for instance:

$$s \models \mathsf{A}\,\varphi \text{ if and only if } \sigma \models \varphi \text{ for all paths } \sigma \text{ starting in } s$$

where $\mathsf{A}\,\varphi$ is a PLTL-formula whose interpretation is defined over states (rather than paths). In linear temporal logic, we thus can state properties over all possible computations that start in a state, but not easily about *some* of such computations. To some extent this may be overcome by exploiting the duality between universal and existential quantification. For instance, to check whether there exists some computation starting in s that satisfies φ we may check whether $s \models \mathsf{A}\,\neg\,\varphi$; if this formula is satisfied by s, then there must be a computation that meets φ, otherwise they should all refute φ.

For more complicated properties, like "for every computation it is always possible to return to the initial state", this is, however, not possible. A naive attempt would be to require $\mathsf{G}\,\mathsf{F}\,\mathit{start}$ for every computation, i.e., $s \models \mathsf{A}\,\mathsf{G}\,\mathsf{F}\,\mathit{start}$, where the proposition *start* uniquely identifies the initial state/ This is, however, too strong as it requires a computation to always return to the initial state, not just possibly. Other attempts to specify the intended property also fail, and it even turns out to be the case that the property cannot be specified in PLTL.

To overcome these problems, in the early eighties another strand of temporal logics for specification and verification purposes was introduced by Clarke and Emerson [46]. The semantics of this kind of temporal logic is not based on a linear notion of time – an infinite sequence of states – but on a *branching* notion of time – an infinite *tree* of states. Branching time refers to the fact that at each moment there may be several different possible futures. Each moment of time may thus split into several possible futures. Due to this branching notion of time, this class of temporal logic is known as *branching* temporal logic. The underlying notion of the semantics of a branching temporal logic is a *tree* of states rather than a sequence. Each path in the tree is intended to represent a single possible computation. The tree itself thus represents all possible computations, and is directly obtained from a Kripke structure by "unfolding" the automaton at the state of interest. The tree rooted at state s thus represents all possible infinite computations in the Kripke structure that start in s.

The temporal operators in branching temporal logic allow the expression of properties of *some* or *all* computations that start in a state. To that end, it supports an existential path quantifier (denoted E) and a universal path quantifier (denoted A) For instance, the property $\mathsf{EF}\,\Phi$ denotes that there exists a computation along which $\mathsf{F}\,\Phi$ holds. That is, it states that there is at least one possible computation in which a state that satisfies Φ is eventually reached. This does not, however, exclude the fact that there can also be computations for which this property does not hold, for instance, computations for which Φ is always refuted. The property $\mathsf{AF}\,\Phi$, in contrast, states that all computations

satisfy the property $\mathsf{F}\,\Phi$. More complicated properties can be expressed by nesting universal and existential path quantifiers. For instance, the aforementioned property "for every computation it is always possible to return to the initial state" can be faithfully expressed by $\mathsf{AG}\,\mathsf{EF}\,\mathit{start}$: in any state ($\mathsf{G}$) of any possible computation (A), there is a possibility (E) to eventually return to the start state ($\mathsf{F}\,\mathit{start}$).

The existence of two types of temporal logic – linear and branching temporal logic – has resulted in the development of two model-checking "schools", one based on linear and one based on branching temporal logic. Although much can be said about the differences and the appropriateness of linear versus branching temporal logic, there are two main issues that justify the treatment of model checking based on these different logics:

- The expressiveness of many linear and branching temporal logics is incomparable. This means that some properties that are expressible in a linear temporal logic cannot be expressed in certain branching temporal logics, and vice versa.
- The traditional techniques for model-checking linear and branching temporal logics are quite different.[1] This results, for instance, in significantly different time and space complexity results.

In this chapter, we consider Computation Tree Logic (CTL), a temporal logic based on propositional logic with a discrete notion of time, and only future modalities. CTL is an important branching temporal logic that is sufficiently expressive for the formulation of an important set of system properties. It was originally used by Clarke and Emerson [46] and (in a slightly different form) by Quielle and Sifakis [155] for model checking. More importantly, it is a logic for which efficient, and – as we will see in Chapter 7 – very elegant and simple, model-checking algorithms do exist.

6.2 Syntax of CTL

The most elementary statements about systems that one can make in CTL are atomic propositions, as in the definition of PLTL. The finite set of atomic propositions is denoted by AP, with typical elements p, q, and r. We define the syntax of CTL in the following way:

Definition 6.1. (**Syntax of computation tree logic**)
Let p be an atomic proposition. Formulas in CTL are either state-formulas or path-formulas. State-formulas satisfy the following rules:

[1] Despite some promising unifying developments are taking place [114].

1. p is a state-formula
2. If Φ is a state-formula, then $\neg\,\Phi$ is a state-formula
3. If Φ and Ψ are state-formulas, then $\Phi \vee \Psi$ is a state-formula
4. If φ is a path-formula, then $\mathsf{E}\,\varphi$ and $\mathsf{A}\,\varphi$ are state-formulas
5. Anything else is not a state-formula.

Path-formulas satisfy the following rules:

1. If Φ is a state-formula, then $\mathsf{X}\,\Phi$ is a path-formula
2. If Φ and Ψ are state-formulas, then $\Phi\,\mathsf{U}\,\Psi$ is a path-formula
3. Anything else is not a path-formula.

CTL distinguishes between state-formulas and path-formulas. Intuitively, state-formulas express a property of a state, while path-formulas express a property of a path, i.e., an infinite sequence of states. The temporal operators X and U have the same meaning as in PLTL and are path-operators. Formula $\mathsf{X}\,\Phi$ holds for a path if Φ holds at the next state in the path, and $\Phi\,\mathsf{U}\,\Psi$ holds for a path if there is some future state along the path for which Ψ holds, and Φ holds in all states prior to that state. Path-formulas can be turned into state-formulas by prefixing them with either the path-quantifier E (pronounced "for some path") or the path-quantifier A (pronounced "for all paths"). Note that the linear temporal operators X and U are required to be immediately preceded by E or A to obtain a legal state-formula. Formula $\mathsf{E}\,\varphi$ holds in a state if there exists *some* path satisfying φ that starts in that state. Dually, $\mathsf{A}\,\varphi$ holds in a state if *all* paths that start in that state satisfy φ. Like for universal and existential quantification in first-order logic, we have $\mathsf{A}\,\varphi \equiv \neg\,\mathsf{E}\,\neg\,\varphi$, for all path-formulas φ.[2]

Example 6.1. *Let* $AP = \{\, x = 1, x < 2, x \geqslant 3 \,\}$ *be a set of atomic propositions. Examples of syntactically correct CTL-formulas are:*

$$\mathsf{EX}\,(x = 1), \mathsf{AX}\,(x = 1), \text{ and } x < 2 \;\vee\; x = 1$$

and

$$\mathsf{E}\,((x < 2)\,\mathsf{U}\,(x \geqslant 3)) \text{ and } \mathsf{A}\,(\mathit{true}\,\mathsf{U}\,(x < 2))$$

Some example formulas that are syntactically incorrect are:

$$\mathsf{E}\,(x = 1 \;\wedge\; \mathsf{AX}\,(x \geqslant 3)) \text{ and } \mathsf{EX}\,(\mathit{true}\,\mathsf{U}\,(x = 1))$$

[2] Note, however, that $\neg\,\mathsf{E}\,\neg\,\varphi$ is not a CTL-formula since $\neg\,\varphi$ is not a legal path-formula.

The first is not a CTL-formula since $x = 1 \ \wedge \ \mathsf{AX}\,(x \geqslant 3)$ *is not a path-formula and thus cannot be preceded by* E. *The second formula is not a CTL-formula since* $\mathit{true}\,\mathsf{U}\,(x = 1)$ *is a path-formula rather than a state-formula, and thus cannot be preceded by* X. *Note that*

$$\mathsf{EX}\,(x = 1 \ \wedge \ \mathsf{AX}\,(x \geqslant 3)) \ \textit{and} \ \mathsf{EX}\,\mathsf{A}\,(\mathit{true}\,\mathsf{U}\,(x = 1))$$

are, however, syntactically correct CTL-formulas. *(End of example.)*

The syntax of CTL requires that the linear temporal operators X, F, G, and U are immediately preceded by a path quantifier E or A. If this restriction is dropped, and we allow an arbitrary PLTL-formula to be preceded by E or A, then the more expressive branching temporal logic CTL* [46] is obtained. For instance, $\mathsf{E}\,(p \ \wedge \ \mathsf{X}\,q)$ and $\mathsf{A}\,(\mathsf{F}\,p \ \wedge \ \mathsf{G}\,q)$ are formulas in CTL*, but not in CTL. CTL* can therefore be considered as *the* branching counterpart of PLTL since each PLTL sub-formula can be used in a CTL*-formula. The precise relationship between PLTL, CTL and CTL* will be described in Section 6.5. We do not consider model-checking CTL* since it is of intractable complexity: the model-checking problem for this logic is PSPACE-complete in the size of the system specification [47]. Here, we only consider CTL for which efficient model-checking algorithms do exist. Although CTL does not possess the full expressive power of CTL*, there is substantial empirical evidence that it is usually sufficiently powerful to express relevant properties.

6.3 Semantics of CTL

6.3.1 Kripke Structures

As for PLTL, the interpretation of CTL is defined in terms of a Kripke structure. We recall the following definition and notations:

Definition 6.2. (**Kripke structure**)
A *Kripke structure* $\mathcal{K}$ is a tuple $(S, I, R, \mathit{Label})$ where

- S is a countable set of states,
- $I \subseteq S$ is a set of initial states,
- $R \subseteq S \times S$ is a transition relation satisfying $\forall s \in S.\,(\exists s' \in S.\,(s, s') \in R)$
- $\mathit{Label} : S \longrightarrow 2^{AP}$ is an interpretation function on S.

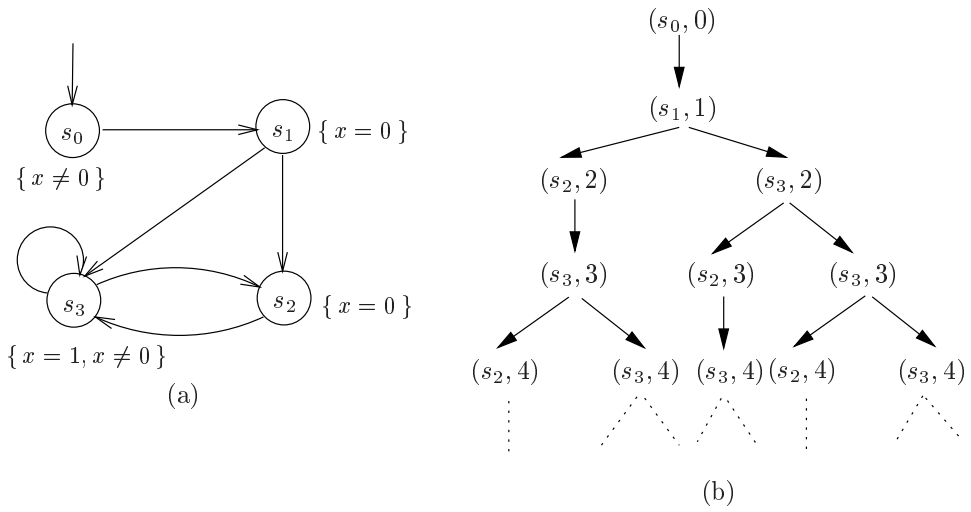


Figure 6.1: An example (a) Kripke structure and (b) a prefix of one of its infinite computation trees

Definition 6.3. (**Path**)
A *path* in Kripke structure $\mathcal{K} = (S, I, R, Label)$ is an infinite sequence of states $s_0\, s_1\, s_2 \ldots$ such that $(s_i, s_{i+1}) \in R$ for all $i \geqslant 0$.

A path is thus an infinite sequence of states such that between successive states transitions do exist. For path $\sigma = s_0\, s_1\, s_2 \ldots$ and integer $i \geqslant 0$ we use $\sigma[i]$ to denote the $(i{+}1)$-th state of σ , i.e., $\sigma[i] = s_i$. The set of paths that start in state s is denoted $Paths(s)$. As each state in a Kripke structure is required to have at least one successor, it follows $Paths(s) \neq \varnothing$ for any state s. A state s for which $p \in Label(s)$ is sometimes called a p-state. The path σ is called a p-path if it consists solely of p-states.

For any Kripke structure $\mathcal{K} = (S, I, R, Label)$ and state $s \in S$ there is an infinite computation tree with root labeled s such that (s', s'') is an arc in the tree if and only if $(s', s'') \in R$. This tree is obtained by unfolding the Kripke structure at state s. The out-degree of a node in the tree is given by the number of outgoing transitions in the Kripke structure.

Example 6.2. Consider the Kripke structure of Figure 6.1(a). It consists of four states with a single initial state s_0. A finite prefix of the infinite computation tree rooted at state s_0 is depicted in Figure 6.1(b). This tree is obtained by unfolding the Kripke structure starting at state s_0. For convenience, each node in the tree consists of a pair indicating the state in the Kripke structure and the level of the node in the tree. Paths are obtained by traversing the tree in a downward fashion starting from the root. Examples paths are $s_0\, s_1\, s_2\, s_3^\omega$, $s_0\, s_1\, (s_2\, s_3)^\omega$ and $s_0\, s_1\, (s_3\, s_2)^*\, s_3^\omega$. A similar strategy can be followed for state

s_3. One thus obtains for the set of paths that start in state s_3:

$$Paths(s_3) = (s_3^+\ s_2)^* s_3^\omega \cup (s_3^+\ s_2)^\omega.$$

(End of example.)

6.3.2 Semantics of CTL

The semantics of CTL formulas is defined by two satisfaction relations (both denoted by $\models$): one for the state-formulas and one for the path-formulas. For the state-formulas, $\models$ is a relation between a Kripke structure $\mathcal{K}$, one of its states s, and a state-formula Φ. We write $\mathcal{K}, s \models \Phi$ rather than $((\mathcal{K}, s), \Phi) \in \models$. The intended interpretation is: $\mathcal{K}, s \models \Phi$ if and only if state-formula Φ holds in state s of structure $\mathcal{K}$. For the path-formulas, $\models$ is a relation between a Kripke structure $\mathcal{K}$, one of its paths σ, and a path-formula φ. We write $\mathcal{K}, \sigma \models \Phi$ rather than $((\mathcal{K}, \sigma), \Phi) \in \models$. The intended interpretation is: $\mathcal{K}, \sigma \models \varphi$ if and only if path σ in model $\mathcal{K}$ satisfies path-formula φ. For convenience, we omit $\mathcal{K}$ if it is clear from the context.

Definition 6.4. (**Semantics of CTL**)
Let $p \in AP$ be an atomic proposition, $\mathcal{K} = (S, I, R, Label)$ be a Kripke structure, $s \in S$, Φ, Ψ be CTL state-formulas, and φ be a CTL path-formula. The satisfaction relation $\models$ is defined for state-formulas by:

$$\begin{array}{lll}
s \models p & \text{iff} & p \in Label(s) \\
s \models \neg\, \Phi & \text{iff} & \text{not } s \models \Phi \\
s \models \Phi \vee \Psi & \text{iff} & (s \models \Phi) \text{ or } (s \models \Psi) \\
s \models \mathsf{E}\,\varphi & \text{iff} & \sigma \models \varphi \text{ for some } \sigma \in Paths(s) \\
s \models \mathsf{A}\,\varphi & \text{iff} & \sigma \models \varphi \text{ for all } \sigma \in Paths(s)
\end{array}$$

For path σ the satisfaction relation $\models$ for path-formulas is defined by:

$$\begin{array}{lll}
\sigma \models \mathsf{X}\,\Phi & \text{iff} & \sigma[1] \models \Phi \\
\sigma \models \Phi\,\mathsf{U}\,\Psi & \text{iff} & \exists\, j \geqslant 0.\ (\sigma[j] \models \Psi \ \wedge\ (\forall\, 0 \leqslant k < j.\ \sigma[k] \models \Phi))
\end{array}$$

The interpretations for atomic propositions, negation and conjunction are as usual, where it should be noted that in CTL they are interpreted over states, whereas in PLTL they are interpreted over paths. State-formula $\mathsf{E}\,\varphi$ is valid in state s if and only if there exists some path starting in s that satisfies φ. In contrast, $\mathsf{A}\,\varphi$ is valid in state s if and only if all paths starting in s satisfy φ. The semantics of the path-formulas is identical (although slightly different formulated) to that for PLTL. For instance, $\mathsf{EX}\,\Phi$ is valid in state s if and only if there exists some path σ starting in s such that in the next state of this path, state $\sigma[1]$, the property Φ holds. $\mathsf{A}\,(\Phi\,\mathsf{U}\,\Psi)$ is valid in state s if and only if every

Property	*Formalization in CTL*
Possibly the system never goes down	$\mathsf{EG}\ \neg\, down$
Invariantly the system never goes down	$\mathsf{AG}\ \neg\, down$
It is always possible to start as new	$\mathsf{AG}\,\mathsf{EF}\ up_3$
The system only goes down when operational	$\mathsf{A}\ ((up_3\ \vee\ up_2)\,\mathsf{U}\, down)$

Table 6.1: Some properties for the TMR system and their formalization in CTL

path starting in s has an initial finite prefix (possibly only containing s) such that Ψ holds in the last state of this prefix and Φ holds in all other states along the prefix. $\mathsf{E}\,(\Phi\,\mathsf{U}\,\Psi)$ is valid in s if and only if there exists a path starting in s that satisfies $\Phi\,\mathsf{U}\,\Psi$. As for PLTL, the semantics of CTL here is non-strict in the sense that the formula $\Phi\,\mathsf{U}\,\Psi$ is valid if the current state satisfies Ψ.

Example 6.3. Consider the Kripke structure modeling the TMR system, as

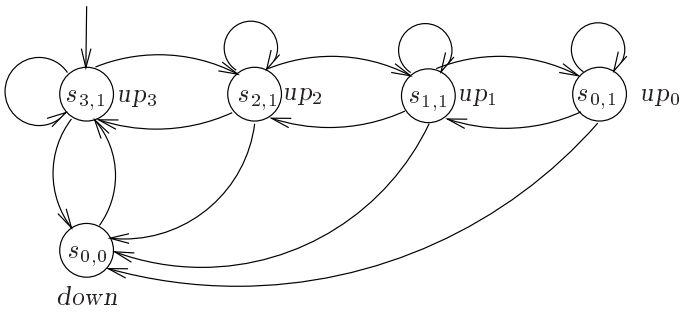


Figure 6.2: A Kripke structure of the TMR system

introduced in Chapter 3 (cf. Figure 6.2). Recall that states are of the form $s_{i,j}$ where i denotes the number of processors that is currently up ($0 < i \leqslant 3$) and j the number of operational voters ($j = 0, 1$). We consider the TMR system to be operational if at least two processors are functioning properly. Some interesting properties of this system and their formulation in CTL are listed in Table 6.1. We consider each of the formulae in isolation:

- *State-formula* $\mathsf{EG}\ \neg\, \mathrm{down}$ *holds in state* $s_{3,1}$*, as there is a path that starts in that state that never reaches the* down*-state. An example of such path is* $(s_{3,1}\, s_{2,1})^{\omega}$*.*
- *Formula* $\mathsf{AG}\ \neg\, \mathrm{down}$*, however, does not hold in state* $s_{3,1}$*, as there is a path starting from that state that satisfies* $\neg\, \mathsf{G}\ \neg\, \mathrm{down}$*, or equivalently* $\mathsf{F}\, \mathrm{down}$*, i.e., that eventually goes down. An example of such path is* $(s_{3,1})^{+}\, s_{0,0} \ldots$*.*
- *Formula* $\mathsf{AG}\,\mathsf{EF}\ \mathrm{up}_3$ *holds in state* $s_{3,1}$*, as in any state of any of its paths it is possible to return to the initial state, e.g., first moving to state* $s_{0,0}$

and then to $s_{3,1}$. This property should not be confused with the CTL-formula $\mathsf{AF}\, up_3$*, which expresses that each path eventually will visit the initial state. (Note that this formula is trivially valid for state $s_{3,1}$ as it satisfies up_3.)*

- *The last property of Table 6.1 does not hold in state $s_{3,1}$ as there exists a path, such as $s_{3,1}\, s_{2,1}\, s_{1,1}\, s_{0,0} \ldots$, for which the path-formula* $(up_3 \vee up_2)\ \mathsf{U}\ down$ *does not hold. The formula is refuted since the path visits state $s_{1,1}$, a state that neither satisfies down, nor up_3, nor up_2.*

(End of example.)

6.3.3 Auxiliary Temporal Operators

The Boolean operators true, false, $\wedge$, $\Rightarrow$ and $\Leftrightarrow$ are defined in the usual way (see Chapter 3). For instance, $\Phi \wedge \Psi = \neg(\neg\Phi \vee Not\Psi)$. Given that $\mathsf{F}\,\Phi = \mathrm{true}\,\mathsf{U}\,\Phi$ we define the following abbreviations:

$$\begin{aligned} \mathsf{EF}\,\Phi &\equiv \mathsf{E}\,(\mathrm{true}\,\mathsf{U}\,\Phi) \\ \mathsf{AF}\,\Phi &\equiv \mathsf{A}\,(\mathrm{true}\,\mathsf{U}\,\Phi) \end{aligned}$$

$\mathsf{EF}\,\Phi$ is pronounced "Φ holds potentially" and $\mathsf{AF}\,\Phi$ is pronounced "Φ is inevitable". Since $\mathsf{G}\,\Phi \equiv \neg\,\mathsf{F}\,\neg\Phi$ and $\mathsf{A}\,\varphi \equiv \neg\,\mathsf{E}\,\neg\,\varphi$ we have in addition:

$$\begin{aligned} \mathsf{EG}\,\Phi &\equiv \neg\,\mathsf{AF}\,\neg\Phi \\ \mathsf{AG}\,\Phi &\equiv \neg\,\mathsf{EF}\,\neg\Phi \\ \mathsf{AX}\,\Phi &\equiv \neg\,\mathsf{EX}\,\neg\Phi \end{aligned}$$

$\mathsf{EG}\,\Phi$ is pronounced "potentially always Φ", $\mathsf{AG}\,\Phi$ is pronounced "invariantly Φ" and $\mathsf{AX}\,\Phi$ is pronounced "for all paths next Φ". The operators E and A bind equally strongly and have the highest precedence among the unary operators. The binding power of the other operators is identical to that of the linear temporal logic PLTL. Thus, for example, $(\mathsf{AG}\,p) \Rightarrow (\mathsf{EG}\,q)$ is simply denoted by $\mathsf{AG}\,p \Rightarrow \mathsf{EG}\,q$, and should not be confused with $\mathsf{AG}\,(p \Rightarrow \mathsf{EG}\,q)$.

The interpretation of the temporal operators $\mathsf{AX}\,\Phi$, $\mathsf{EF}\,\Phi$, $\mathsf{EG}\,\Phi$, $\mathsf{AF}\,\Phi$ and $\mathsf{AG}\,\Phi$ can be derived using the semantics of CTL, cf. Definition 6.4. To illustrate this we derive for $\mathsf{EG}\,\Phi$:

$$\begin{aligned} & s \models \mathsf{EG}\,\Phi \\ \Leftrightarrow\ & \{\ \text{definition of EG}\ \} \\ & s \models \neg\,\mathsf{AF}\,\neg\Phi \\ \Leftrightarrow\ & \{\ \text{definition of F}\ \} \end{aligned}$$

$$
\begin{aligned}
& s \models \neg \mathsf{A}\,(\mathrm{true}\,\mathsf{U}\,\neg\Phi) \\
\Leftrightarrow\ & \{\ \text{semantics of } \neg\ \} \\
& \neg\,(s \models \mathsf{A}\,(\mathrm{true}\,\mathsf{U}\,\neg\Phi)) \\
\Leftrightarrow\ & \{\ \text{semantics of } \mathsf{A}\,\mathsf{U}\ \} \\
& \neg\,(\forall\,\sigma \in \mathit{Paths}(s).(\exists\, j \geqslant 0.\, \sigma[j] \models \neg\Phi\ \wedge\ (\forall\, 0 \leqslant k < j.\, \sigma[k] \models \mathrm{true}))) \\
\Leftrightarrow\ & \{\ s \models \mathrm{true} \text{ for all states } s \text{ ; predicate calculus } \} \\
& \exists\,\sigma \in \mathit{Paths}(s).\,(\forall\, j \geqslant 0.\, \sigma[j] \models \Phi)
\end{aligned}
$$

Thus $\mathsf{EG}\,\Phi$ is valid in state s if and only if there exists some path starting at s such that for each state on this path the formula Φ holds. In a similar way one can derive that $\mathsf{AG}\,\Phi$ is valid in state s if and only if for all states on any path starting at s the formula Φ holds. The formula $\mathsf{EF}\,\Phi$ is valid in state s if and only if Φ holds eventually along some path that starts in s, and $\mathsf{AF}\,\Phi$ is valid if and only if this property holds for all paths that start in s. The derivation of the formal interpretation of these temporal operators is left to the interested reader. A schematic overview of the validity of EG, EF, AF and AG is given in Figure 6.3, where black colored states satisfy the predicate *black*, while other states do not.

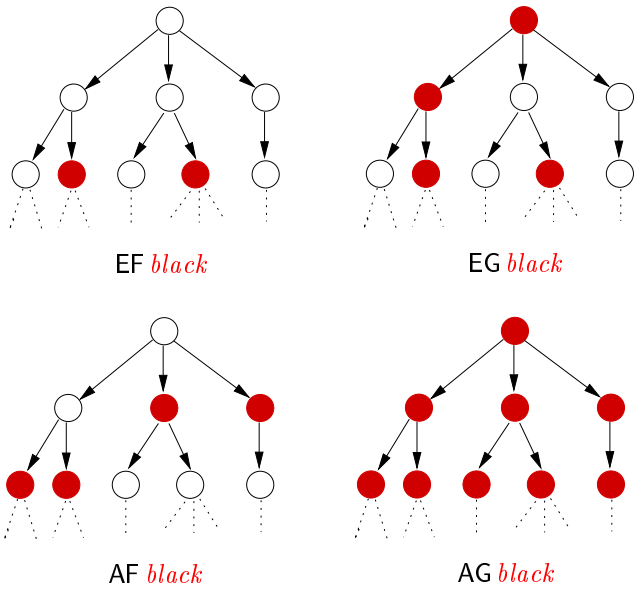


Figure 6.3: Example unfolding and the validity of some basic CTL formulae

Example 6.4. *A Kripke structure $\mathcal{K}$ is depicted at the top of Figure 6.4(a), and underneath the validity of several CTL-formulas is indicated for each state of $\mathcal{K}$. (For simplicity, the initial states are not indicated.) A state in the model is colored black if the formula is valid in that state, and otherwise colored white. Seven formulas are considered:*

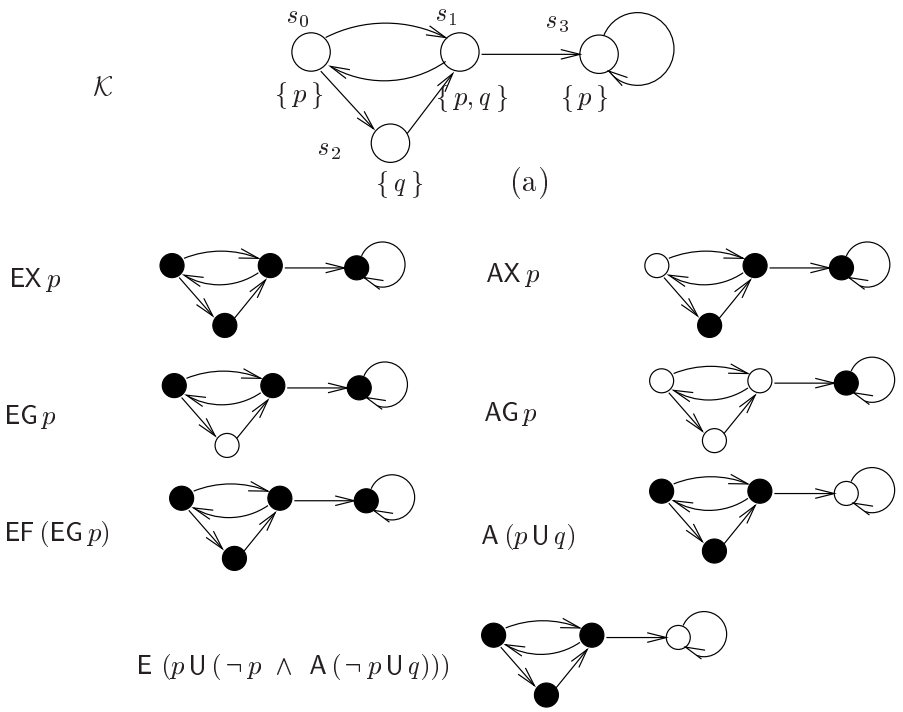


Figure 6.4: Interpretation of several CTL-formulas for a Kripke structure $\mathcal{K}$

- *The formula* $\mathsf{EX}\,p$ *is valid for all states since all states have some direct successor state that satisfies* p*.*
- $\mathsf{AX}\,p$ *is not valid for state* s_0*, since a possible path starting at* s_0 *goes directly to state* s_2 *for which* p *does not hold. Since the other states have only direct successors for which* p *holds,* $\mathsf{AX}\,p$ *is valid for all other states.*
- *For all states except state* s_2*, it is possible to have a computation that leads to state* s_3 *(such as* $s_0\,s_1\,s_3^\omega$ *when starting in* s_0*) for which* p *is globally valid. Therefore,* $\mathsf{EG}\,p$ *is valid in these states. Since* $p \notin \mathrm{Label}(s_2)$ *there is no path starting at* s_2 *for which* p *is globally valid.*
- $\mathsf{AG}\,p$ *is only valid for* s_3 *since its only path,* s_3^ω*, always visits a state in which* p *holds. For all other states it is possible to have a path which contains* s_2 *that does not satisfy* p*. So, for these states* $\mathsf{AG}\,p$ *is not valid.*
- $\mathsf{EF}\,(\mathsf{EG}\,p)$ *is valid for all states since from each state another state (either* s_0*,* s_1 *or* s_3*) can be eventually reached from which some computation can start along which* p *is globally valid.*
- $\mathsf{A}\,(p\,\mathsf{U}\,q)$ *is not valid in* s_3 *since its only computation (*s_3^ω*) never reaches a state for which* q *holds. In state* s_0 *proposition* p *holds until* q *holds, and in states* s_1 *and* s_2 *proposition* q *holds immediately. So, for these states the formula is true.*

- *Finally,* $\mathsf{E}(p\,\mathsf{U}\,(\neg p \wedge \mathsf{A}(\neg p\,\mathsf{U}\,q)))$ *is not valid in* s_3*, since from* s_3 *a* q*-state can never be reached. For the states* s_0 *and* s_1 *the formula is valid, since state* s_2 *can be reached from these states via a* p*-path,* $\neg p$ *is valid in* s_2*, and from* s_2 *all possible paths satisfy* $\neg p\,\mathsf{U}\,q$*, since* s_2 *is a* q*-state. For instance, for state* s_0 *the path* $(s_0\,s_2\,s_1)^\omega$ *satisfies* $p\,\mathsf{U}\,(\neg p \wedge \mathsf{A}(\neg p\,\mathsf{U}\,q))$ *since* $p \in \mathit{Label}(s_0)$*,* $p \notin \mathit{Label}(s_2)$ *and* $q \in \mathit{Label}(s_1)$*. For state* s_2 *the property is valid since* p *is invalid in* s_2 *and for all paths starting at* s_2 *the first state is a* q*-state.*

(End of example.)

Formulas $\mathsf{A}(\Phi\,\mathsf{U}\,\Psi)$ and $\mathsf{E}(\Phi\,\mathsf{U}\,\Psi)$ are valid if it is guaranteed that (either for all or for some paths) eventually a Ψ-state is reached. This is sometimes a rather strong requirement. A weaker variant of until, the *unless* operator W, states that Φ holds continuously either until Ψ holds for the first time, or throughout the path. This operator is defined by:

$$\mathsf{A}(\Phi\,\mathsf{W}\,\Psi) \equiv \mathsf{AG}\,\Phi \vee \mathsf{A}(\Phi\,\mathsf{U}\,\Psi)$$

or equivalently by:

$$\mathsf{A}(\Phi\,\mathsf{W}\,\Psi) \equiv \mathsf{AF}\,\neg\Phi \Rightarrow \mathsf{A}(\Phi\,\mathsf{U}\,\Psi)$$

In a similar way, the variant with an existential path quantifier can be defined. Later on, we will discuss the practical relevance of this operator.

6.4 Axiomatization

In Chapter 3 on linear temporal logic we have seen that rather than a reasoning on the basis of the semantics, axioms can be used to prove the equivalence of formulas. These axioms are defined on the syntax of formulas and proofs can thus be performed at a syntactic level. An important axiom for PLTL is the expansion axiom for until:

$$\Phi\,\mathsf{U}\,\Psi \equiv \Psi \vee (\Phi \wedge \mathsf{X}(\Phi\,\mathsf{U}\,\Psi))$$

For CTL, axioms similar to the expansion axiom exist. As the linear temporal operator U can be prefixed with either an existential or a universal path quantifier, we have axioms for $\mathsf{E}(\Phi\,\mathsf{U}\,\Psi)$ and $\mathsf{A}(\Phi\,\mathsf{U}\,\Psi)$. These axioms are listed in the last two rows of Table 6.2. The soundness of these axioms can be proved using the semantics of CTL. These proofs are similar to the proof of the expansion axiom for U for PLTL as discussed in Chapter 3, and are left to the reader. A complete axiomatization of CTL does exist [21, 68], but falls outside the scope

of this book.

$\mathsf{EF}\,\Phi$	$\equiv$	$\Phi \vee \mathsf{EX}\,(\mathsf{EF}\,\Phi)$
$\mathsf{AF}\,\Phi$	$\equiv$	$\Phi \vee \mathsf{AX}\,(\mathsf{AF}\,\Phi)$
$\mathsf{EG}\,\Phi$	$\equiv$	$\Phi \wedge \mathsf{EX}\,(\mathsf{EG}\,\Phi)$
$\mathsf{AG}\,\Phi$	$\equiv$	$\Phi \wedge \mathsf{AX}\,(\mathsf{AG}\,\Phi)$
$\mathsf{E}\,(\Phi\,\mathsf{U}\,\Psi)$	$\equiv$	$\Psi \vee (\Phi \wedge \mathsf{EX}\,\mathsf{E}\,(\Phi\,\mathsf{U}\,\Psi))$
$\mathsf{A}\,(\Phi\,\mathsf{U}\,\Psi)$	$\equiv$	$\Psi \vee (\Phi \wedge \mathsf{AX}\,\mathsf{A}\,(\Phi\,\mathsf{U}\,\Psi))$

Table 6.2: Expansion axioms for CTL

The first four axioms of Table 6.2 can be derived from the last two axioms. For example, we derive for $\mathsf{AF}\,\Phi$:

$\mathsf{AF}\,\Phi$
$\Leftrightarrow$ { definition of AF }
$\mathsf{A}\,(\mathsf{true}\,\mathsf{U}\,\Phi)$
$\Leftrightarrow$ { axiom for $\mathsf{A}\,(\Phi\,\mathsf{U}\,\Psi)$ }
$\Phi \vee (\mathsf{true} \wedge \mathsf{AX}\,(\mathsf{A}\,(\mathsf{true}\,\mathsf{U}\,\Phi)))$
$\Leftrightarrow$ { predicate calculus; definition of AF }
$\Phi \vee \mathsf{AX}\,(\mathsf{AF}\,\Phi)$

Using this result, we derive for $\mathsf{EG}\,\Phi$:

$\mathsf{EG}\,\Phi$
$\Leftrightarrow$ { definition of EG }
$\neg\,\mathsf{AF}\,\neg\,\Phi$
$\Leftrightarrow$ { result of above derivation }
$\neg\,(\neg\,\Phi \vee \mathsf{AX}\,(\mathsf{AF}\,\neg\,\Phi))$
$\Leftrightarrow$ { predicate calculus }
$\Phi \wedge \neg\,\mathsf{AX}\,(\mathsf{AF}\,\neg\,\Phi)$
$\Leftrightarrow$ { definition of AX }
$\Phi \wedge \mathsf{EX}\,(\neg\,(\mathsf{AF}\,\neg\,\Phi))$
$\Leftrightarrow$ { definition of EG }
$\Phi \wedge \mathsf{EX}\,(\mathsf{EG}\,\Phi)$

Similar derivations can be performed in order to derive the axioms for EF and AG.

The basic idea behind these axioms is to express the validity of a formula by a statement about the current state (without the need to use temporal operators) and a statement about the direct successors of this state (using either EX or AX depending on whether an existential or a universally quantified formula is treated). For instance, $\mathsf{EG}\,\Phi$ is valid in state s if Φ is valid in s (a statement about the current state) and Φ holds for all states along some path starting at s (a statement about the successor states).

The fact that for the expansion axiom for PLTL similar axioms for CTL do exist seems to suggest that any axiom for PLTL can be lifted to CTL. This is, however, not true. Consider, for example, the following statement:

$$\mathsf{F}\,(\Phi \vee \Psi) \equiv \mathsf{F}\,\Phi \vee \mathsf{F}\,\Psi$$

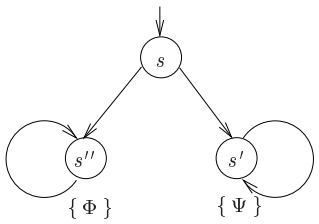
which is valid for any path. The same is true for:

$$\mathsf{EF}\,(\Phi \vee \Psi) \equiv \mathsf{EF}\,\Phi \vee \mathsf{EF}\,\Psi$$

This can be seen as follows. We first consider the implication from right to left, and then treat the reverse direction.

- $\mathsf{EF}\,\Phi \vee \mathsf{EF}\,\Psi \Rightarrow \mathsf{EF}\,(\Phi \vee \Psi)$ is a valid statement. Assume that $s \models \mathsf{EF}\,\Phi \vee \mathsf{EF}\,\Psi$. Then, without loss of generality, we may assume that $s \models \mathsf{EF}\,\Phi$. This means that there is some state s' (possibly $s = s'$), reachable from state s, such that $s' \models \Phi$. But then $s' \models \Phi \vee \Psi$. This means that there exists a reachable state from s which satisfies $\Phi \vee \Psi$. By the semantics of CTL it now follows $s \models \mathsf{EF}\,(\Phi \vee \Psi)$.
- $\mathsf{EF}\,(\Phi \vee \Psi) \Rightarrow \mathsf{EF}\,\Phi \vee \mathsf{EF}\,\Psi$ is a valid statement. Let s be an arbitrary state such that $s \models \mathsf{EF}\,(\Phi \vee \Psi)$. Then there exists a state s' (possibly $s = s'$) such that $s' \models \Phi \vee \Psi$. Without loss of generality we may assume that $s' \models \Phi$. But then we can conclude $s \models \mathsf{EF}\,\Phi$, as s' is reachable from s. Therefore we also have $s \models \mathsf{EF}\,\Phi \vee \mathsf{EF}\,\Psi$.

However, $\mathsf{AF}\,(\Phi \vee \Psi) \not\equiv \mathsf{AF}\,\Phi \vee \mathsf{AF}\,\Psi$ since $\mathsf{AF}\,(\Phi \vee \Psi) \Rightarrow \mathsf{AF}\,\Phi \vee \mathsf{AF}\,\Psi$ is invalid as shown by the following Kripke structure:



For each path that starts in state s we have that $\mathsf{F}\,(\Phi \vee \Psi)$ holds, so $s \models \mathsf{AF}\,(\Phi \vee \Psi)$. This follows directly from the fact that each path visits either

state s' or state s'' eventually, and $s' \models \Phi \vee \Psi$ and the same applies to s''. However, state s does not satisfy $\mathsf{AF}\,\Phi \vee \mathsf{AF}\,\Psi$. For instance, path $s\,(s'')^\omega \models \mathsf{F}\,\Phi$ but $s\,(s'')^\omega \not\models \mathsf{F}\,\Psi$. Thus, $s \not\models \mathsf{AF}\,\Psi$. By a similar reasoning applied to path $s\,(s')^\omega$ it follows $s \not\models \mathsf{AF}\,\Phi$. Thus, $s \not\models \mathsf{AF}\,\Phi \vee \mathsf{AF}\,\Psi$. Stated in words, it is neither true that all computations that start in state s eventually reach a Φ-state nor that they all eventually reach a Ψ-state.

6.5 Expressiveness of CTL and PLTL

One of the main differences between CTL and PLTL is their expressiveness. More precisely, there are properties that one can express in CTL, but that cannot be expressed in PLTL, and vice versa. Formally speaking, the expressiveness of CTL and PLTL is *incomparable*. An extension of CTL, called CTL*, unifies both logics. The expressiveness of CTL* comprises that of both CTL and PLTL. In order to treat the type of properties for which CTL and PLTL differ, we introduce the syntax of CTL* and give an alternative characterization of the syntax of PLTL, basically formalizing what has been used in the introduction of this chapter. This alternative syntax distinguishes between PLTL state-formulas and PLTL path-formulas, a distinction that was absent in Chapter 3. The path-formulas are the usual PLTL-formulas. A state-formula is of the form $\mathsf{A}\,\varphi$ for path-formula φ. The basic idea is that $s \models \mathsf{A}\,\varphi$ if and only if all paths that start in state s satisfy φ. Thus, a PLTL-formula like $\mathit{red}\,\mathsf{U}\,\mathit{green}$ is "translated" into the branching-time formula $\mathsf{A}\,(\mathit{red}\,\mathsf{U}\,\mathit{green})$. This implicit universal quantification over all possible computations is commonly adopted when comparing branching and linear temporal logics [69]. Moreover, this conforms to the concept that a PLTL-formula holds for a state if it holds for all computations that start in that state. Together with the syntax of CTL, the syntax definitions of PLTL (in the new style) and CTL* are summarized in Table 6.3.

CTL* is an extension of CTL as it allows path quantifiers E and A to be arbitrarily nested with linear temporal operators such as X and U. In contrast, in CTL each linear temporal operator must be immediately preceded by a path quantifier. For example, $\mathsf{A}\,\mathsf{X}\,\mathsf{X}\,p$ is a legal CTL*-formula but does not belong to CTL. The same applies to the CTL*-formulas $\mathsf{EGF}\,p$ and $\mathsf{AGF}\,p$.

Comparing the expressiveness of logics by just considering their syntax is not sufficient, though. For example, the CTL-formula $\mathsf{AF}\,\mathsf{AF}\,p$ is syntactically different from the CTL*-formula $\mathsf{AGF}\,p$, but expresses the same thing! A state satisfies $\mathsf{AGF}\,p$ if all its computations go infinitely often through a p-state. $\mathsf{AF}\,\mathsf{AF}\,p$ holds if for any computation eventually a state is reached from which always a p-state is reached, i.e., we will in any computation visit a p-state infinitely often. In general, we therefore have to consider whether for some formula Φ stated in one logic $\mathcal{L}$, say, there does not exist an equivalent formula Ψ – that

PLTL	state-formulas	$\Phi ::= \mathsf{A}\,\varphi$
	path-formulas	$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{X}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi$
CTL	state-formulas	$\Phi ::= p \mid \neg\Phi \mid \Phi \vee \Phi \mid \mathsf{E}\,\varphi \mid \mathsf{A}\,\varphi$
	path-formulas	$\varphi ::= \mathsf{X}\,\Phi \mid \Phi\,\mathsf{U}\,\Phi$
CTL*	state-formulas	$\Phi ::= p \mid \neg\Phi \mid \Phi \vee \Phi \mid \mathsf{E}\,\varphi \mid \mathsf{A}\,\varphi$
	path-formulas	$\varphi ::= \Phi \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{X}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi$

Table 6.3: Syntax of PLTL, CTL and CTL* by distinguishing state- and path-formulas

syntactically may differ from Φ – in the other logic $\mathcal{L}'$. By means of "equivalent" we intuitively mean "express the same thing". In the following definition, we define what equivalence of formulas precisely means.

Definition 6.5. (**Equivalence of formulas**)
State-formulas Φ and Ψ are *equivalent* if and only if for all Kripke structures $\mathcal{K}$ and states s:

$$s \models \Phi \text{ if and only if } s \models \Psi$$

Note that this definition requires Φ and Ψ to be defined using the same semantics $\models$ – what else could we say if they would have been defined by two distinct interpretations? We are now in a position to define formally what it means for two temporal logics to be equally expressive.

Definition 6.6. (**Comparison of expressiveness**)
Temporal logic $\mathcal{L}$ is *at least as expressive* as temporal logic $\mathcal{L}'$ if and only if for any formula $\Phi \in \mathcal{L}$ we have:

$$\exists\, \Psi \in \mathcal{L}'.\ (\forall\, \mathcal{K}.\, \forall\, s.\, (\mathcal{K}, s \models \Phi \text{ if and only if } \mathcal{K}, s \models \Psi))$$

If $\mathcal{L}$ is at least as expressive as $\mathcal{L}'$ and $\mathcal{L}'$ is at least as expressive as $\mathcal{L}$, then $\mathcal{L}'$ and $\mathcal{L}$ are said to be *equally expressive*.

If $\mathcal{L}$ is at least as expressive as $\mathcal{L}'$, but the reverse does not hold, then $\mathcal{L}$ is also called more expressive than $\mathcal{L}'$. In Figure 6.5 we depict the relationship between the three logics considered as a Venn diagram where each ellipse represents the set of formulas that can be expressed in a logic. We see that CTL*

is more expressive than both PLTL and CTL, whereas PLTL and CTL are incomparable. Below, examples of formulas are given that show the differences in expressiveness of each of the regions in Figure 6.5.

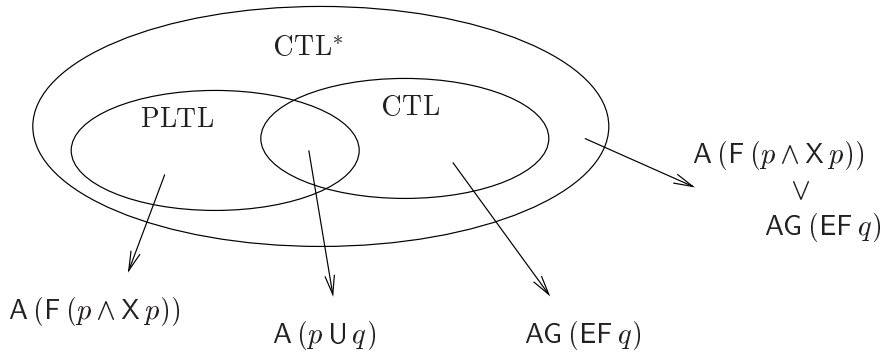


Figure 6.5: Relationship between PLTL, CTL and CTL*

In PLTL, but not in CTL. $\mathsf{A}(\mathsf{F}(p \wedge \mathsf{X}p))$ is a PLTL-formula for which there does not exist an equivalent formula in CTL. The proof of this fact can be found in [69] and falls outside the scope of this work. The formula expresses that each path eventually reaches a point at which two successive p-states are reached. The obvious CTL candidates $\mathsf{AF}(p \wedge \mathsf{EX}p)$ and $\mathsf{AF}(p \wedge \mathsf{AX}p)$ do not express the same property. The first formula is actually too weak (i.e., to liberal), whereas the second one is too strong (i.e., to restrictive).

- To see this, consider $\mathsf{AF}(p \wedge \mathsf{EX}p)$ and the Kripke structure depicted in Figure 6.6 on the left (a). We have $s_0 \models \mathsf{AF}(p \wedge \mathsf{EX}p)$, as any path starting in s_0 visits a p-state from which another p-state can be reached. In particular, this holds for the computation $(s_0\, s_1)^\omega$. On the other hand, however, $s_0 \not\models \mathsf{A}(\mathsf{F}(p \wedge \mathsf{X}p))$ as the path $(s_0\, s_1)^\omega$ never reaches a p-state that has only p-states as direct successor. Thus, $\mathsf{AF}(p \wedge \mathsf{EX}p)$ is too weak.
- Now consider $\mathsf{AF}(p \wedge \mathsf{AX}p)$ and the Kripke structure depicted in Figure 6.6 on the right (b). All paths that start in s_0 have a prefix $s_0\, s_1$ or $s_0\, s_3\, s_4$. Clearly, all such paths satisfy $\mathsf{F}(p \wedge \mathsf{X}p)$, and so, $s_0 \models \mathsf{AF}(p \wedge \mathsf{AX}p)$. On the other hand, however, $s_0 \not\models \mathsf{AF}(p \wedge \mathsf{AX}p)$ as the path $s_0\, s_1\, (s_2)^\omega$ does not satisfy $\mathsf{F}(p \wedge \mathsf{AX}p)$: state s_0 has a non p-state as direct successor. Thus, $\mathsf{AF}(p \wedge \mathsf{AX}p)$ is too strong.

Another example of a PLTL-formula for which an equivalent formulation in CTL does not exist is $\mathsf{A}(\mathsf{GF}p \Rightarrow \mathsf{F}q)$ which states that if p holds infinitely often, then q will be valid eventually. This is an interesting *fairness* property that occurs frequently in verification. For instance, a typical property for a communication protocol over an unreliable communication medium (such as a

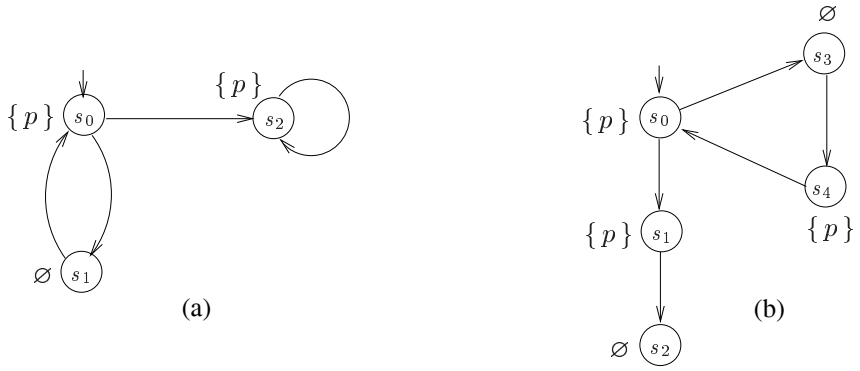
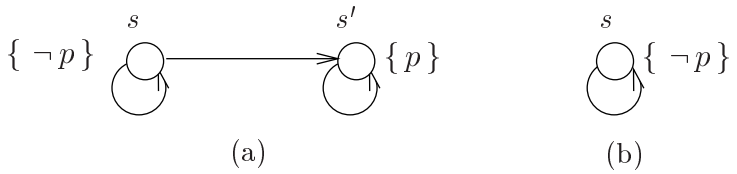


Figure 6.6: Kripke structures for $\mathsf{A}\,(\mathsf{F}\,(p \wedge \mathsf{X}\,p))$

radio or infra-red connection) is that "if a message is being sent infinitely often, it will eventually arrive at its destination".

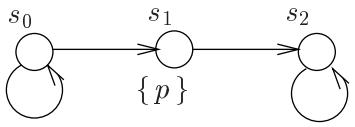
In CTL, but not in PLTL. The formula $\mathsf{AG}\,\mathsf{EF}\,p$ is a CTL-formula for which there does not exist an equivalent formulation in PLTL. The property is of use in practice since it expresses the fact that it is possible to reach a state for which p holds irrespective of the current state. If p characterizes a state where a certain error is repaired, the formula expresses the fact that it is always possible to recover from that error. In fact, we can prove that $\mathsf{AG}\,\mathsf{EF}\,p$ is not expressible in PLTL. A sketch of the proof is as follows [102]. Let Φ be a PLTL-formula such that $\mathsf{A}\,\Phi$ is equivalent to $\mathsf{AG}\,\mathsf{EF}\,p$. Since $\mathcal{K}, s \models \mathsf{AG}\,\mathsf{EF}\,p$ in the left-hand figure below (a), it follows that $\mathcal{K}, s \models \mathsf{A}\,\Phi$ because $\mathsf{A}\,\Phi$ is equivalent to $\mathsf{AG}\,\mathsf{EF}\,p$. Note that path s^ω also satisfies $\mathsf{GEF}\,p$ as state s, the only state on that path, satisfies $\mathsf{EF}\,p$. Let $\mathcal{K}'$ be the sub-model of $\mathcal{K}$ shown in the right-hand diagram (b). The paths starting from s in $\mathcal{K}'$ are also paths starting from s in $\mathcal{K}$, so we have $\mathcal{K}', s \models \mathsf{A}\,\Phi$. However, it is not the case that $\mathcal{K}', s \models \mathsf{AG}\,\mathsf{EF}\,p$, since $\mathsf{EF}\,p$ is never valid along the only path s^ω.



Relationship between CTL* and PLTL. The relationship between PLTL and CTL* can be expressed as follows [45]: any CTL*-formula Φ can be expressed in PLTL if and only if Φ is equivalent to the PLTL-formula $\mathsf{A}\,\mathit{ctl*2pltl}(\Phi)$ where $\mathit{ctl*2pltl}(\Phi)$ is obtained from Φ by eliminating all path-quantifiers. For instance, for Φ equal to $\mathsf{AG}\,\mathsf{EF}\,p$ we have $\mathit{ctl*2pltl}(\Phi) = \mathsf{G}\,\mathsf{F}$. A definition of this function can easily be given by structural induction on the syntax of CTL*, and is omitted here. Note that the result yields that $\mathsf{AG}\,\mathsf{EF}\,p$ cannot be expressed

in PLTL as $ctl^*2pltl(\Phi) = \mathsf{GF}$ is not equivalent to it.

As a final example of the difference in expressiveness of PLTL, CTL and CTL* consider the PLTL-formula $\mathsf{GF}\,p$, which says infinitely often p. It is not difficult to see that prefixing this formula with an existential or a universal path quantifier leads to a CTL*-formula: $\mathsf{A}(\mathsf{GF})\,p$ and $\mathsf{E}(\mathsf{GF})\,p$ are both CTL*-formulas. $\mathsf{AGF}\,p$ is equivalent to $\mathsf{AG\,AF}\,p$ — for any model $\mathcal{K}$ the validity of these two formulas is identical — and thus for $\mathsf{A}(\mathsf{GF})\,p$ an equivalent CTL-formula does exist, since $\mathsf{AG\,AF}\,p$ is a CTL-formula. For $\mathsf{E}(\mathsf{GF})\,p$, however, no equivalent CTL-formula does exist. This can be seen by considering the following model, where empty sets of atomic propositions are omitted.



We have $s_0 \models \mathsf{E}(\mathsf{GF})\,p$ since for any state on the path $(s_0)^\omega$ a p-state is eventually reachable. (Note that this does not mean that we finally should reach that p-state, though.) However, $s_0 \not\models \mathsf{A}(\mathsf{GF})p$ since there is no path starting in s_0 such that p is infinitely often valid.

6.6 Fairness and CTL

Recall that fairness assumptions (cf. Section 3.7) are used to rule out computations that are unreasonable for the system under consideration. These unreasonable computations are the "unfair" ones; the remaining computations are thus the "fair" ones. In model-checking PLTL one can guarantee that a formula Φ only holds for a set of fair computations by imposing an extra premise on Φ. That is to say, one adapts the formula to be checked to the desired fairness constraint, and verifies the resulting modified PLTL-formula. As there are different notions of fairness, various distinct constraints can be imposed: unconditional, weak or strong fairness. For CTL, this (direct) approach does not work, as most fairness properties cannot be expressed as CTL-formulas. Therefore, an alternative approach is taken.

To deal with fairness constraints in CTL, the semantics of CTL is slightly modified such that the state-formulas $\mathsf{A}\,\varphi$ and $\mathsf{E}\,\varphi$ are interpreted over all fair paths rather than over all possible paths. A fair path is a path that satisfies a set of fairness constraints. A fairness constraint is defined by identifying certain sets of states. For instance, for checking absence of individual starvation of a mutual exclusion algorithm such a fairness constraint could be "process one is not in its critical section". Imposing this fairness constraint on paths means that a fair path must have infinitely many states where process one is not in its

critical section.

Kripke structures are equipped with fairness constraints in the following way:

Definition 6.7. (**Fair Kripke structure**)
A *fair* Kripke structure is a quadruple $\mathcal{K} = (S, I, R, Label, \mathcal{F})$ where $(S, I, R, Label)$ is a Kripke structure and $\mathcal{F} \subseteq 2^S$ is a set of fairness constraints.

Definition 6.8. (**Fair path**)
A path $\sigma = s_0\, s_1\, s_2 \ldots$ is called *fair* if for every set of states $F_i \in \mathcal{F}$ $(0 \leqslant i < |\mathcal{F}|)$ there are infinitely many states in σ that belong to F_i.

Notice that this condition is identical to the condition for accepting runs of a generalized Büchi automaton (see Chapter 4). Indeed, a fair Kripke structure is an ordinary Kripke structure that is extended with a generalized Büchi acceptance condition. For $\mathcal{F} = \varnothing$ the requirement that every set F_i is visited infinitely often is vacuously true, and any path is fair in that case. The fair Kripke structure $(S, I, R, Label, \varnothing)$ thus has the same paths as Kripke structure $(S, I, R, Label)$.

The semantics of CTL in terms of fair Kripke structures is identical to the semantics given earlier (cf. Definition 6.4), except that all quantifications over paths are interpreted over fair paths rather than over all paths. Let $Paths_{fair}(s)$ be the set of fair paths in $\mathcal{K}$ that start in state s. Clearly, $Paths_{fair}(s) \subseteq Paths(s)$ as paths that do not visit any F_i infinitely often are ruled out, while no new paths are considered. Note that $Paths_{fair}(s) = \varnothing$ if all paths starting in s do not visit all F_i infinitely often. The fair interpretation of CTL is defined in terms of the satisfaction relation $\models_f$ (subscript f denotes fair). $(\mathcal{K}, s) \models_f \Phi$ if and only if Φ is valid in state s of fair Kripke structure $\mathcal{K}$.

Definition 6.9. (**Fair semantics of CTL**)
Let $p \in AP$ be an atomic proposition, $\mathcal{K} = (S, I, R, Label, \mathcal{F})$ be a fair Kripke structure, $s \in S$, Φ, Ψ be CTL state-formulas, and φ be a CTL path-formula. The satisfaction relation $\models_f$ is defined for state-formulas by:

$$
\begin{array}{lll}
s \models_f p & \text{iff} & Paths_{fair}(s) \neq \varnothing \wedge p \in Label(s) \\
s \models_f \neg \Phi & \text{iff} & \text{not } (s \models_f \Phi) \\
s \models_f \Phi \vee \Psi & \text{iff} & (s \models_f \Phi) \text{ or } (s \models_f \Psi) \\
s \models_f \mathsf{E}\,\varphi & \text{iff} & \exists\, \sigma \in Paths_{fair}(s).\, \sigma \models_f \varphi \\
s \models_f \mathsf{A}\,\varphi & \text{iff} & \forall\, \sigma \in Paths_{fair}(s).\, \sigma \models_f \varphi
\end{array}
$$

For fair path σ, the satisfaction relation $\models_f$ is defined for path-formulas by:

$$
\begin{array}{lll}
\sigma \models_f \mathsf{X}\,\Phi & \text{iff} & \sigma[1] \models_f \Phi \\
\sigma \models_f \Phi\,\mathsf{U}\,\Psi & \text{iff} & \exists\, j \geqslant 0.\, (\sigma[j] \models_f \Psi \;\wedge\; (\forall\, 0 \leqslant k < j.\, \sigma[k] \models_f \Phi))
\end{array}
$$

The clauses for the propositional logic terms are identical to the semantics given earlier except for atomic propositions. In order to have $s \models_f p$ it is of importance, besides the fact that s should be labeled by p, whether a fair path that starts in state s does exist. For the traditional CTL-semantics such constraint is vacuously true as for each state in a Kripke structure at least one path does exist. In case of a fair semantics this is no longer the case. For the path quantifiers E and A the difference lies in the quantifications that are over fair paths rather than over all paths. The expressiveness of CTL under a fair interpretation (let us call this *fair CTL*) is strictly larger than that of CTL, and is (like CTL) a subset of CTL*. As for CTL, the expressiveness of fair CTL is incomparable to that of PLTL.

Example 6.5. Consider the Kripke structure depicted in Figure 6.7 (i.e., $\mathcal{F} = \varnothing$) and suppose we are interested in establishing whether or not $\mathcal{K}, s_0 \models \mathsf{AG}\,(p \Rightarrow \mathsf{AF}\,q)$. This formula is invalid since the path $s_0\,s_1\,(s_2\,s_4)^\omega$ never goes through a q-state. The reason that this property is not valid is as follows. At state s_2 there is a non-deterministic choice between moving either to state s_3 or to s_4. By continuously ignoring the possibility of going to s_3 we obtain a computation for which $\mathsf{AG}\,(p \Rightarrow \mathsf{AF}\,q)$ is invalid, hence:

$$\mathcal{K}, s_0 \not\models \mathsf{AG}\,(p \Rightarrow \mathsf{AF}\,q)$$

Usually, though, the intuition is that if there is infinitely often a choice of moving to s_3 then s_3 should be visited in some fair way.

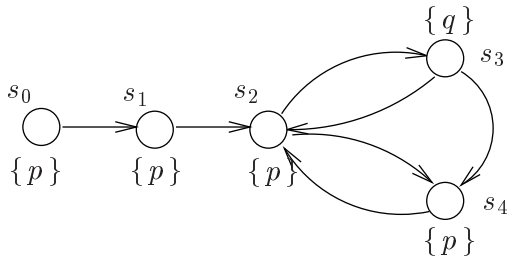


Figure 6.7: An example Kripke structure

We now modify the Kripke structure shown in Figure 6.7 by defining $\mathcal{F} = \{ F_1, F_2 \}$ where $F_1 = \{ s_3 \}$ and $F_2 = \{ s_4 \}$. The resulting model is named $\mathcal{K}'$. Let us now check $\mathsf{AG}\,(p \Rightarrow \mathsf{AF}\,q)$ on this fair model, that is, consider the verification problem $\mathcal{K}', s_0 \models_f \mathsf{AG}\,(p \Rightarrow \mathsf{AF}\,q)$. Due to the fairness constraints, any fair path has to go infinitely often through some state in F_1 and some state in F_2. This means that states s_3 and s_4 must be visited infinitely often. Paths like $s_0\,s_1\,(s_2\,s_4)^\omega$ are now excluded, since s_3 is never visited along this path. Thus we deduce that indeed:

$$\mathcal{K}', s_0 \models_f \mathsf{AG}\,(p \Rightarrow \mathsf{AF}\,q)$$

It is left to the reader to check that a fair Kripke structure $\mathcal{K}''$ with $\mathcal{F} = \{\{s_3, s_4\}\}$ does not exclude the path $s_0\, s_1\, (s_2\, s_4)^\omega$, and hence

$$\mathcal{K}'', s_0 \not\models_f \mathsf{AG}\,(p \;\Rightarrow\; \mathsf{AF}\, q)$$

(End of example.)

Whereas for PLTL fairness constraints are specified as part of the formula to be checked, for CTL similar constraints are imposed on the underlying model of the system under consideration, i.e., the Kripke structure. Using a somewhat more expressive logic like CTL* we can, however, also give a logical characterization of the kind of fairness in fair Kripke structures. This works as follows. Suppose $\Phi_1, \ldots, \Phi_n$ are the desired fairness constraints and Ψ is the state-formula to be checked for Kripke structure $\mathcal{K}$. Let $\mathcal{K}_f$ be equal to $\mathcal{K}$ with the exception that $\mathcal{K}_f$ is a fair Kripke structure where the fairness constraints $\Phi_1, \ldots, \Phi_n$ are realized by appropriate acceptance sets F_i ($0 < i \leqslant n$, i.e., each acceptance set F_i corresponds to states satisfying Φ: $F_i = \{\, s \mid s \models \Phi_i \,\}$. Then checking Ψ on the fair model $\mathcal{K}'$ amounts to check $\mathsf{A}\,((\forall i.\, \mathsf{G}\,\mathsf{F}\,\Phi_i) \;\Rightarrow\; \Psi)$ on the ordinary (unfair) model $\mathcal{K}$:

$$\mathcal{K}_f, s \models_f \Psi \text{ if and only if } \mathcal{K}, s \models \mathsf{A}\,((\forall i.\, \mathsf{G}\,\mathsf{F}\,\Phi_i) \;\Rightarrow\; \Psi)$$

Note that $\mathsf{A}\,((\forall i.\, \mathsf{G}\,\mathsf{F}\,\Phi_i) \;\Rightarrow\; \Psi)$ is indeed not a CTL-formula – otherwise we could have incorporated the fairness constraint into the formula Φ (as for PLTL) – but a CTL*-formula. The intuition is that the formulas $\mathsf{G}\,\mathsf{F}\,\Phi_i$ exactly characterize those paths that visit F_i infinitely often.

6.7 Practical use of CTL

This section discusses the practical usage of CTL as a logic to specify relevant system properties. We do so by considering the classes of properties as distinguished in Section 3.8 (and which is adopted from [22]): reachability, safety, liveness, and fairness properties.

Reachability properties express that some particular situation can be reached. Reachability properties are of the type "there exists a path such that some scenario can be reached", and can be naturally expressed in CTL by $\mathsf{EF}\,\Phi$ where Φ characterizes the set of states to be reached. A negated reachability property expresses that some undesired situation (such as a deadlock) cannot be reached. Such property can be expressed by $\neg\,\mathsf{EF}\,\Phi$, or, equivalently, by $\mathsf{AG}\,\neg\,\Phi$. These (negated) reachability properties do not impose any conditions on the paths that are traversed until reaching the desired goal states; recall that $\mathsf{EF}\,\Phi \equiv \mathsf{true}\,\mathsf{U}\,\Phi$. *Conditional* reachability properties restrict the set of paths that may be traversed until reaching any of the goal states. An example of such property

is "it is possible that the system will go down by having at least two processors being operational up to that point". Conditional reachability properties can be naturally expressed in CTL by EU, for instance, $\mathsf{E}\,((up_3 \vee up_2)\,\mathsf{U}\,down)$ expresses the aforementioned property for the TMR system. When reachability applies to *any* state, the CTL temporal operators AG (in any state of each path) and EF (simple reachability) are nested. For instance, the property "it is always possible for the system to start as new" is formalized by $\mathsf{AG}\,\mathsf{EF}\,up_3$. This can be applied also to conditional reachability properties, for which AG and EU are nested. Such properties cannot be expressed in PLTL.

Safety properties express that, under certain conditions, something (usually "bad") never occurs. The operator AG is typical for safety properties, e.g., $\mathsf{AG}\,\neg\,(P_1@cs \wedge P_2@cs)$ expresses that processes P_1 and P_2 can never occupy their critical section simultaneously. Like for PLTL, conditional safety properties can be conveniently expressed by the unless operator. For instance, the property "as long as the user does not provide a 25 cents coin, the coffee machine won't offer coffee" is expressed by $\mathsf{A}\,(\,\neg\,coffee\,\mathsf{W}\,coin)$.

Liveness properties express that, under certain conditions, something will ultimately occur. Typically, these properties are expressed in CTL by a nested combination of AG and AF. For instance, the liveness property "once red, the traffic light will become green" is expressed by $\mathsf{AG}\,(red \Rightarrow \mathsf{AF}\,green)$. The until operator is used for conditional liveness properties, since $\Phi\,\mathsf{U}\,\Psi$ is valid if Ψ is will hold eventually. As we have discussed just above, the liveness (and reachability) property $\mathsf{AG}\,\mathsf{EF}\,up_3$ cannot be expressed in PLTL.

Fairness properties express that, under certain conditions, an event will occur (or will fail to occur) infinitely often. Such properties cannot be expressed in CTL as CTL prohibits the nesting of F and G without a path quantifier in between. The only relevant fairness property that can be expressed in CTL is $\mathsf{AG}\,\mathsf{AF}\,\Phi$, but there is neither a CTL-equivalent to $\mathsf{E}\,\mathsf{F}\,\mathsf{G}\,\Phi$ nor to $\mathsf{E}\,(\mathsf{F}\,\mathsf{G}\,\Phi \wedge \mathsf{F}\,\mathsf{G}\,\Psi)$. Note that the latter formulas are legal CTL*-formulas.

Table 6.4 lists the most commonly used specification patterns for CTL, describes their property category and their empirically established importance.

6.8 Bibliographic Notes

Branching temporal logics. Various types of branching temporal logic have been proposed in the literature. They basically differ in expressiveness, i.e., the type of formulas that one can state in the logic. We mention a few important ones in increasing expressive power: Hennessy-Milner logic (HML [91]), Unified System of Branching-Time Logic [21], Computation Tree Logic (CTL [46]), Extended Computation Tree Logic (CTL* [46]), and Modal μ-Calculus [111]. The modal

pattern	*category*	*CTL-formula*	*frequency*
response	liveness	$\mathsf{AG}(\Phi \Rightarrow \mathsf{AF}\,\Psi)$	43.4 %
universality	safety	$\mathsf{AG}\,\Phi$	19.8 %
absence	negated reachability	$\neg\,\mathsf{EF}\,\Phi$	7.4 %
precedence	liveness	$\mathsf{AG}\,\mathsf{A}(\neg\,\Phi\,\mathsf{W}\,\Psi)$	4.5 %
absence		$\mathsf{AG}\,((\Phi \wedge \neg\,\Psi \wedge \mathsf{AF}\,\Psi) \Rightarrow \mathsf{A}\,(\neg\,\Phi'\,\mathsf{U}\,\Psi))$	3.2 %
absence	safety	$\mathsf{AG}(\Psi \Rightarrow \mathsf{AG}\,\neg\,\Phi)$	2.1 %
existence	liveness	$\mathsf{EF}\,\Phi$	2.1 %
			$\approx$ 80 %

Table 6.4: Most commonly used specification patterns for CTL [66]

μ-calculus is the most expressive among these languages, and HML is the least expressive. The fact that the modal μ-calculus is the most expressive logic means that for any formula Φ expressed in one of the logics mentioned, an equivalent formula Ψ in the modal μ-calculus can be given. CTL* and the μ-calculus can express fairness properties. CTL has been extended with fairness by Emerson and Halpern [69] and by Emerson and Lei [72].

Theoretical results for CTL and CTL*. Emerson [67] has shown that checking CTL satisfiability, i.e., checking whether for a given CTL-formula a model does exist, is in the complexity class EXPTIME. This means that the time complexity of checking CTL satisfiability is exponential in the length of the formula. For CTL* this problem is double exponential [70], i.e., the time complexity of checking CTL* satisfiability is double exponential in the length of the formula. A complete axiomatization of CTL has been given by Ben-Ari, Pnueli and Manna [21] and Emerson and Halpern [68].

Expressiveness of branching versus linear temporal logics. The discussion of the relative merits of linear- versus branching-time logics goes back to the early 1980s. Pnueli [152] established that linear and branching temporal logics are based on two distinct notions of time. Various papers [45, 69, 117] show that the expressiveness of PLTL and CTL are incomparable. A somewhat more practical view on comparing the usefulness of PLTL versus CTL was recently given by Vardi [181]. The logic CTL* that encompasses PLTL and CTL was defined by Clarke and Emerson [46].

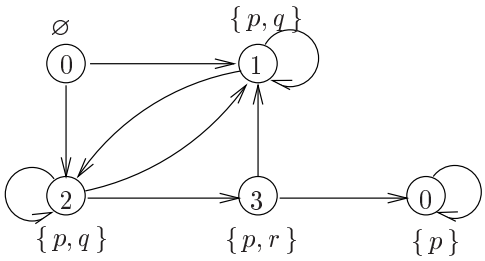
6.9 Exercises

EXERCISE 6.1. Let p and q be atomic propositions. Indicate for each of the following formulas whether or not they are CTL-formulas:

1. $\mathsf{AF}\,\mathsf{EG}\,p$

2. $\mathsf{A}\,\mathsf{EF}\,p$
3. $\mathsf{AF}\,(p\,\mathsf{U}\,\mathsf{EG}\,(p \Rightarrow q))$
4. $\mathsf{EF}\,\mathsf{AG}\,p$

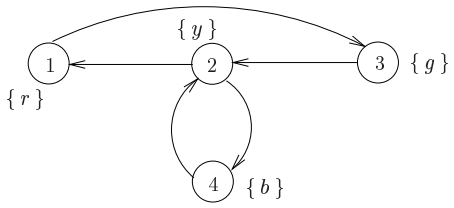
EXERCISE 6.2. Consider the following Kripke structure:



Determine for any of the following CTL-formulas which states in this Kripke structure satisfy it. Motivate your answers.

1. $\mathsf{EG}\,p$
2. $\mathsf{AG}\,p$
3. $\mathsf{EF}\,\mathsf{AG}\,p$
4. $\mathsf{EF}\,\mathsf{E}\,(p\,\mathsf{U}\,\mathsf{EG}\,(p \Rightarrow q))$

EXERCISE 6.3. Consider the following model that consists of just four states.



The following atomic propositions are used: r (red), y (yellow), g (green) and b (black). The model is intended to describe a traffic light that is able to blink yellow. You are requested to indicate for each of the following CTL-formulas the set of states for which these formulas are valid.

1. $\mathsf{AF}\,y$
2. $\mathsf{AG}\,y$
3. $\mathsf{AG}\,\mathsf{AF}\,y$
4. $\mathsf{AF}\,g$
5. $\mathsf{EF}\,g$
6. $\mathsf{EG}\,g$
7. $\mathsf{EG}\,\neg\,g$
8. $\mathsf{A}\,(b\,\mathsf{U}\,\neg\,b)$
9. $\mathsf{E}\,(b\,\mathsf{U}\,\neg\,b)$
10. $\mathsf{A}\,(\neg\,b\,\mathsf{U}\,\mathsf{EF}\,b)$
11. $\mathsf{A}\,(g\,\mathsf{U}\,\mathsf{A}\,(y\,\mathsf{U}\,r))$
12. $\mathsf{A}\,(\neg\,b\,\mathsf{U}\,b)$

EXERCISE 6.4. Prove that the following CTL-formula are satisfiable and/or valid, or give a counterexample:

1. $\mathsf{EG}\, p \;\Rightarrow\; \mathsf{AG}\, p$
2. $\mathsf{AG}\, p \;\Rightarrow\; \mathsf{EG}\, p$
3. $\mathsf{AF}\, p \vee \mathsf{AF}\, q \;\Rightarrow\; \mathsf{AF}\,(p \vee q)$
4. $\mathsf{AF}\,(p \vee q) \;\Rightarrow\; \mathsf{AF}\, p \vee \mathsf{AF}\, q$

EXERCISE 6.5. Prove that the following CTL-formula are valid or give a counterexample:

1. $\mathsf{AG}\,(r \;\Rightarrow\; (\neg q \;\wedge\; \mathsf{EX}\, r))$ if and only if $(r \;\Rightarrow\; \neg \mathsf{AF}\, q)$
2. $\mathsf{AG}\,(p \;\Rightarrow\; q)$ if and only if $(\mathsf{EX}\, p \;\Rightarrow\; \mathsf{EX}\, q)$.
3. $\mathsf{A}\,(p\,\mathsf{U}\,q) \;\Rightarrow\; \neg\,(\mathsf{E}\,(\neg q\,\mathsf{U}\,(\neg p \;\wedge\; \neg q)) \;\vee\; \mathsf{EG}\neg q)$

EXERCISE 6.6. Consider a lift system that services $N > 0$ floors numbered 0 through $N-1$. There is a lift door at each floor with a call-button and an indicator light that signals whether or not the lift has been called. In the lift cabin there are N send-buttons (one per floor) and N indicator lights that inform to which floor(s) is sent. For simplicity consider $N = 4$. Present a set of atomic propositions – try to minimize the number of propositions – that are needed to describe the following properties of the lift system as CTL-formulas and give the corresponding CTL-formulas:

1. The doors are "safe", i.e., a floor door is never open if the cabin is not present at the given floor.
2. The indicator lights correctly reflect correctly reflect the current requests. That is, each time a button is pressed, there is a corresponding request that needs to be memorized until fulfillment (if ever).
3. The cabin only services the requested floors and does not move when there is no request.
4. All requests are eventually satisfied.

EXERCISE 6.7. There are several sets of formulae that can be used as a basis to define CTL. That is to say, various sets of basic operators can be identified such that any arbitrary CTL-formula can be written in terms of these basic operators. Questions:

1. Show that CTL can be defined in terms of the basic operators EX, EU and AU, by providing a translation of any CTL-formula into these operators. (EU and AU denote the combination of a path quantifier with the until operator.)
2. Do the same for the basic operators AG, AX and AU.

EXERCISE 6.8. Give a Kripke structure that shows that the PLTL-formula $\mathsf{A\,F\,G}\, p$ (expressed as CTL*-formula) and the CTL-formula $\mathsf{AF\,AG}\, p$ are not equivalent.

EXERCISE 6.9. Let $\mathcal{K} = (S, P, Label)$, where S is a set of states, P a set of paths and $Label$ an assignment of (sets of) atomic propositions to states. Suppose we impose the following conditions on the set of paths P:

I $\quad \sigma \in P \;\Rightarrow\; \sigma^1 \in P.$

II $\quad (\rho\, s\, \sigma \in P \;\wedge\; \rho'\, s\, \sigma' \in P) \;\Rightarrow\; \rho\, s\, \sigma' \in P.$

Here σ, σ' are paths, σ^1 is σ where the first element of σ is removed, and ρ, ρ' are finite sequences of states.

Questions:

1. Give the intuitive interpretation of constraints I and II.
2. Check whether the following sets of paths satisfy I and II. Motivate your answers.
   (a) $\{\, a\, b\, c^\omega, d\, b\, e^\omega \,\}$
   (b) $\{\, a\, b\, c^\omega, d\, b\, e^\omega, b\, c^\omega, c^\omega, b\, e^\omega, e^\omega \,\}$
   (c) $\{\, a\, a^*\, b^\omega \,\}$.

Recall that a^ω denotes an infinite sequence of a's, and a^* denotes a finite (possibly empty) sequence of a's.

EXERCISE 6.10. In PLTL we have that for atomic proposition p the formulas $\mathsf{X}\,\mathsf{F}\,p$ and $\mathsf{F}\,\mathsf{X}\,p$ are equivalent. Check whether this also holds for the CTL-formulas $\mathsf{AX}\,\mathsf{AF}\,p$ and $\mathsf{AF}\,\mathsf{AX}\,p$.

EXERCISE 6.11. Does there exist an equivalent formulation of the CTL-formula $\mathsf{AF}\,\mathsf{AG}\,p$ (for some atomic proposition) p in PLTL? Give either an equivalent PLTL-formula and justify why it is equivalent, or, otherwise, provide a counter-example that shows that such equivalent formula does not exist.

EXERCISE 6.12. Consider the single pulser circuit, a hardware circuit that is part of a set of benchmark circuits for hardware verification. The single pulser has the following informal specification: "for every pulse at the input *inp* there appears exactly one pulse of length one at output *outp*, independent from the length of the input pulse". Thus, the single pulser circuit is required to generate an output pulse between two rising edges of the input signal. The following questions require the formulation of the circuit in terms of CTL. Suppose we have the proposition *rise_edge* at our disposal which is true if the input was low (0) at time instant $n-1$ and high (1) at time instant n (for natural $n > 0$). It is assumed that input sequences of the circuit are well-behaved, i.e., more that rising edge appears in the input sequence.
Questions: specify the following requirements of the circuit in CTL:

1. A rising edge at the inputs leads to an output pulse.

2. There is at most one output pulse for each rising edge.
3. There is at most one rising edge for each output pulse.

Chapter 7

Model-Checking CTL

This chapter is concerned with CTL model checking. First, the core recursive algorithm is presented that is based on a bottom-up traversal of the parse tree of Φ. The theoretical foundations of this algorithm are discussed and an efficiency improvement to the core algorithm is considered. The generation of counterexamples is considered and the required adaptations needed to model-check fair CTL are treated.

7.1 Introduction

The model-checking problem for CTL is to verify for a given Kripke structure $\mathcal{K}$, state $s \in S$, and CTL-formula Φ whether $\mathcal{K}, s \models \Phi$. That is, we need to establish whether the formula Φ is valid in state s of structure $\mathcal{K}$. In this chapter, we assume that the state space of a Kripke structure is not only denumerable (as in Chapters 3 and 6), but also finite. Thus the set S of states in $\mathcal{K}$ is finite.

The basic procedure for CTL model checking is rather straightforward:

- the set $Sat(\Phi)$ of all states satisfying Φ is computed recursively, and
- it follows that $s \models \Phi$ if and only if $s \in Sat(\Phi)$.

For a Kripke structure with set of states S we have $Sat(\Phi) = \{ s \in S \mid s \models \Phi \}$. Notice that by computing $Sat(\Phi)$ a more general problem than just checking whether $\mathcal{K}, s \models \Phi$ is solved. In fact, it checks for *any* state s in $\mathcal{K}$ whether $\mathcal{K}, s \models \Phi$, and not just for a given one. In addition, since $Sat(\Phi)$ is computed in a recursive way by considering all its sub-formulae, the sets $Sat(\Psi)$ for any sub-formula Ψ of Φ are computed, and thus $\mathcal{K}, s \models \Psi$ can be easily checked as well.

The recursive computation basically boils down to a bottom-up traversal of the *parse tree* of the formula Φ. For each node of the parse tree, i.e., for each sub-formula Ψ of Φ, the set $Sat(\Psi)$ of states is computed for which Ψ holds. This computation is carried out level-wise, starting from the leafs of the parse tree – the nodes that correspond to the atomic propositions – and finishing at the root of tree, the (only) node in the parse tree that corresponds to Φ. At an intermediate node, the results of the computions of its children are used and combined in an appropriate way to establish the states of its associated sub-formula. The type of computation at such node depends on the operator (e.g., $\wedge$, EX or EU) that is at the "top level" of the sub-formula treated. By following this bottom-up procedure, one obtains the set of states for which the requested formula Φ holds at the root of the parse tree.

Before providing a precise description of this algorithm, we first observe that it is not needed to consider all different operators of CTL. It rather suffices to consider a set of base operators such that any CTL-formula can be defined in a combination of these basic operators. An example of such set of base operators is: $\mathsf{EX}, \mathsf{EU}, \mathsf{EG}$ and the propositional operators. For convenience we also consider true and false as base operators. The following equations are essential to translate CTL-formulae in terms of these base operators:

$$\begin{aligned} \mathsf{AX}\,\Phi &\equiv \neg\,\mathsf{EX}\,\neg\,\Phi \\ \mathsf{A}\,(\Phi\,\mathsf{U}\,\Psi) &\equiv \neg\,(\mathsf{E}\,(\neg\,\Psi\,\mathsf{U}\,\neg\,(\Phi\ \vee\ \Psi))\ \vee\ \mathsf{EG}\,\neg\,\Psi) \end{aligned}$$

The validity of these rules can be proven using either the formal semantics of CTL or its axiomatisation (cf. Chapter 6) and is left as an exercise to the reader. Recall that, by definition, we have:

$$\begin{aligned} \mathsf{EF}\,\Phi &\equiv \mathsf{E}\,(\mathrm{true}\,\mathsf{U}\,\Phi) \\ \mathsf{AF}\,\Phi &\equiv \neg\,\mathsf{EG}\,\neg\,\Phi \\ \mathsf{AG}\,\Phi &\equiv \neg\,\mathsf{EF}\,\neg\,\Phi \end{aligned}$$

Using these five rules, it is not hard to show that any CTL-formula can be transformed into an equivalent formula that only contains the base operators. The following translation, denoted by f, will do:

$$\begin{aligned} f(p) &= p & f(\mathsf{EF}\,\Phi) &= \mathsf{E}\,(\mathrm{true}\,\mathsf{U}\,f(\Phi)) \\ f(\neg\,\Phi) &= \neg\,f(\Phi) & f(\mathsf{AF}\,\Phi) &= \neg\,\mathsf{EG}\,\neg\,f(\Psi) \\ f(\Phi\ \vee\ \Psi) &= f(\Phi)\ \vee\ f(\Psi) & f(\mathsf{EG}\,\Phi) &= \mathsf{EG}\,f(\Phi) \\ f(\mathsf{EX}\,\Phi) &= \mathsf{EX}\,f(\Phi) & f(\mathsf{AG}\,\Phi) &= \neg\,\mathsf{E}\,(\mathrm{true}\,\mathsf{U}\,\neg\,f(\Phi)) \\ f(\mathsf{AX}\,\Phi) &= \neg\,\mathsf{EX}\,\neg\,f(\Phi) & f(\mathsf{E}\,(\Phi\,\mathsf{U}\,\Psi)) &= \mathsf{E}\,(f(\Phi)\,\mathsf{U}\,f(\Psi)) \end{aligned}$$

$$f(\mathsf{A}\,(\Phi\,\mathsf{U}\,\Psi)) = \neg\,(\mathsf{E}\,(\neg\,f(\Psi)\,\mathsf{U}\,\neg\,(f(\Phi)\ \vee\ f(\Psi)))\ \vee\ \mathsf{EG}\,\neg\,f(\Psi))$$

It thus suffices to give a detailed description of the model-checking algorithm

for the base operators only; the verification procedure for the other operators directly follows from that.

We are now in a position to characterise the set of sub-formulae of a CTL base-formula. The set of sub-formulae of Φ is denoted by $Sub(\Phi)$ and is inductively defined as follows.

Definition 7.1. (**Sub-formulae of a CTL-formula**)
Let $p \in AP$, and Φ, Ψ be CTL-formulae. Then

$$\begin{aligned}
Sub(p) &= \{\, p \,\} \\
Sub(\neg\, \Phi) &= Sub(\Phi) \cup \{\, \neg\, \Phi \,\} \\
Sub(\Phi \vee \Psi) &= Sub(\Phi) \cup Sub(\Psi) \cup \{\, \Phi \vee \Psi \,\} \\
Sub(\mathsf{EX}\, \Phi) &= Sub(\Phi) \cup \{\, \mathsf{EX}\, \Phi \,\} \\
Sub(\mathsf{EG}\, \Phi) &= Sub(\Phi) \cup \{\, \mathsf{EG}\, \Phi \,\} \\
Sub(\mathsf{E}\, (\Phi\, \mathsf{U}\, \Psi)) &= Sub(\Phi) \cup Sub(\Psi) \cup \{\, \mathsf{E}\, (\Phi\, \mathsf{U}\, \Psi) \,\}.
\end{aligned}$$

Note that the set of sub-formulae of a state-formula is a set of state-formulas. In particular, the path-formula $\Phi\, \mathsf{U}\, \Psi$ is not a sub-formula of the state-formula $\mathsf{E}\, (\Phi\, \mathsf{U}\, \Psi)$.

If we define the length of formula Φ to be the cardinality of the set $Sub(\Phi)$, then the above sketched algorithm in terms of the parse tree of Φ can be formulated as an iterative procedure as follows. One starts with the sub-formulae of length one, i.e., the atomic propositions, true, and false. In the $(i{+}1)$-th iteration of the algorithm, sub-formulae of length $i{+}1$ are considered. The results of the previous iteration are used, e.g., it is decided that $\Phi \vee \Psi$ holds in state s, if from the previous iteration(s) it follows that either Φ or Ψ hold in s. The algorithm ends by considering the (only) sub-formula of length $\mid \Phi \mid$, i.e., the formula Φ itself.

The recursive algorithm is given in Table 7.1 and is explained in the following. The computation of $Sat(\Phi)$ is done by considering the syntactical structure of Φ. For $\Phi = \text{true}$ the program just returns S, the entire state space of $\mathcal{K}$, as true holds in any state. Accordingly, $Sat(\text{false}) = \varnothing$, since false is nowhere valid. For atomic propositions, the labelling $Label(s)$ in the Kripke structure provides all the information: $Sat(p)$ is simply the set of states that is labelled by $Label$ with p. For the negation $\neg\, \Phi$ we compute $Sat(\Phi)$ and take its complement with respect to S. Disjunction amounts to a union of sets. For $\mathsf{EX}\, \Phi$ the set $Sat(\Phi)$ is recursively computed and all states s are considered that can reach some state in $Sat(\Phi)$ by traversing a single transition. Note that for state s the set $R(s)$ denotes the set of direct successor states of s, i.e., $R(s) = \{\, s' \in S \mid (s, s') \in R \,\}$.

```
function compSat (Φ : Formula) : set of State;
(* pre: Φ is a base CTL-formula *)
begin
  switch(Φ) :
      case Φ = true then return S
      case Φ = false then return ∅
      case Φ ∈ AP then return { s | Φ ∈ Label(s) }
      case Φ = ¬ Φ1 then return S − compSat(Φ1)
      case Φ = Φ1 ∨ Φ2 then return (compSat(Φ1) ∪ compSat(Φ2))
      case Φ = EX Φ1 then return { s ∈ S | R(s) ∩ compSat(Φ1) ≠ ∅ }
      case Φ = E (Φ1 U Φ2) then return compSatEU(Φ1, Φ2)
      case Φ = EG Φ1 then return compSatEG(Φ1)
  end switch
(* post: compSat(Φ) = Sat(Φ) = { s | s ⊨ Φ } *)
end
```

Table 7.1: The recursive algorithm for model checking CTL

Some words on a possible implementation are in order. By applying the current (abstract) algorithm, sub-formulas that occur more than once in Φ are considered as many times as they occur. This may lead to a considerable increase of the verification run-time. An obvious efficiency improvement is to store the results of verifying sub-formulae (e.g., as bit-vectors) to avoid recomputations.

For $\varphi = \mathsf{E}(\Phi \,\mathsf{U}\, \Psi)$ the specific function *compSatEU*, see Table 7.2, is invoked that performs the computation of the set $Sat(\varphi)$. Recall that a state satisfies φ if there exists a path starting in s that reaches a Ψ-state by only visiting Φ-states prior to that. Intuitively, the function *compSatEU* works as follows. As each Ψ-state obviously satisfies φ, all states in $Sat(\Phi)$ are initially considered to satisfy φ. An iterative procedure is subsequently started that can be considered to systematically check the state space in a "backwards" manner. In each iteration, all Φ-states are determined that can move by a single transition to (one of) the states of which we already know to satisfy φ. Thus, in the i-th iteration of the procedure, all Φ-states are considered that can move to a Ψ-state in at most i steps. Note that if *pre* is the set of states resulting from the i-th iteration then the set after the $(i{+}1) - th$ iteration is determined by:

$$pre \cup (\{\, s \mid R(s) \cap pre \neq \varnothing \,\} \cap Sat(\Phi))$$

where $R(s)$ denotes the set of direct successor states of state s. Termination of the algorithm intuitively follows, as the number of states in the Kripke structure is finite. To avoid the re-computation of $Sat(\Phi)$ at each iteration, it is stored in an auxiliary variable, *satphi*.

Example 7.1. Consider the Kripke structure depicted in Figure 7.1, and and suppose we are interested in checking the formula $\mathsf{EF}\,\Phi$ *with* $\Phi = ((p = r) \wedge (p \neq q))$. *Recall that* $\mathsf{EF}\,\Phi = \mathsf{E}(true\,\mathsf{U}\,\Phi)$. *To check* $\mathsf{EF}\,\Phi$ *we invoke*

```
function compSatEU(Φ, Ψ : Formula) : set of State;
(* pre: Φ and Ψ are base CTL-formulae *)
begin var now, pre, satphi : set of State;
        now, pre, satphi := compSat(Ψ), ∅, compSat(Φ);
        while now ≠ pre
        do pre := now;
            now := pre ∪ ({ s | R(s) ∩ pre ≠ ∅ } ∩ satphi)
        od;
return now;
(* post: compSatEU(Φ, Ψ) = { s ∈ S | s ⊨ E(Φ U Ψ) } *)
end
```

Table 7.2: Algorithm to determine the states satisfying $\mathsf{E}(\Phi\,\mathsf{U}\,\Psi)$

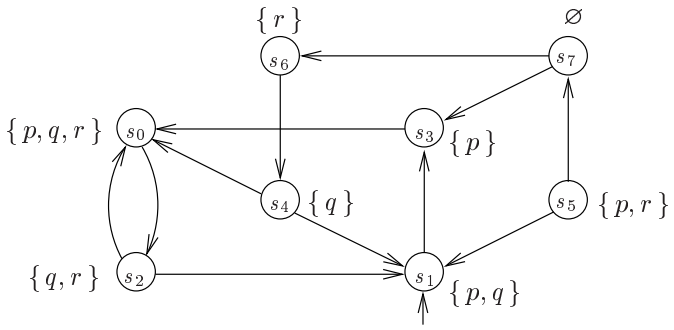


Figure 7.1: An example Kripke structure

compSatEU$(true, \Phi)$. *This algorithm recursively computes the set of states satisfying true and those for which* $(p = r) \wedge (p \neq q)$ *is valid. Accordingly,* now *is initialised to* $\{ s_4, s_5 \}$*, the only states for which* Φ *holds while* satphi *equals* S*, the set of all states. The set* pre *is initially empty. This corresponds to the situation depicted in Figure 7.2(a), where states in the set* now *are colored black, and white otherwise. In the first iteration, all states are added that have either* s_4 *or* s_5 *as a direct successor. Thus, state* s_6 *is added, cf. Figure 7.2(b). During the second iteration, the only predecessor of* s_6 *is added to* now*, yielding the snapshot in Figure 7.2(c). After the third iteration, the algorithm terminates as* now *equals* pre*, i.e., there are no further predecessors of* Φ*-states. (End of example.)*

Naively, an algorithm for checking the formula $\mathsf{EG}\,\Phi$ can be obtained by using the fact that:

$$\mathsf{EG}\,\Phi \equiv \Phi \ \wedge\ \mathsf{EX}\,(\mathsf{EG}\,\Phi)$$

and providing an algorithm that is similar in spirit to *compSatEU*. (Later in this chapter we will present a more efficient algorithm for checking $\mathsf{EG}\,\Phi$.) The thus resulting function is depicted in Table 7.3. Intuitively, the function *compSatEG* works also by a backwards traversal of the state space, but rather starting from

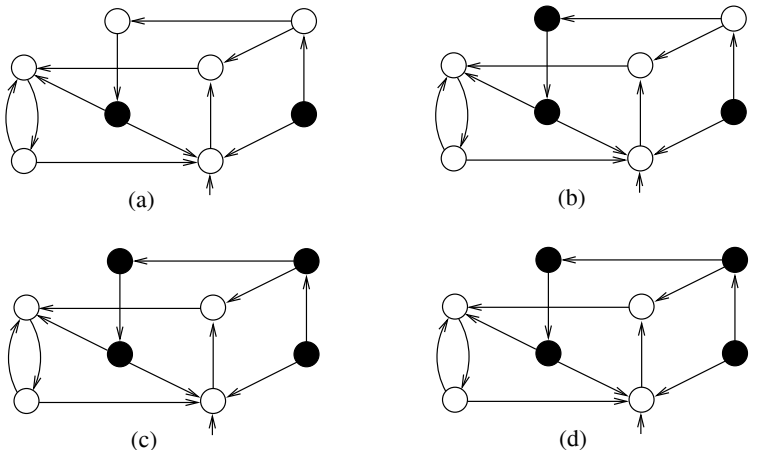


Figure 7.2: Example of running algorithm *compSatEU*(true, $(p{=}r) \wedge (p{\neq}q)$)

a small set of states (like $Sat(\Phi)$ in *compSatEU*) this function starts from the entire state space S and iteratively tries to eliminate states that refute $\mathsf{EG}\,\Phi$. In each iteration of the algorithm we only keep those states that satisfy Φ and have at least one outgoing transition to the set of states considered so far. Termination of the algorithm is again guaranteed by the finiteness of the state space.

```
function compSatEG (Φ : Formula) : set of State;
(* pre: Φ is a base CTL-formulae *)
begin var now, pre, satphi : set of State;
        now, pre, satphi := S, ∅, compSat(Φ);
        while now ≠ pre
        do pre := now;
            now := pre ∩ ({ s | R(s) ∩ pre ≠ ∅ } ∩ satphi)
        od;
return now;
(* post: compSatEG(Φ) = { s ∈ S | s ⊨ EG Φ } *)
end
```

Table 7.3: Algorithm to determine the states satisfying $\mathsf{EG}\,\Phi$

Example 7.2. Consider again the Kripke structure in Figure 7.1 and suppose the formula to be checked is $\mathsf{EG}\,q$. *To that purpose the algorithm* compSatEG(q) *is invoked. Initially, the set* now *equals* S, *the set of all states, see Figure 7.3(a). In the first iteration, the set of q-states is determined, cf. Figure 7.3(b). During the second iteration, set of q-states is computed that has at least one outgoing transition to another q-state. The (single) p-state* s_1 *that has no direct q-successor becomes "unlabeled", i.e., it is not considered any further, cf. Figure 7.3(c). By continuing this recipe, in the i-th iteration all states have been unlabeled from which no sequence of length at most i exists that only consists of q-states. At the end of the third iteration, the sets* now *and* pre *are equal and*

the computation is finished, cf. Figure 7.3(d). *(End of example.)*

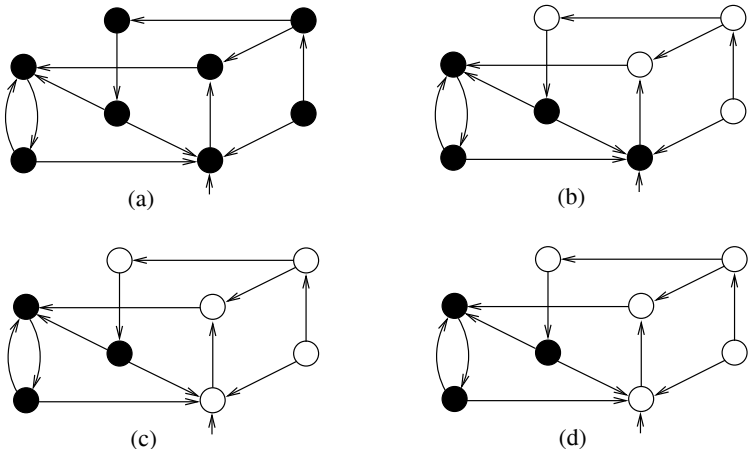


Figure 7.3: Example of running algorithm *compSatEG*(q)

Although the principle of the presented algorithms is reasonably clear and simple, establishing their correctness involves some formal *fixpoint theory* based on partial orders.

7.2 Foundations of CTL Model Checking

7.2.1 A Primer on Fixed Points

This section briefly presents some results and definitions from basic domain theory as far as they are needed to understand the fundamentals of model checking CTL. For more information on domain theory see, for instance, [83, 129]. Let A be a finite set of elements.

Definition 7.2. (**Partial order**)
A binary relation $\sqsubseteq \subseteq A \times A$ is a *partial order* if and only if, it is reflexive, anti-symmetric and transitive. That is, for all $a, a', a'' \in A$:

1. $a \sqsubseteq a$ (reflexivity)
2. $(a \sqsubseteq a' \ \wedge \ a' \sqsubseteq a) \ \Rightarrow \ a = a'$ (anti-symmetry)
3. $(a \sqsubseteq a' \ \wedge \ a' \sqsubseteq a'') \ \Rightarrow \ a \sqsubseteq a''$ (transitivity).

The pair $\langle A, \sqsubseteq \rangle$ is a partially ordered set, or shortly, *poset*. If $a \not\sqsubseteq a'$ and $a' \not\sqsubseteq a$ then a and a' are said to be incomparable. For instance, for S a set of states,

it follows that $\langle 2^S, \subseteq \rangle$, where 2^S denotes the power-set of S and $\subseteq$ the usual subset-relation, is a poset.

Definition 7.3. (**Least upper bound**)
Let $\langle A, \sqsubseteq \rangle$ be a poset and $A' \subseteq A$.

1. $a \in A$ is an *upper bound* of A' if and only if $\forall\, a' \in A' : a' \sqsubseteq a$.
2. $a \in A$ is a *least* upper bound (lub) of A', written $\sqcup A'$, if and only if
   (a) a is an upper bound of A' and
   (b) $\forall\, a'' \in A.\, a''$ is an upper bound of $A' \;\Rightarrow\; a \sqsubseteq a''$.

The concepts of the lower bound of $A' \subseteq A$, and the notion of greatest lower bound, denoted $\sqcap A'$, can be defined similarly and are omitted here. Let $\langle A, \sqsubseteq \rangle$ be a poset.

Definition 7.4. (**Complete lattice**)
$\langle A, \sqsubseteq \rangle$ is a *complete lattice* if for each $A' \subseteq A$, $\sqcup A'$ and $\sqcap A'$ do exist.

A complete lattice has a unique least element $\sqcap A = \bot$ and a unique greatest element $\sqcup A = \top$.

Example 7.3. Let $S = \{0, 1, 2\}$ and consider $\langle 2^S, \subseteq \rangle$. It is not difficult to check that for any two subsets of 2^S a least upper bound and greatest lower bound do exist. For instance, for $\{0, 1\}$ and $\{0, 2\}$ the lub is $\{0, 1, 2\}$ and the glb $\{0\}$. That is, the poset $\langle 2^S, \subseteq \rangle$ is a complete lattice where intersection and union correspond to $\sqcap$ and $\sqcup$. The least and greatest elements of this example lattice are $\varnothing$ and S, respectively. *(End of example.)*

Definition 7.5. (**Monotonic function**)
Function $F : A \longrightarrow A$ is *monotonic* if for each $a, a' \in A$ we have $a \sqsubseteq a' \;\Rightarrow\; F(a) \sqsubseteq F(a')$.

Function F is thus monotonic if it preserves the ordering $\sqsubseteq$. For instance, the function $F(S) = S \cup \{0\}$ is monotonic on $\langle 2^S, \subseteq \rangle$, as for $S', S'' \in 2^S$ we have that $F(S') \subseteq F(S'')$ if $S' \subseteq S''$.

Definition 7.6. (**Fixed point**)
For function $F : A \longrightarrow A$, $a \in A$ is called a *fixed point* of F if $F(a) = a$.

a is the *least* fixed point of F if for all $a' \in A$ such that $F(a') = a'$ we have $a \sqsubseteq a'$. The greatest fixed point of F is defined similarly.

Theorem 7.1.
Every monotonic function over a complete lattice has a complete lattice of fixed points.

Notice that the lattice of fixed points is in general different from the lattice on which the monotonic function is defined. The following result is a direct consequence of Kleene's first recursion theorem [108].

Theorem 7.2.
Every monotonic function F over a complete lattice $\langle A, \sqsubseteq \rangle$ has a unique least fixed point $\sqcup_i F^i(\bot)$ and a unique greatest fixed point $\sqcap_i F^i(\top)$. If A contains $n \geqslant 0$ elements then

$$lfp(F) = \sqcup_i F^i(\bot) = F^{n+1}(\bot) \text{ and } gfp(F) = \sqcap_i F^i(\top) = F^{n+1}(\top)$$

The least fixed point of monotonic function F on the complete lattice $\langle A, \sqsubseteq \rangle$ can thus be computed by the lub of the series $\bot, F(\bot), F(F(\bot)), \ldots$. This series is totally ordered under $\sqsubseteq$, that is, $F^i(\bot) \sqsubseteq F^{i+1}(\bot)$ for all i. This follows from the fact that $\bot \sqsubseteq F(\bot)$, since $\bot$ is the least element in the lattice, and the fact that $F(\bot) \sqsubseteq F(F(\bot))$, since F is monotonic. (In fact this second property is the key step in a proof by induction that $F^i(\bot) \sqsubseteq F^{i+1}(\bot)$.) The greatest fixed point can be computed by the glb of the series $\top, F(\top), F(F(\top)), \ldots$, a sequence that is totally ordered by $F^{i+1}(\top) \sqsubseteq F^i(\top)$ for all i.

In the sequel we are interested in monotonic functions on lattices of CTL-formulae. Such functions are of particular interest to us, since each monotonic function on a complete lattice has a unique least and greatest fixed point (cf. Theorem 7.1). These fixed points can be easily computed (cf. Theorem 7.2) and such computations form the key to the correctness of functions *compSatEU* and *compSatEG*.

7.2.2 Fixed-point Characterization of CTL

The theoretical underpinning of the functions *compSatEU* and *compSatEG* is based on a fixed point characterization of CTL-formulae. Here, the technique is to characterize $\mathsf{E}(\Phi \,\mathsf{U}\, \Psi)$ as the least (or greatest) fixed point of a function on CTL-formulae, and to apply an iterative algorithm – suggested by Theorem 7.2 – to compute such fixed points. To do this basically two main issues need to be resolved:

1. A complete lattice on CTL-formulae needs to be defined such that the existence (and uniqueness) of least and greatest fixed points is guaranteed. The basis of this lattice is a partial order relation on CTL-formulae.

2. Monotonic functions on CTL-formulae have to be determined such that $\mathsf{E}(\Phi \,\mathsf{U}\, \Psi)$ and $\mathsf{A}(\Phi \,\mathsf{U}\, \Psi)$ can be characterized as least (or greatest) fixed points of these functions. For this purpose an axiomatization of CTL is useful.

These ingredients will be treated in the rest of this section.

A Complete Lattice of CTL-formulae

The partial order $\sqsubseteq$ on CTL-formulae is defined by associating with each formula Φ the set of states in $\mathcal{K}$ for which Φ holds. Thus Φ is identified with the set

$$[\![\Phi]\!] = \{ s \in S \mid \mathcal{K}, s \models \Phi \}.$$

(Strictly speaking $[\![\ ]\!]$ is a function of $\mathcal{K}$ as well, i.e., $[\![\ ]\!]_{\mathcal{K}}$ would be a more correct notation, but since in all cases $\mathcal{K}$ is known from the context we omit this subscript.) The order $\sqsubseteq$ is now defined by:

$$\Phi \sqsubseteq \Psi \text{ if and only if } [\![\Phi]\!] \subseteq [\![\Psi]\!].$$

$\sqsubseteq$ thus corresponds to $\subseteq$, the well-known subset relation on sets. Notice that $[\![\Phi]\!] \subseteq [\![\Psi]\!]$ is equivalent to $\Phi \Rightarrow \Psi$. Clearly, $\langle 2^S, \subseteq \rangle$ is a poset, and given that for any two subsets $S_1, S_2 \in 2^S$, $S_1 \cap S_2$ and $S_1 \cup S_2$ are defined, it follows that it is a complete lattice. Here, $\cap$ is the lower bound construction and $\cup$ the upper bound construction. The least element $\bot$ in the lattice $\langle 2^S, \subseteq \rangle$ is $\varnothing$ and the greatest element $\top$ equals S, the set of all states.

Since $\sqsubseteq$ directly corresponds to $\subseteq$, it follows that the poset $\langle CTL, \sqsubseteq \rangle$ is a complete lattice. The lower bound construction in this lattice is conjunction:

$$[\![\Phi]\!] \cap [\![\Psi]\!] = [\![\Phi \ \wedge \ \Psi]\!]$$

and the upper bound corresponds to disjunction:

$$[\![\Phi]\!] \cup [\![\Psi]\!] = [\![\Phi \ \vee \ \Psi]\!].$$

Since the set of CTL-formulae is closed under conjunction and disjunction, it follows that for any Φ and Ψ their upper bound and lower bound do exist. The least element $\bot$ of the lattice $\langle CTL, \sqsubseteq \rangle$ is false, since $[\![\text{false}]\!] = \varnothing$, which is the bottom element for 2^S. Similarly, true is the greatest element in $\langle CTL, \sqsubseteq \rangle$ since $[\![\text{true}]\!] = S$.

CTL-formulae as Fixed Points

Recall from Chapter 6 (cf. Table 6.2) the expansion axiom:

$$\mathsf{E}(\Phi\,\mathsf{U}\,\Psi) \equiv \Psi \vee (\Phi \wedge \mathsf{EX}(\mathsf{E}(\Phi\,\mathsf{U}\,\Psi)))$$

for existential until-formulae. It states that $\mathsf{E}(\Phi\,\mathsf{U}\,\Psi)$ is valid if and only if Ψ is valid in the current state, or the current state satisfies Φ, and one of its next states satisfies $\mathsf{E}(\Phi\,\mathsf{U}\,\Psi)$. The recursive nature of this expansion rule suggests to consider the formula $\mathsf{E}(\Phi\,\mathsf{U}\,\Psi)$ as a fixed point of the function F that maps CTL-formulae onto CTL-formulee, defined by:

$$F(z) = \Psi \vee (\Phi \wedge \mathsf{EX}\,z)$$

(For simplicity, we do not put Ψ and Φ explicitly as parameters of F.) It is straightforward to see that $F(\mathsf{E}(\Phi\,\mathsf{U}\,\Psi))$ indeed equals $\mathsf{E}(\Phi\,\mathsf{U}\,\Psi)$ using the aforementioned expansion axiom.

The functions for the other temporal operators can be determined in a similar way. In order to explain the model-checking algorithms in a more elegant and compact way it is convenient to consider the set-theoretical counterpart of F (this approach has been adopted from [102]). More precisely, $[\![\,\mathsf{E}(\Phi\,\mathsf{U}\,\Psi)\,]\!]$ is a fixed point of the function $F_{EU} : 2^S \longrightarrow 2^S$, where F is defined by:

$$F_{EU}(Z) = [\![\,\Psi\,]\!] \cup ([\![\,\Phi\,]\!] \cap \{\, s \in S \mid R(s) \cap Z \neq \varnothing \,\}).$$

The following result shows that similar formulations can be obtained for the other temporal operators:

Theorem 7.3.

For CTL-formulae Φ and Ψ we have:

1. $[\![\,\mathsf{E}(\Phi\,\mathsf{U}\,\Psi)\,]\!]$ *is the lfp of*
$$F_{EU}(Z) = [\![\,\Psi\,]\!] \cup ([\![\,\Phi\,]\!] \cap \{\, s \in S \mid R(s) \cap Z \neq \varnothing \,\})$$
2. $[\![\,\mathsf{A}(\Phi\,\mathsf{U}\,\Psi)\,]\!]$ *is the lfp of* $F_{AU}(Z) = [\![\,\Psi\,]\!] \cup ([\![\,\Phi\,]\!] \cap \{\, s \in S \mid R(s) \subseteq Z \,\})$
3. $[\![\,\mathsf{EG}\,\Phi\,]\!]$ *is the gfp of* $F_{EG}(Z) = [\![\,\Phi\,]\!] \cap \{\, s \in S \mid R(s) \cap Z \neq \varnothing \,\}$
4. $[\![\,\mathsf{AG}\,\Phi\,]\!]$ *is the gfp of* $F_{AG}(Z) = [\![\,\Phi\,]\!] \cap \{\, s \in S \mid R(s) \subseteq Z \,\}$
5. $[\![\,\mathsf{EF}\,\Phi\,]\!]$ *is the lfp of* $F_{EF}(Z) = [\![\,\Phi\,]\!] \cup \{\, s \in S \mid R(s) \cap Z \neq \varnothing \,\}$
6. $[\![\,\mathsf{AF}\,\Phi\,]\!]$ *is the lfp of* $F_{AF}(Z) = [\![\,\Phi\,]\!] \cup \{\, s \in S \mid R(s) \subseteq Z \,\}$

It is not difficult to check using the axioms of Table 6.2 (see page 131) that for each case the CTL-formula is indeed a fixed point of the function indicated.

To determine whether it is the least or greatest fixed point is slightly more involved. Both proof obligations will be dealt with in the following.

Proof of a Fixed-Point Characterization

We illustrate the proof of Theorem 7.3 by checking the case for $[\![\mathsf{E}(\Phi \mathsf{U} \Psi)]\!]$; the proofs for the other cases are conducted in a similar way and are left to the interested reader. The proof consists of two parts: we first prove that the function $F_{EU}(Z)$ is monotonic on $\langle 2^S, \subseteq \rangle$, and then we prove that $[\![\mathsf{E}(\Phi \mathsf{U} \Psi)]\!]$ is the least fixed point of F_{EU}.

Lemma 7.1. Function $F_{EU}(Z)$ is monotonic on $\langle 2^S, \subseteq \rangle$.

Proof: Let $Z_1, Z_2 \in 2^S$ such that $Z_1 \subseteq Z_2$. It must be proven that $F_{EU}(Z_1) \subseteq F_{EU}(Z_2)$. We derive:

$$
\begin{array}{ll}
 & F_{EU}(Z_1) \\
= & \quad \{ \text{ definition of } F_{EU} \ \} \\
 & [\![\Psi]\!] \cup ([\![\Phi]\!] \cap \{ s \in S \mid R(s) \cap Z_1 \neq \varnothing \}) \\
\subseteq & \quad \{ \ Z_1 \subseteq Z_2 \text{; set calculus } \} \\
 & [\![\Psi]\!] \cup ([\![\Phi]\!] \cap \{ s \in S \mid R(s) \cap Z_2 \neq \varnothing \}) \\
= & \quad \{ \text{ definition of } F_{EU} \ \} \\
 & F_{EU}(Z_2). \qquad \textit{qed.}
\end{array}
$$

This means that for any arbitrary Ψ and Φ we have that $F_{EU}(Z_1) \subseteq F_{EU}(Z_2)$, and thus F_{EU} is monotonic on 2^S. Given this result and the fact that $\langle 2^S, \subseteq \rangle$ is a complete lattice, it follows by Theorem 7.1 that F_{EU} has a complete lattice of fixed points, including a unique least and greatest fixed point.

Lemma 7.2. $[\![\mathsf{E}(\Phi \mathsf{U} \Psi)]\!]$ is the least fixed point of F_{EU}.

Proof: Recall that the least element of $\langle 2^S, \subseteq \rangle$ is $\varnothing$. From Theorem 7.2 and the monotonicity of F_{EU}, it follows that the lfp of F_{EU} equals $F_{EU}^{n+1}(\varnothing)$ when S consists of $n+1$ states. We now prove that

$$[\![\mathsf{E}(\Phi \mathsf{U} \Psi)]\!] = F_{EU}^{n+1}(\varnothing) \text{ by induction on } n.$$

By definition we have $F_{EU}^{0}(\varnothing) = \varnothing$. For $F_{EU}(\varnothing)$ we derive:

$$
\begin{array}{ll}
 & F_{EU}(\varnothing) \\
= & \quad \{ \text{ definition of } F_{EU} \ \} \\
 & [\![\Psi]\!] \cup ([\![\Phi]\!] \cap \{ s \in S \mid R(s) \cap \varnothing \neq \varnothing \})
\end{array}
$$

$$
\begin{aligned}
&= \quad \{ \text{ calculus } \} \\
&\quad [\![\Psi]\!] \cup ([\![\Phi]\!] \cap \varnothing) \\
&= \quad \{ \text{ calculus } \} \\
&\quad [\![\Psi]\!].
\end{aligned}
$$

Thus $F_{EU}(\varnothing) = [\![\Psi]\!]$, the set of states that can reach $[\![\Psi]\!]$ in 0 steps. Now

$$
\begin{aligned}
&\quad F^2_{EU}(\varnothing) \\
&= \quad \{ \text{ definition of } F_{EU} \ \} \\
&\quad [\![\Psi]\!] \cup ([\![\Phi]\!] \cap \{ s \in S \mid R(s) \cap F_{EU}(\varnothing) \neq \varnothing \}) \\
&= \quad \{ \ F_{EU}(\varnothing) = [\![\Psi]\!] \ \} \\
&\quad [\![\Psi]\!] \cup ([\![\Phi]\!] \cap \{ s \in S \mid R(s) \cap [\![\Psi]\!] \neq \varnothing \}).
\end{aligned}
$$

Thus $F_{EU}(F_{EU}(\varnothing))$ is the set of states that can reach $[\![\Psi]\!]$ along a path of length at most one (while traversing $[\![\Phi]\!]$). By mathematical induction it can be proven that $F^{k+1}_{EU}(\varnothing)$ is the set of states that can reach $[\![\Psi]\!]$ along a path through $[\![\Phi]\!]$ of length at most k. But since this holds for any k we have:

$$
\begin{aligned}
&\quad [\![\mathsf{E}\,(\Phi \,\mathsf{U}\, \Psi)]\!] \\
&= \quad \{ \text{ by the above reasoning } \} \\
&\quad \textstyle\bigcup_{k \geqslant 0} F^{k+1}_{EU}(\varnothing) \\
&= \quad \{ \ F^i_{EU}(\varnothing) \subseteq F^{i+1}_{EU}(\varnothing) \ \} \\
&\quad F^{n+1}_{EU}(\varnothing). \qquad \textit{qed.}
\end{aligned}
$$

Verifying EU and EG

What do these results mean for CTL model checking? We discuss this by means of an example. Consider $\varphi = \mathsf{E}\,(\Phi \,\mathsf{U}\, \Psi)$. Since $[\![\mathsf{E}\,(\Phi \,\mathsf{U}\, \Psi)]\!]$ is the lfp of

$$F_{EU}(Z) = [\![\Psi]\!] \cup ([\![\Phi]\!] \cap \{ s \in S \mid R(s) \cap Z \neq \varnothing \})$$

the problem of computing $Sat(\varphi)$ boils down to computing the fixed point of F_{EU}, which is equal to $\bigcup_i F^i_{EU}(\varnothing)$. $\bigcup_i F^i_{EU}(\varnothing)$ is computed by means of iteration: $\varnothing$, $F_{EU}(\varnothing)$, $F_{EU}(F_{EU}(\varnothing)), \ldots$. This computation will terminate as, by Theorem 7.2, there exists some k such that $F^{k+1}_{EU}(\varnothing) = F^k_{EU}(\varnothing)$. Then, $F^{k+1}_{EU}(\varnothing) = \bigcup_i F^i_{EU}(\varnothing)$.

Intuitively, this iterative procedure can be understood as follows: one starts with no state. This corresponds to the approximation for which the formula is nowhere valid, i.e. $F^0_{EU}(\varnothing) = \varnothing$. Then we consider $F^1_{EU}(\varnothing)$ which reduces to $[\![\Psi]\!]$ (see proof above), and consider all states for which Ψ holds. In the next iteration we consider $F^2_{EU}(\varnothing)$ and continue in this way until the fixed point

$F^k_{EU}(\varnothing)$ is reached for some k. This is precisely what the function *compSatEU* does. At the beginning of the $(i{+}1)$-th iteration we have as an invariant $now = F^{i+1}_{EU}(\varnothing)$ and $pre = F^i_{EU}(\varnothing)$. The iterations end when $now = pre$, that is, when $F^{i+1}_{EU}(\varnothing) = F^i_{EU}(\varnothing)$.

For $\mathsf{EG}\,\Phi$, which – in contrast to $\mathsf{E}(\Phi\,\mathsf{U}\,\Psi)$ – is a *greatest* fixed point, the procedure is slightly different. Recall that

$$F_{EG}(Z) = [\![\,\Phi\,]\!] \cap \{\, s \in S \mid R(s) \cap Z \neq \varnothing \,\}$$

Greatest fixed points are equal to $\bigcap_i F^i_{EG}(S)$, which is computed iteratively by $S, F_{EG}(S), F_{EG}(F_{EG}(S)), \ldots$ where $S \supseteq F_{EG}(S) \supseteq F_{EG}(F_{EG}(S)) \supseteq \ldots$. Intuitively, this means that one starts by considering all states and then successively shrinks the set of states of interest in an iterative manner until the required set $[\![\,\mathsf{EG}\,\Phi\,]\!]$ is obtained. The initial step corresponds to setting $F^0_{EG}(S) = S$. We then consider $F^1_{EG}(S)$ which equals

$$[\![\,\Phi\,]\!] \cap \{\, s \in S \mid R(s) \cap F^0_{EG}(S) \neq \varnothing \,\}$$

that is, $[\![\,\Phi\,]\!]$. In the next iteration, $F^2_{EG}(S)$ is determined, i.e., the set of states from which a path consisting of Ψ-states of length at most one does exist:

$$[\![\,\Phi\,]\!] \cap \{\, s \in S \mid R(s) \cap [\![\,\Phi\,]\!] \neq \varnothing \,\}$$

The set $F^i_{EG}(S)$ characterises the set of states from which a path consisting of Φ-states of length at most i does exist. This process continues until the unique fixed point $[\![\,\mathsf{EG}\,\Phi\,]\!]$ is reached. This is precisely what the function *compSatEG* does.

Example 7.4. Consider again the Kripke structure of Figure 7.1. We illustrate the computation of $[\![\,\mathsf{EF}\,((p = r) \wedge (p \neq q))\,]\!]$. *This computation is identical to compSatEU except that* $\mathsf{Sat}(\Phi)$ *equals the entire state space* S. *The series* $\varnothing$, $F_{EF}(\varnothing)$, $F_{EF}(F_{EF}(\varnothing))$, ... *is computed until a fixed point is reached. According to Theorem 7.3:*

$$F^{i+1}_{EF}(Z) = [\![\,(p = r) \wedge (p \neq q)\,]\!] \cup \{\, s \in S \mid R(s) \cap F^i_{EF}(Z) \neq \varnothing \,\}.$$

We start the computation by:

$$F_{EF}(\varnothing) = [\![\,(p = r) \wedge (p \neq q)\,]\!] \cup \{\, s \in S \mid R(s) \cap \varnothing \neq \varnothing \,\} = \{\, s_4, s_5 \,\}$$

This corresponds to the situation just before starting the first iteration in the algorithm compSatEU, cf. Figure 7.2(a). For the second iteration we obtain

$$\begin{array}{ll} & F_{EF}(F_{EF}(\varnothing)) \\ = & \{\ \textit{definition of } F_{EF}\ \} \end{array}$$

$$
\begin{aligned}
& [\![(p = r) \wedge (p \neq q)]\!] \cup \{ s \in S \mid R(s) \cap F_{EF}(\varnothing) \neq \varnothing \} \\
= \quad & \{ F_{EF}(\varnothing) = \{ s_4, s_5 \} = [\![(p = r) \wedge (p \neq q)]\!] \} \\
& \{ s_4, s_5 \} \cup \{ s \in S \mid R(s) \cap \{ s_4, s_5 \} \neq \varnothing \} \\
= \quad & \{ s_4 \text{ has predecessor } \{ s_6 \}; \; s_5 \text{ has no predecessors} \} \\
& \{ s_4, s_5 \} \cup \{ s_6 \} \\
= \quad & \{ \text{set theory} \} \\
& \{ s_4, s_5, s_6 \}.
\end{aligned}
$$

The states that are now colored are those states from which a state satisfying $(p = r) \wedge (p \neq q)$ *can be reached via a path of length at most one, cf. Figure 7.2(b).*

This result and the fact that the direct predecessors of $\{ s_4, s_5, s_6 \}$ *are* $\{ s_6, s_7 \}$*, yields for the next iteration:*

$$F^3_{EF}(\varnothing) = \{ s_4, s_5 \} \cup \{ s_6, s_7 \} = \{ s_4, s_5, s_6, s_7 \}$$

Intuitively, the state s_7 *is considered in the second iteration since it can reach a state satisfying* $(p = r) \wedge (p \neq q)$ *via a path of length two, cf. Figure 7.2(c).*

Since there are no other states in $\mathcal{K}$ *that can reach such a state, the computation is finished. This can be checked formally by computing* $F^4_{EF}(\varnothing)$*. The interested reader can check that indeed* $F^4_{EF}(\varnothing) = F^3_{EF}(\varnothing)$*, that is, a fixed point has been reached, cf. Figure 7.2(d).* *(End of example.)*

7.3 On Efficiency

The time complexity of model checking CTL is determined as follows. Assume that we have a Kripke structure $\mathcal{K} = (S, R, L)$ with N states and M transitions, i.e., $|S| = N$ and $|R| = M$. Note that M equals N^2 in the worst case when each state has a transition to any other state.

From the simple recursive structure of the main algorithm *compSat* it follows that *Sat* is computed for each sub-formula of Φ, i.e., $|Sub(\Phi)|$ times. As the size of $Sub(\Phi)$ is defined as the length of Φ, the worst-case time-complexity of *compSat* is linear in the size of the CTL-formula Φ. The time complexity of the algorithm *compSatEU* is determined as follows. N iterations are needed in worst case, adding a single state to the set *now* per iteration. To determine the new value of *now*, all transitions in R are considered in each iteration. The time per iteration is thus $\mathcal{O}(M+N)$. For the algorithm *compSatEU* a similar

reasoning applies. To summarise, the worst-case time-complexity of *compSat* is

$$\mathcal{O}\left(\mid \Phi \mid \cdot N \cdot (N + M)\right)$$

Stated in words, our first algorithm for model-checking a CTL-formula is quadratic in the size of the Kripke structure.

7.3.1 An Efficiency Improvement

This time complexity can be reduced by a factor N by following some more direct, and efficient, procedures for checking EG and EU. Let us first consider *compSatEU*. Our first obervation is that it suffices to consider in each iteration only those states that are added to *pre* in the previous iteration, and consider their direct predecessors. By doing so, all incoming transitions of all states are checked exactly once. In order to make this more explicit, let us add a variable *diff* to the algorithm *compSatEU* of Table 7.2 that keeps track of the set-difference between *now* and *pre*. This yields the algorithm depicted in Table 7.4. It is not difficult to see that this algorithm computes exactly $Sat(\mathsf{E}\,(\Phi\,\mathsf{U}\,\Psi))$. By this modification it is prevented to consider the predecessors of a state more than once. This corresponds to a "backward" breadth-first search of the Kripke structure under consideration.

```
function compSatEU_eff (Φ, Ψ : Formula) : set of State;
(* pre: Φ and Ψ are base CTL-formulae *)
begin var diff, now, pre, satphi : set of State;
        now, pre, satphi := compSat(Ψ), ∅, compSat(Φ);
        diff := now − pre;
        while diff ≠ ∅
        do pre := now;
            now := pre ∪ ({ s | R(s) ∩ diff ≠ ∅ } ∩ satphi);
            diff := now − pre;
        od;
return now;
(* post: compSatEU_eff(Φ, Ψ) = { s ∈ S | s ⊨ E (Φ U Ψ) } *)
end
```

Table 7.4: Revised algorithm to determine the states satisfying $\mathsf{E}\,(\Phi\,\mathsf{U}\,\Psi)$

Secondly, an alternative, more direct approach is taken for *compSatEG*. The concept behind this variant is to check $\mathsf{EG}\,\Phi$ in state s of $\mathcal{K} = (S, R, L)$ as follows:

1. First, consider only states that satisfy Φ, eliminate all other states, and delete all transitions to these eliminated states. This yields a subgraph of $\mathcal{K}$ only consisting of Φ-states: $\mathcal{K}$ is reduced to $\mathcal{K}[\Phi] = (S', R', I', L')$ with

$S' = Sat(\Phi)$, $R' = R \cap (S' \times S')$, $I' = I \cap S'$, and $L'(s) = L(s)$ for all states $s \in S'$, and undefined otherwise. The justification of this step is that all removed states will not satisfy $\mathsf{EG}\,\Phi$ – as they do not satisfy Φ themselves – and thus can safely be ignored.

2. Then, determine all non-trivial strongly connected components (SCCs) [1] in the graph induced by $\mathcal{K}[\Phi]$, i.e., the directed graph with set of vertices S' and edges R'. All states in each of such SCC satisfy $\mathsf{EG}\,\Phi$, as any state is reachable from any other state, and – by construction – all states satisfy Φ.

3. Finally, check whether there is such non-trivial SCC that is reachable from state s in $\mathcal{K}[\Phi]$. If state s belongs to the reduced model and there exists such a path then – by construction of $\mathcal{K}[\Phi]$ – the property $\mathsf{EG}\,\Phi$ is satisfied; otherwise, it is not. This search can be done in a backward manner.

The resulting algorithm is presented in Table 7.5. Here, the algorithm to compute $\mathcal{K}[\Phi]$ is left implicit; it is left to the reader to give an algorithm that computes this structure with a time complexity that is linear in the size of $\mathcal{K}$. The algorithm *determineSCC* computes the set of non-trivial SCCs of the graph induced by $\mathcal{K}[\Phi]$. This rather standard graph algorithm is based on a depth-first search and is omitted here (see, for instance [58]). It runs in $\mathcal{O}(N + M)$. In the outermost iteration a backward search is performed starting from the states in $Sat(\Phi)$. In the innermost iteration all Φ-states that are predecessors of some selected state $s \in now$ are added to *now* and *sateg* provided they have not found before. The algorithm terminates if *now* is empty, indicating that the backward search has visited all states.

Example 7.5. Again consider the Kripke structure of Figure 7.1 and formula $\mathsf{EG}\,q$. *Its reduction* $\mathcal{K}[q]$ *consists of the four states that are labeled with q, cf. the gray states in Figure 7.4(a). The only non-trivial SCC of this structure is indicated by the black states, i.e., the states in the set* sateg, *in Figure 7.4(b). In the first outermost iteration state* s_0 *(the uppermost black state) is considered. Accordingly, its q-predecessor* s_4 *is added to* now *and* sateg. *As the remaining states in* now, *state* s_2 *and* s_4, *have no q-predecessors, in the subsequent iterations no new states are added to* now *and the computation finishes, cf. Figure 7.4(c).* *(End of example.)*

The correctness of the above algorithm is based on the following result:

[1] A strongly connected component (SCC) of a directed graph G is a maximal, connected subgraph of G. Stated differently, the SCCs of a graph are the equivalence classes of vertices under the "are mutually reachable" relation. A non-trivial SCC is an SCC that contains at least one transition.

```
function compSatEG_eff (Φ : Formula) : set of State;
(* pre: Φ is a base CTL-formula *)
begin var now, sateg, satphi : set of State;
          S' : set of (set of State);
      satphi := compSat(Φ);
      for each S' ∈ determineSCC(K[Φ])
      do now, sateg := now ∪ S', sateg ∪ S'; od;
      while now ≠ ∅
      do choose s ∈ now;
          now := now − { s };
          for each s' ∈ satphi ∩ R^{-1}(s)
          do if s' ∉ sateg
              then now, sateg := now ∪ { s' }, sateg ∪ { s' };
          od;
      od;
return sateg;
(* post: compSatEG_eff(Φ) = { s ∈ S | s ⊨ EG Φ } *)
end
```

Table 7.5: Revised algorithm to determine the states satisfying $\mathsf{EG}\,\Phi$

Theorem 7.4.
For state s in Kripke structure $\mathcal{K}$ and CTL-formula Φ we have: $s \models \mathsf{EG}\,\Phi$ if and only if $s \in \mathcal{K}[\Phi]$ and there exists a non-trivial SCC in $\mathcal{K}[\Phi]$ reachable from s.

Proof: "if": suppose $s \models \mathsf{EG}\,\Phi$. Clearly, $s \in \mathcal{K}[\Phi]$, as $s \models \mathsf{EG}\,\Phi$ implies $s \models \Phi$. Let σ be a path in $\mathcal{K}$ that starts in s such that Φ holds along σ; σ is thus also a path in $\mathcal{K}[\Phi]$. As $\mathcal{K}$ is finite, σ has a suffix $\rho = s_1, \ldots s_k$ for $k > 1$, representing a cycle that is traversed infinitely often. As σ is also a path in $\mathcal{K}[\Phi]$, the states s_1 through s_k are all in $\mathcal{K}[\Phi]$. Since ρ is traversed infinitely often, it represents a cycle, and thus, any pair of states s_i and s_j is mutually reachable. Stated differently, $\{ s_1, \ldots, s_k \}$ is either an SCC or contained in some SCC in $\mathcal{K}[\Phi]$. As σ is a path starting in s, these states are reachable from s.

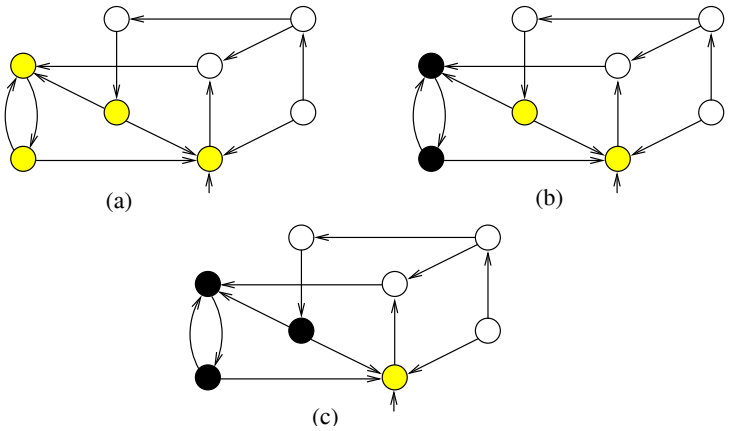


Figure 7.4: Example of running algorithm $compSatEG_{eff}(q)$

"only if": suppose $s \in \mathcal{K}[\Phi]$ and there exists an SCC in $\mathcal{K}[\Phi]$ reachable from s. Let s' be a state in such SCC. As the SCC is non-trivial, s' is reachable from itself by a sequence of length at least one. Let ρ be such path. By construction $\rho \models \mathsf{G}\,\Phi$. The path from s to s' followed by ρ now satisfies $\mathsf{G}\,\Phi$ and starts in s. Thus, $s \models \mathsf{EG}\,\Phi$. *qed.*

7.3.2 Time Complexity

The time complexity of the revised algorithms is in $\mathcal{O}(N+M)$. Thus, we conclude that:

Theorem 7.5.
For Kripke structure $\mathcal{K} = (S, I, R, Label)$ with $|\,S\,| = N$ and $|\,R\,| = M$, and CTL state-formula Φ, the worst-case time-complexity of the model checking algorithm compSat$_{eff}$ is $\mathcal{O}\,(|\,\Phi\,|\cdot(N+M))$.

So, the time complexity of model-checking CTL is linear in the length of the formula and linear in the size (i.e., number of states and transitions) of the Kripke structure.

Let us compare this complexity bound with PLTL model checking, cf. Chapter 5. Recall that model checking PLTL is exponential in the size of the formula. Although the difference in time complexity with respect to the length of the formula seems drastic, a few remarks on this are in order. First, formulae in PLTL are never longer than, and mostly shorter than, their equivalent formulation in CTL. This follows directly from the fact that for formulae that can be translated from CTL into PLTL, the PLTL-equivalent formula is obtained by removing all path quantifiers, and as a result is (usually) shorter (see Chapter 6). In fact, for each Kripke structure $\mathcal{K}$ there does exist a PLTL-formula Φ such that any CTL-formula – if any – equivalent to $\mathsf{E}\,\Phi$ (or $\mathsf{A}\,\Phi$) has exponential length! This is nicely illustrated by the following example, which we adopted from [113].

In summary, for a property that can be specified in both CTL and PLTL, the shortest possible formulation in PLTL is never longer than the CTL-formula, and can even be exponentially shorter. Thus, the "advantage" that CTL model checking is linear in the length of the formula, whereas PLTL model checking is exponential in the length, is diminished (or even completely eliminated) by the fact that a given property needs a (much) longer formulation in CTL than in PLTL.

Example 7.6. Consider the NP-complete problem of finding a Hamiltonian path in an arbitrary, connected, directed graph $G = (V, E)$ where V denotes the set of vertices and $E \subseteq V \times V$, the set of edges. Let $V = \{\, v_1, \ldots, v_n \,\}$.

A Hamiltonian path is a (finite) path through the graph which visits each state exactly once and returns in the initial state. (It is a travelling salesman problem where the cost of traversing an edge is the same for all edges.)

We first describe the Hamiltonian path problem in PLTL. To that purpose, graph G is transformed into Kripke structure $\mathcal{K}_G = (S, I, R, \mathrm{Label})$ with for $0 < i \leqslant n$ and atomic propositions p_i and q:

- *state space $S = V \cup \{ w \}$, for $w \notin V$*
- *initial states $I = V$*
- *transitions $R = E \cup (S \times \{ w \})$*
- *labelling $\mathrm{Label}(v_i) = \{ p_i \}$ and $\mathrm{Label}(w) = \{ q \}$.*

All vertices of G are states in the Kripke structure. A new state w is introduced, uniquely identified by atomic proposition q, that is a direct successor of any state (including w itself). Figure 7.5 shows this construction for (a) an example directed graph; the resulting Kripke structure is depicted in Figure 7.5(b).

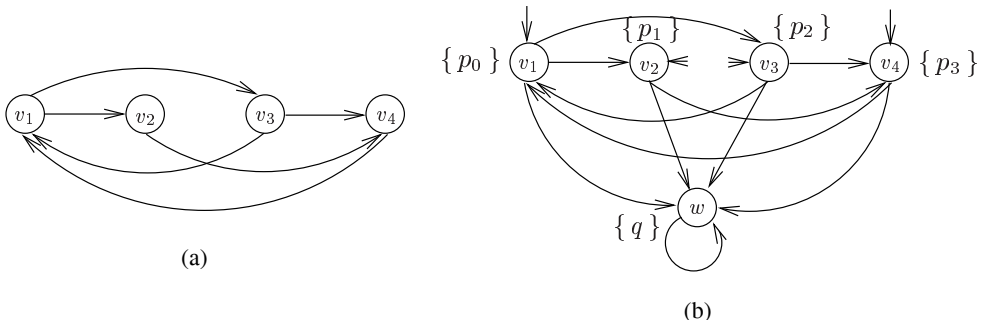


Figure 7.5: Encoding the Hamiltonian path problem in a Kripke structure

The existence of a Hamiltonian path in a graph can now, for instance, be formulated in PLTL as follows[2]*:*

$$\mathsf{E}\,((\mathsf{F}\,p_1 \wedge \ldots \wedge \mathsf{F}\,p_n) \;\wedge\; \mathsf{X}^n q)\,.$$

where $\mathsf{X}^0 q = q$ and $X^{k+1} q = \mathsf{X}\,(\mathsf{X}^k q)$ for $k \geqslant 0$. This formula is valid in each state from which a path starts that fulfills each atomic proposition once. This corresponds to visiting each state v_i once (and w infinitely often). Note that each path satisfying $(\mathsf{F}\,p_0 \wedge \ldots \wedge \mathsf{F}\,p_n) \wedge \mathsf{X}^n q$ has a suffix w^ω, by construction

[2]This is not a well-formed PLTL-formula since the path quantifier E is not part of PLTL. However, for $\mathsf{E}\,\Phi$, where Φ does not contain any path quantifiers (as in this example) we can take the equivalent $\neg\,\mathsf{A}\,\neg\,\Phi$ where $\mathsf{A}\,\neg\,\Phi$ is a well-formed PLTL-formula according to Table 6.3.

of $\mathcal{K}_G$. The length of the above formula is linear in the number of vertices in the graph. Graph G contains a Hamiltonian path if and only if all initial states of $\mathcal{K}_G$ satisfy the PLTL-formula indicated above.

The Hamiltonian path problem can formulated in CTL in the following way. We start by constructing a CTL-formula $g(p_1, \ldots, p_n)$ which is valid when there exists a path from the current state that visits the states for which $p_1 \ldots p_n$ is valid in this order:

$$g(p_1, \ldots, p_n) = p_1 \ \wedge \ \mathsf{EX}\,(p_2 \ \wedge \ \mathsf{EX}\,(\ldots \ \wedge \ \mathsf{EX}\, p_n)\ldots).$$

Because of the branching interpretation of CTL, a formulation of the Hamiltonion path problem in CTL requires an explicit enumeration of all possible Hamiltonian paths. Let Π_n be the set of permutations on $\{1, \ldots, n\}$. There exists a Hamiltonian path in graph G if and only if all initial states of $\mathcal{K}_G$ satisfy the CTL-formula:

$$(\exists\, \theta \in \Pi_n .\, g(p_{\theta_1}, \ldots, p_{\theta_n})) \ \wedge \ (\mathsf{EX})^n q.$$

By the explicit enumeration of all possible permutations we obtain a formula with a length that is exponential in the number of vertices in the graph.

This does not prove that there does not exist an equivalent, but shorter, CTL-formula which describes the Hamiltonion tour problem. However, if a formula of polynomial length would exist, then we were able to solve an NP-complete problem in polynomial time, as verifying a CTL-formula takes polynomial time.
(End of example.)

7.4 Model Checking Fair CTL

In order to be able to ignore certain "unrealistic" computations, CTL may be interpreted with respect to a *fair* Kripke structure. Recall from Chapter 6 that:

Definition 7.7. (**Fair Kripke structure**)
A *fair* Kripke structure is a quadruple $\mathcal{K} = (S, I, R, Label, \mathcal{F})$ where $(S, I, R, Label)$ is a Kripke structure and $\mathcal{F} \subseteq 2^S$ is a set of fairness constraints.

Fairness conditions are thus simply sets of states. A path is fair if and only if each set of states $F_i \in \mathcal{F}$ is visited infinitely often. Note that the notion of fairness depends on the chosen sets F_i. As $\mathcal{F}$ will be clear from the context, we do not make this dependency explicit in the notations, e.g., we simple write $\mathcal{K}$ rather than $\mathcal{K}_{\mathcal{F}}$.

In fair CTL, the formulas are interpreted over all fair paths rather than over all possible paths. Paths that do not visit some F_i infinitely often are thus not considered for the validity of the formula under consideration. How should we adapt *compSat* and its sub-procedures to model-check fair CTL? First, observe that for all propositional operators, the fair interpretation coincides with the ordinary semantics of CTL. So, for these operators no modifications have to be made to the earlier algorithms. The fair interpretations of EX and EU can be obtained in the following way:

$$\begin{aligned} \mathsf{E}_f(\Phi\,\mathsf{U}\,\Psi) &\equiv \mathsf{E}\,(\Phi\,\mathsf{U}\,(\Psi \wedge \mathsf{E}_f\mathsf{G}\,\mathrm{true})) \\ \mathsf{E}_f\mathsf{X}\,\Phi &\equiv \mathsf{EX}\,(\Phi \wedge \mathsf{E}_f\,\mathsf{G}\,\mathrm{true}) \end{aligned}$$

Here, E_f denotes an existential path quantification over fair paths. These equations can be justified by the fact that a finite path is fair if and only if any suffix of it is fair. This fact is used in the second equation by adding the conjunct $\mathsf{E}_f\mathsf{G}\,\mathrm{true}$, thus requiring that the remaining path after establishing Φ goes infinitely often through the sets in $\mathcal{F}$. A similar construction is used in the first equation. Due to these equations, formulae $\mathsf{E}_f(\Phi\,\mathsf{U}\,\Psi)$ and $\mathsf{E}_f\mathsf{X}\,\Phi$ can be checked using the same approach as before given a procedure for dealing with $\mathsf{E}_f\mathsf{G}\,\Phi$.

The algorithm to treat the fair counterpart of $\mathsf{EG}\,\Phi$ is based on the direct, more efficient, algorithm treated before (cf. Table 7.5). Accordingly, the structure $\mathcal{K}[\Phi]$ is constructed, and the SCCs of its induced graph are determined. The fairness sets $\mathcal{F}'$ in $\mathcal{K}[\Phi]$ are defined as $\{\, F_i \cap S' \mid F_i \in \mathcal{F} \,\}$ where S' is the set of states in $\mathcal{K}$ satisfying Φ. The only difference is that the SCCs that do not contain some state in F_i, for some F_i, need to be ignored, as within such SCCs no fair paths will occur. The algorithm *determineSCC* therefore is followed by a procedure that rules out the unfair SCCs. (This simple procedure can, of course, also be integrated with *determineSCC*.) Checking whether a given SCC is fair can be done in time that is proportional to the size of the SCC. For the remaining fair SCCs a backward search is carried out to determine all states for which such fair SCC is reachable.

The correctness of this algorithm follows from the following result:

Theorem 7.6.
For state s in fair *Kripke structure $\mathcal{K}$ and CTL-formula Φ we have: $s \models \mathsf{E}_f\mathsf{G}\,\Phi$ if and only if $s \in \mathcal{K}[\Phi]$ and there exists a non-trivial* fair *SCC in $\mathcal{K}[\Phi]$ reachable from s.*

Proof: Similar to the proof of Theorem 7.4 by exploiting the fact that a path is fair if and only if it has a fair suffix. *qed.*

The thus obtained procedure to check fair CTL, *compfairSat*, has the following

time complexity:

Theorem 7.7.
For fair Kripke structure $\mathcal{K} = (S, R, I, Label, \mathcal{F})$ *with* $|S| = N$, $|R| = M$ *and* $|\mathcal{F}| = K$, *and CTL state-formula* Φ, *the worst-case time-complexity of the model checking algorithm* $compfairSat_{eff}$ *is* $\mathcal{O}(|\Phi| \cdot K \cdot (N+M))$.

So model-checking fair CTL is K times more expensive than model checking CTL, where K is the number of fairness constraints. The worst-case complexity, however, is still linear in both the length of the formula and the size of the Kripke structure.

7.5 Generating Counterexamples

A major strength of model checking is the generation of counterexamples in case a formula is refuted. In this section, we discuss a method to generate counterexamples for fair CTL model checking.

A counterexample is intended to be a (finite or infinite) sequence of states that indicates why the formula under consideration is refuted. For instance, consider the formula $\mathsf{AG}\, p$ in the initial state of our running example. This formula is refuted, and a possible counterexample is $s_1\, s_3\, s_0\, s_2$ as depicted in Figure 7.6, since p does not hold in state s_2. Note that this is a prefix of a path satisfying $\neg\, \mathsf{AG}\, p$, i.e., $\mathsf{EF}\, \neg p$. For formulae of the form $\mathsf{AG}\, \Phi$, a counterexample is thus a sequence that leads to a state for which $\neg\, \Phi$ holds. Similarly, a counterexample to $\mathsf{AX}\, \Phi$ is a sequence satisfying $\mathsf{EX}\, \neg\, \Phi$. A counterexample to $\mathsf{A}\, (\Phi\, \mathsf{U}\, \Psi)$ is a witness to $\mathsf{E}\, (\neg\, \Psi\, \mathsf{U}\, (\neg\, (\Phi\ \vee\ \Psi))\ \vee\ \mathsf{EG}\, \neg\, \Psi$. In general, a counterexample to a universally quantified formula is a witness to an existentially quantified formula. Note that for formulae of the form $\mathsf{E}\, \varphi$ it is impossible to generate sequences as counterexamples in case the verification fails: $\mathsf{E}\, \varphi$ is refuted if and only if there is *no* path that satisfies φ. However, in case of a successful verification, a witness for $\mathsf{E}\, \varphi$ can be returned.

The remaining part of this section is devoted to the generation of witnesses for $\mathsf{EG}\, \Phi$. This will be presented for fair CTL, where it is assumed that the fairness constraints $\mathcal{F} = \{\, F_1, \ldots, F_k \,\}$ with $k > 0$. Since EX- and EU-formulae are defined in terms of $\mathsf{E}_f\, \mathsf{G}\, \mathrm{true}$, the algorithm to determine witnesses for $\mathsf{E}_f\, \mathsf{G}\, \Phi$ can be extended in a rather straightfoward way so as to generate witness for EX and EU. A witness for $s \models \mathsf{E}_f\, \mathsf{G}\, \Phi$ is a fair path σ starting in s such that Φ holds in any state along σ. As the path is fair, each accept set F_i is visited infinitely often. Since Kripke structures are finite, paths have a finite prefix followed by a repetitive finite sequence, representing the (infinite) traversal of a cycle. This allows witnesses to be represented in a finite way.

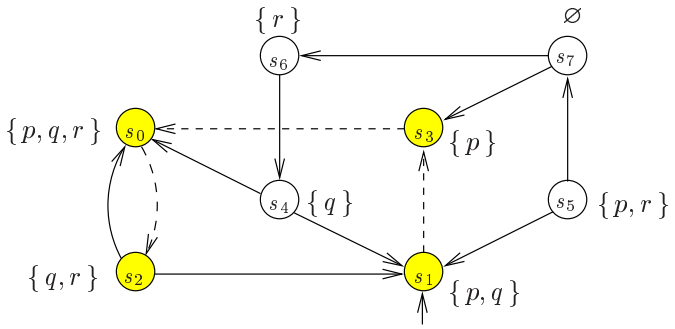


Figure 7.6: A counterexample for $\mathsf{AG}\, p$

Example 7.7. Consider again our running example, cf. Figure 7.1, and assume there is a single accept set $F_1 = \{ s_3 \}$. This corresponds to imposing the fairnes constraint $\neg(q \vee r)$. Example witnesses for $\mathsf{E}_f\mathsf{G}\,(p \vee (q = r))$ under this fairness constraint are, for instance, $(s_1\, s_3\, s_0\, s_2\, s_1)^\omega$ and $(s_1\, s_3\, (s_0\, s_2)^*\, s_1)^\omega$. Although path $s_1\, s_3\, (s_0\, s_2)^\omega$ satisfies $\mathsf{E}_f\mathsf{G}\,(p \vee (q = r))$, it is not a legal witness as it is not fair. *(End of example.)*

For a given formula $\mathsf{E}_f\mathsf{G}\,\Phi$ there may exist several witnesses. Preferably, the shortest witness is provided as feedback to the user. However, the problem of finding the shortest witness for a simple formula like $\mathsf{E}_f\mathsf{G}\,\mathsf{true}$ is NP-complete, and therefore witnesses will be constructed that are short, but not guaranteed of minimal length. (Recall that for PLTL model checking shortest counterexamples can be generated by using a breadth-first search.)

A witness for $s \models \mathsf{E}_f\mathsf{G}\,\Phi$ is incrementally constructed by successively adding states to a finite sequence of states. This process starts with the sequence consisting of state s only, and terminates until a cycle is entirely visited. During this path construction it must be ensured that a fair path results. This is guaranteed by only adding states that satisfy $\mathsf{E}_f\mathsf{G}\,\Phi$. The foundations of this step-by-step construction of the witness are laid down by the following fixed-point characterisation. $[\![\, \mathsf{E}_f\mathsf{G}\,\Phi \,]\!]$ is the gfp of the following function:

$$F_{\mathsf{E}_f\mathsf{G}}(Z) = [\![\, \Phi \,]\!] \cap \{\, s \mid \forall 0 < i \leqslant k.\, R(s) \cap [\![\, \mathsf{E}\,(\Phi\, \mathsf{U}\, at_{Z \cap F_i}) \,]\!] \neq \varnothing \,\}$$

where $at_{S'}$ for set of states S' is an atomic proposition that holds in any state in S' and nowhere else; $at_{S'}$ thus represents the set S'. Note that $[\![\, \mathsf{E}\,(\Phi\, \mathsf{U}\, at_{Z \cap F_i}) \,]\!]$ is characterised as an lfp (cf. Theorem 7.3), so $[\![\, \mathsf{E}_f\mathsf{G}\,\Phi \,]\!]$ is characterised as a *nested* greatest fixed point. Intuitively, $\mathsf{E}_f\mathsf{G}\,\Phi$ holds in state s if s satisfies Φ and there exists a path starting in s on which Φ always holds and for which each F_i is visited infinitely often.

Let us consider the computation of $[\![\, \mathsf{E}_f\mathsf{G}\,\Phi \,]\!]$ in more detail. The first observation is that $[\![\, \mathsf{E}\,(\Phi\, \mathsf{U}\, at_{Z \cap F_i}) \,]\!]$ for fairness constraint F_i can be iteratively

computed using the earlier presented algorithm *compSatEU*. This yields a sequence of approximations $B_i^0, B_i^1, B_i^2, \ldots$ with $B_i^j \subseteq B_i^{j+1}$ where B_i^{n+1} is the set of states from which a state in $Z \cap F_i$ can be reached in at most n steps while satisfying Φ. In the last iteration of the outermost gfp, i.e., when Z equals $[\![\mathsf{E}_f \mathsf{G}\, \Phi]\!]$, the sequence of approximations B_i is stored for each fairness constraint F_i. This is the starting-point for the computation of the witness.

Suppose state $s \models \mathsf{E}_f \mathsf{G}\, \Phi$. Due to the fixed-point characterisation given above, s has some direct successor in $[\![\mathsf{E}\,(\Phi\, \mathsf{U}\, at_{Z \cap F_i})]\!]$ for each $0 < i \leqslant k$. In order to obtain a short witness, we choose the first fairness constraint that can be reached from s. More precisely, fairnes constraint F_i is considered where i is such that

$$n_i = min\{\, k \mid R(s) \cap B_i^n \neq \varnothing \,\} \tag{7.1}$$

is minimal. Suppose state $s_i \in R(s) \cap B_i^n$. Since $s_i \in B_i^n$ and B_i^n is an under-approximation of the set of states $[\![\mathsf{E}\,(\Phi\, \mathsf{U}\, ((\mathsf{E}_f \mathsf{G}\, \Phi) \wedge at_{F_i}))]\!]$, it follows that starting from s_i there exists a Φ-path to a state in $[\![(\mathsf{E}_f \mathsf{G}\, \Phi) \wedge at_{F_i}]\!]$ (of length at most n). By the selection of i as the minimal one satisfying (7.1), s_i (and also s) has the shortest path to a state in the accept set F_i. As s_i is a successor of s in the witness for $\mathsf{E}_f \mathsf{G}\, \Phi$, it follows that $s_i \models \mathsf{E}_f \mathsf{G}\, \Phi$.

The witness constructed so far is $s\, s_i \ldots t_i$, where s is the state we started, s_i is a direct successor of s in B_i^n, and t_i is a state in F_i. In the next step, the states inbetween s_i and t_i need to be determined. This is done in the following way. Let $n > 0$ (otherwise states s_i and t_i coincide). By construction, there exists a direct successor of s_i in B_i^{n-1}. Accordingly, we arbitrarily select one of the states in the set $R(s_i) \cap B_i^{n-1}$. This procedure is repeated for B_i^{n-2}, until B_i^0 is reached, i.e., the state in F_i is reached. As a result, the witness constructed uptil now has the form

$$s\, s_i^n\, s_i^{n-1}\, s_i^{n-2}\, \ldots\, s_i^0$$

where s_i^j is a state in $R^{n-j+1}(s) \cap B_i^j$ (note that $s_i^n = s_i$ and $s_i^0 = t_i \in F_i$). Recall that $R^0(s) = s$ and $R^{j+1}(s) = R(R^j(s))$ for $j \geqslant 0$. The prefix of the witness thus leads to a state in the accept set F_i. In order to deal with the other fairness constraints, the above procedure has to be repeated for the remaining fairness constraints, until all of them are treated. If we order the fairness constraints according to the order in which they are encountered during the above described process, the witness ρ so far is of the following form:

$$\rho = s\ \underbrace{s_1^{n_1} \ldots s_1^0}_{\text{constraint } F_1}\ \underbrace{s_2^{n_2} \ldots s_2^0}_{\text{constraint } F_2}\ \ldots\ \underbrace{s_k^{n_k} \ldots s_k^0}_{\text{constraint } F_k}$$

The last step in completing the witness is to construct a cycle from the final state of the witness constructed so far (i.e., s_k^0) to $s_1^{n_1}$ along which Φ holds. In

this way, it is guaranteed that the witness visits each accept set F_i infinitely often, and thus the witness is a fair path (as required). How to find a Φ-path from the final state of ρ to its second state? For simplicity, let *start* denote a proposition that is valid in $s_1^{n_1}$ and nowhere else, and *end* denote a proposition true only in state s_k^0. Finding the required cycle thus amounts to constructing a witness for the formula:

$$end \wedge \mathsf{EX}\,\mathsf{E}\,(\Phi\,\mathsf{U}\,start)$$

Now there are two possible scenarios: the *start*- and *end*-state belong to the same SCC in $\mathcal{K}[\Phi]$, or not. In the former case, there exists a path from the *end*- to the *start*-state along which Φ holds, and this path can be used to close the cycle, i.e., to complete the witness ρ.

Example 7.8. Consider the Kripke structure depicted in Figure 7.7 with the

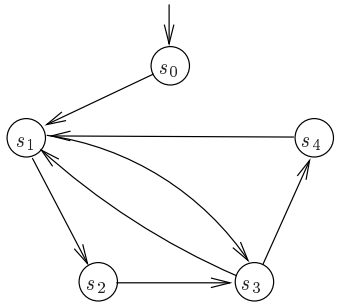


Figure 7.7: An example Kripke structure

fairness constraints $F_1 = \{ s_2 \}$ *and* $F_2 = \{ s_4 \}$*. Suppose we want to generate a witness for* EG *true in state* s_0*. The process starts in state* s_0*, cf. Figure 7.8(a). For the outermost fixed-point computation we have:*

$$F^0_{\mathsf{E}_f\mathsf{G}\,\Phi}(S) = [\![\, true \,]\!] \cap \{\, s \mid R(s) \cap [\![\, \mathsf{EF}\, at_{s_2} \,]\!] \cap [\![\, \mathsf{EF}\, at_{s_4} \,]\!] \,\}$$

In fact, we have $F^0 = F^1$*, so we have reached the fixed point after a single iteration. For* $[\![\, \mathsf{EF}\, at_{s_2} \,]\!]$ *we obtain the series* $B_1^0 = \varnothing$*,* $B_1^1 = \{ s_2 \}$*,* $B_1^2 = \{ s_1, s_2 \}$ $B_1^3 = \{ s_0, s_1, s_2, s_4 \}$*, etc. Note that the first time a direct successor of* s_0 *is "reached" is in* B_1^2*, i.e.,* $n_1(s_0) = 2$*. Similarly, for* $[\![\, \mathsf{EF}\, at_{s_4} \,]\!]$ *the series* $B_2^0 = \varnothing$*,* $B_2^1 = \{ s_4 \}$*,* $B_2^2 = \{ s_3, s_4 \}$*,* $B_2^3 = \{ s_1, s_2, s_3, s_4 \}$*, and so on, is obtained. The first time a direct successor of* s_0 *is "reached" is in* B_2^3*, i.e.,* $n_2(s_0) = 3$*. As* $n_1(s_0) = 2$ *and* $n_2(s_0) = 3$*, the fairness constraint* F_1 *is "reached" first, and the finite sequence* $s_0\, s_1\, s_2$ *is considered as prefix of the witness, cf. Figure 7.8(b).*

In order to treat fairness constraint F_2*, the above procedure is repeated for state* s_2*, the final state of the witness so far. As a direct successor of state* s_2 *is encountered in the second iteration (i.e.,* $R(s_2) \cap B_2^2 \neq \varnothing$*) – formally*

$n_2(s_2) = 2$ – the witness is now extended to $s_0\, s_1\, s_2\, s_3\, s_4$, cf. Figure 7.8(c). This prefix visits each fairness constraint at least once.

Finally, a loop is to be constructed from s_4 to s_1. Since these states belong to the same SCC, a path within this SCC can be chosen, e.g. $s_4\, s_1$. The complete witness for EG *true in state s_0 is thus $s_0\, (s_1\, s_2\, s_3\, s_4)^\omega$, cf. Figure 7.8(d). Note that this is the shortest counterexample. (End of example.)*

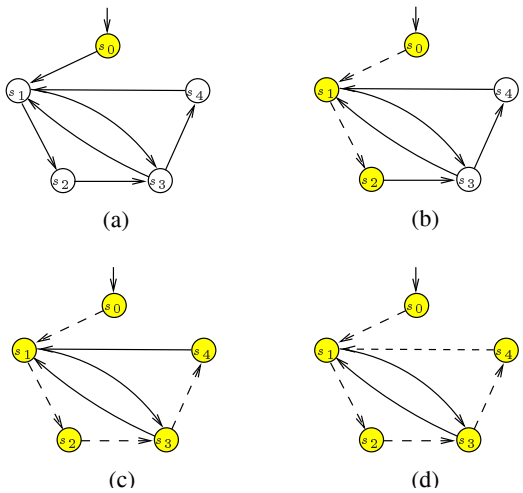


Figure 7.8: Step-wise generation of a witness for EG true

If the *start*- and *end*-state do not belong to the same SCC, a somewhat more involved approach is taken. The basis of the remaining approach is the so-called *condensation* graph of $\mathcal{K}[\Phi]$. Such graphs are defined as follows:

Definition 7.8. (**Condensation graph**)
Let $S_1, \ldots, S_m$ be the set of SCCs of the induced graph of Kripke structure $\mathcal{K} = (S, I, R, Label)$, i.e., $G = (S, R)$. The condensation graph $G_\mathcal{K}$ is the directed graph with the set of vertices $v_1, \ldots, v_m$ such that there is an edge between v_i and v_j if and only if $i \neq j$ and there is an edge in G between some state in S_i and soem state in S_j.

In other words, all states in SCC S_i are condensed into a single vertex v_i. It is not difficult to see that condensation graphs are directed acyclic graphs; otherwise the SCCS are not maximal sub-graphs. In addition, cycles present in the Kripke structure are contained in an SCC, and therefore, represented by a single vertex in the condensation graph.

Example 7.9. Suppose we add the edge from state s_4 to state s_5 to Figure 7.1. The condensation graph of the resulting Kripke structure consists of two

vertices: $v = \{ s_0, s_1, s_2, s_3 \}$ *and* $w = \{ s_4, s_5, s_6, s_7 \}$ *and a single edge from* w *to* v. *(End of example.)*

If the *end*- and *start*-state are not in the same SCC, we "descend" in the condensation graph of $\mathcal{K}[\Phi]$, i.e., we jump from one SCC to the next. In this way, one eventually either encounters a cycle in some SCC, or ends up in the terminal SCC, i.e., an SCC that cannot be left anymore once entered. Note that such terminal SCC is guaranteed to exist as the condensation graph is acyclic. As this last SCC is terminal, it is ensured that it contains a fair Φ-cycle.

Only having considered all fairness constraints, it is checked whether there exists a path leading back to state *start*. If the SCC is left already before the last fairness constraint is satisfied, it is impossible to complete the cycle. To that end, the set of states $[\![\mathsf{E}(\mathsf{EG}\,\Phi\,\mathsf{U}\,\mathit{start})]\!]$ is determined once the *start*-state is encountered. On leaving this set during the witness construction, it is known that the cycle cannot be completed, and the entire witness construction is repeated in the first state that is outside the SCC. The already constructed prefix to that state is to be retained as finally a witness for the original state s has to be returned.

7.6 CTL* model checking

Recall that CTL* contains both CTL and PLTL (cf. Chapter 6), that is to say, for each CTL- or PLTL-formula there exists an equivalent CTL* formula. In fact, CTL* contains even more: it can express formulae that can neither be expressed in PLTL nor in CTL. Interestingly enough, a model-checking algorithm for this logic can be obtained by incorporating a model-checking algorithm for PLTL into the bottom-up tree traversal procedure à la CTL. In order to explain the basic concept behind this elegant combination of two model-checking algorithms, let us recall the syntax of CTL*:

$$\begin{array}{lll} \text{state-formulas} & \Phi ::= p \mid \neg\Phi \mid \Phi \vee \Phi \mid \mathsf{E}\,\varphi \mid \mathsf{A}\,\varphi \\ \text{path-formulas} & \varphi ::= \Phi \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{X}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi \end{array}$$

Whereas in CTL each linear temporal operator such as X and U must be immediately preceded by a path quantifier, CTL* allows for path quantifiers E and A to be arbitrarily nested with these operators. Since for CTL* the following equivalence holds:

$$\mathsf{A}\,\varphi \equiv \neg\,\mathsf{E}\,\neg\,\Phi$$

it suffices to consider an algorithm for $\mathsf{E}\,\varphi$.

Like for CTL, the model-checking algorithm for CTL* is based on a bottom-up traversal of the parse tree of the formula to be checked. Suppose that during this process we encouter a formula of the form $\mathsf{E}\,\varphi$. If $\mathsf{E}\,\varphi$ is a legal CTL-formula, the model-checking procedure for CTL is simply applied. Otherwise, the PLTL model-checking algorithm is used. As φ is not guaranteed to be an PLTL-formula, the model-checking algorithm for PLTL cannot be directly applied. To that end, a preprocessing of φ is performed.

To understand this preprocessing phase, let us introduce the following concepts. Recall that $\mathit{Sub}(\Phi)$ denotes the set of sub-formulae of Φ.[3] A sub-formula of Φ is a *proper* sub-formula if it is a sub-formula different from Φ. The set of proper sub-formulas of Φ is thus simply $\mathit{Sub}(\Phi) - \{\,\Phi\,\}$. A sub-formula is called *maximal* if it is not a proper sub-formula of another sub-formula.

Example 7.10. Consider the CTL-formula* $\mathsf{E}\,\varphi$ *with* $\varphi = p \wedge \mathsf{E}\,(q\,\mathsf{U}\,\mathsf{EX}\,r)$. *The proper sub-formulae of* φ *are* $\mathsf{EX}\,r$, $\mathsf{E}\,(q\,\mathsf{U}\,\mathsf{EX}\,r)$ *and the atomic propositions. The path-formula* $q\,\mathsf{U}\,\mathsf{EX}\,r$ *is not a sub-formula.* $\mathsf{E}\,(q\,\mathsf{U}\,\mathsf{EX}\,r)$ *is a maximal sub-formula, whereas* $\mathsf{EX}\,r$ *is not. (End of example.)*

We now return to model-check $\mathsf{E}\,\varphi$. Due to the bottom-up nature of the algorithm, the states in which any of the (state) sub-formulas of φ holds, are known. This holds in particular for the proper sub-formulae of the form $\mathsf{E}\,\varphi'$. The basic concept is to replace all maximal sub-formula of $\mathsf{E}\,\varphi$ of this form, say $\mathsf{E}\,\varphi_1, \ldots, \mathsf{E}\,\varphi_k$, by "fresh" atomic propositions, $p_1, \ldots, p_k$, say. These propositions do not occur in φ and are such that p_i holds in state s if and only if $\mathsf{E}\,\varphi_i$ holds in s. For instance, applying this recipe on $\mathsf{E}\,\varphi$ with $\varphi = p \wedge \mathsf{E}\,(q\,\mathsf{U}\,\mathsf{EX}\,r)$ yields $\mathsf{E}\,(p \wedge p_1)$. The resulting formula is a PLTL-formula and is fed into a model-checking algorithm for PLTL. (Recall that $\mathsf{E}\,\varphi \equiv \neg\,\mathsf{A}\,\neg\,\varphi$, which is a PLTL-formula, cf. the alternative syntax of PLTL in Chapter 6). Assuming that this algorithm returns the set of states in which the formula holds, this finishes the verification of $\mathsf{E}\,\varphi$. The resulting algorithm is depicted in Table 7.6.

Theorem 7.8.
For Kripke structure $\mathcal{K} = (S, R, I, \mathit{Label})$ *with* $|\,S\,| = N$ *and* $|\,R\,| = M$, *CTL* state-formula* Φ *can be model-checked with a time complexity in* $\mathcal{O}\left(|\,\Phi\,| \cdot 2^N\right)$.

[3] This notion was defined for CTL, but can easily be adapted to CTL*.

```
function compSatE(φ : Formula) : set of State;
(* pre: φ is a CTL* path-formula *)
begin if Eφ ∈ CTL then return compSat_eff(Eφ);
        for each maximal sub-formula Eφ_i of Eφ
        do replace φ_i by p_i in Eφ;
            Sat(p_i) := Sat(Eφ_i);
        od;
         (* let Eφ* be the resulting formula *)
return PLTLchecker(Eφ*);
(* post: compSatE(φ) = { s ∈ S | s ⊨ Eφ } *)
end
```

Table 7.6: Algorithm to determine the states satisfying CTL*-formula $\mathsf{E}\varphi$

7.7 Bibliographic Notes

CTL model checking. The first algorithms for CTL model checking were presented by Clarke and Emerson [46] and (for a logic similar to CTL) by Queille and Sifakis [155], both in 1981. The algorithm by Clarke and Emerson was polynomial in both the size of the Kripke structure and the length of the formula, and could handle fairness. Clarke, Emerson and Sistla [47] presented an efficiency improvement using the detection of strongly connected components and backwards breadth-first search, yielding an algorithm that is linear in both the size of the Kripke structure and the length of the formula. CTL model checking based on a forward search has been proposed by Iwashita, Nakata and Hirose [104]. Emerson and Lei [72] showed that CTL*, that combines PLTL and CTL, can be checked with essentially the same complexity as PLTL using a combination of the algorithms for PLTL and CTL. The same authors consider in [71] CTL model checking under a broad class of fairness assumptions. Practical aspects of CTL* model checking have been reported by Bhat, Cleaveland and Grumberg [23], and more recently, by Visser and Barringer [184]. Cleaveland and Steffen [56] have shown that the alternation-free fragment of the μ-calculus – which is more expressive than CTL – can be checked with the same complexity as CTL, so linear in the size of the formula and the Kripke structure. Alternative algorithms for this logic have been proposed by, amongst others, Andersen [11]. Algorithms for generating counterexamples and witnesses originate from the works by Clarke *et al.* [50] and Hojati, Brayton and Kurshan [94]. More recent developments are the use of satisfiability solvers to find counterexamples up to certain length, as proposed by Clarke *et al.* [44], and the use of tree-like counterexamples as opposed to linear ones by Clarke *et al.* [51].

Global versus local model checking. The algorithms presented in this chapter compute for a given formula Φ the set of all states that satisfy Φ in a bottom-up manner. To do so, in all states all sub-formulae are checked. In this so-called *global* model-checking approach, some states may be checked unnecessarily for some sub-formulae. Alternatively, one could check whether Φ holds in one par-

ticular state, s say. Such *local* approach typically is pursued in a top-down manner by evaluating only those sub-formulae that are needed.] This avoids exploring the parts of the Kripke structure which are irrelevant for the formula to be checked. Local model-checking algorithms for CTL have first been proposed by Vergauwen and Levi [183] and has later been pursued by Heljanko [90]. The worst-case time complexity of local model checking is identical to global model checking.

Automata-based CTL model checking. An alternative model-checking approach for CTL is to use automata over infinite trees (à la automata over infinite words for PLTL). Thomas [174] gives a thorough account on the relationship between CTL and ω-regular languages. Any CTL-formula can be transformed into an automaton over infinite trees, but the size of the resulting automaton is exponential in the length of the formula, yielding a much more inefficient algorithm than the fixed-point computation presented in this chapter. Recent work by Kupferman, Vardi and Wolper [114] significantly improved this situation by using (weak) alternating tree automata. Such tree automata generalise nondeterministic tree automata by allowing several successor states to go down along the same branch of the tree. Based on results of Muller, Saoudi and Schupp [142], it is shown that the translation of CTL-formulae into alternating tree automata is linear in the size of the formula. The space complexity of their algorithm is poly-logarithmic in the size of the Kripke structure. This approach can also be generalised to CTL* and the μ-calculus. To our knowledge, implementations based on this approach have not been reported yet.

CTL model checkers. Clarke and Emerson [46] reported the first (fair) CTL model checker, called EMC. About the same time, Queille and Sifakis [155] announced CESAR a model checker for a branching logic very similar to CTL. EMC was improved in [47] and constituted the basis for SMV (Symbolic Model Verifier), an efficient CTL model checker by McMillan based on a symbolic representation of the state space [133]. Recent variants of SMV are NuSMV [43] developed by the team of Cimatti *et al.*, and SMV by McMillan *et al.* at Cadence Berkeley Laboratories that is focused on compositionality. Both tools are freely available on the internet. Another symbolic CTL model checker is VIS [31]. Software tools that support more expressive branching temporal logics such as the μ-calculus are Truth [119] and the Concurrency Workbench [55].

7.8 Exercises

EXERCISE 7.1. Consider the running example of this chapter as depicted in Figure 7.1, and compute the set $[\![\mathsf{EG}\, q]\!]$ by using the fixed point characterisation of EG.

EXERCISE 7.2. Let $\mathcal{K}$ be a Kripke structure, s be a state in this structure, and Φ, Ψ

CTL-formulae such that $s \models \Phi$. Define $\mathcal{K}' = (\mathcal{K}[\Psi])\,[\neg\,(\Phi \wedge \Psi)]$. Prove or disprove the following statement:

$$\mathcal{K}, s \models \mathsf{E}\,(\Phi\,\mathsf{U}\,\Psi) \text{ if and only if } \mathcal{K}', s \models \mathsf{EF}\,\Psi$$

EXERCISE 7.3. Let $F_{AF}(Z) = [\![\,\Phi\,]\!] \cup \{\, s \mid R(s) \subseteq Z \,\}$ for arbitrary CTL-formula Φ. Questions:

1. Prove that F_{AF} is monotonic on the complete lattice $\langle 2^S, \subseteq \rangle$.
2. Prove that $\mathsf{AF}\,\Phi$ is the least fixed point of F_{AF}.

EXERCISE 7.4. Let $F_{EG}(Z) = [\![\,\Phi\,]\!] \cap \{\, s \mid R(s) \cap Z = \varnothing \,\}$ for arbitrary CTL-formula Φ. Questions:

1. Prove that F_{EG} is monotonic on the complete lattice $\langle 2^S, \subseteq \rangle$.
2. Prove that $\mathsf{EG}\,\Phi$ is the greatest fixed point of F_{EG}.

EXERCISE 7.5. Prove that a path is fair if and only if all its suffixes are fair paths.

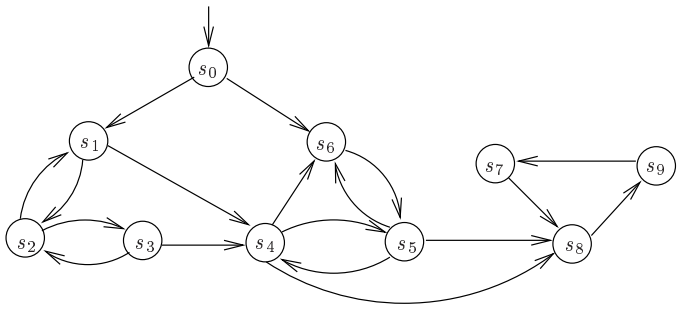
EXERCISE 7.6. Consider the following bounded until-operator: formula $\Phi\,\mathsf{U}^{\leqslant k}\,\Psi$ holds for a path if and only if a Ψ-state is reached along the path via Φ-states only in at most k (for natural k) steps. Questions:

1. Give a fixed-point characterisation of the bounded until-formula $\mathsf{E}\,(\Phi\,\mathsf{U}^{\leqslant k}\,\Psi)$.
2. Give an algorithm to model-check such bounded until-formulae on the basis of this fixed-point characterisation.
3. Suppose the transition relation R of the Kripke structure is given as an incidence matrix $\mathbf{R}$, i.e., $\mathbf{R}(i,j) = 1$ if $(s_i, s_j) \in R$ and 0 otherwise, and suppose $Sat(\Psi)$ is represented as a vector $\underline{i}_\Psi$ such that $\underline{i}_\Psi(s) = 1$ if $s \in Sat(\Psi)$ and 0 otherwise. Give an algorithm to check $\mathsf{EF}^{\leqslant k}\,\Psi$ and indicate how this algorithm can be generalised towards verifying $\mathsf{E}\,(\Phi\,\mathsf{U}^{\leqslant k}\,\Psi)$.

EXERCISE 7.7. Assume that the set of states reachable from some initial state is given (i.e., already computed). Questions:

1. Find an efficient way to verify formulae of the form $\mathsf{AG}\,\Phi$, and justify why your approach is correct.
2. Assume the set of reachable states is computed in an iterative manner, i.e., by successively computing the set of states reachable within 0, 1, 2, 3 ... steps from an initial state. How can the verification of $\mathsf{AG}\,\Phi$ be integrated in this iterative computation yielding an "on-the-fly" algorithm for checking $\mathsf{AG}\,\Phi$?

EXERCISE 7.8. Consider the following Kripke structure with fairness constraints $F_1 = \{ s_2, s_8 \}$ and $F_2 = \{ s_6, s_9 \}$.



Determine a witness for $\mathsf{EG}\,\mathrm{true}$ for state s_0 using the step-wise procedure describned in Section 7.5. Show all intermediate steps in the construction of the witness.

EXERCISE 7.9. The following program is a mutual exclusion protocol for two processes due to Pnueli (taken from [62]). There is a single shared variable s which is either 0 or 1, and initially equals 1. Besides, each process has a local boolean variable y that initially equals 0. The program text for process P_i $(i = 0, 1)$ is as follows:

l0: **loop forever do**
  begin
  l1: Non-critical section
  l2: $(y_i, s) := (1, i)$;
  l3: **wait until** $((y_{1-i} = 0) \ \vee \ (s \neq i))$;
  l4: Critical section
  l5: $y_i := 0$
end.

Here, the statement $(y_i, s) := (1, i)$ is a *multiple assignment* in which variable $y_i := 1$ and $s := i$ is a single, atomic step.

The intuition behind this protocol is as follows. The variables y_0 and y_1 are used by each process to signal the other process of active interest in entering the critical section. On leaving the non-critical section, process P_i sets its own local variable y_i to 1. In a similar way this variable is reset to 0 once the critical section is left. The global variable s is used to resolve a tie situation between the processes. It serves as a logbook in which each process that sets its y variable to 1 signs at the same time. The test at line l3 says that P_0 may enter its critical section if either y_1 equals 0 – implying that its competitor is not interested in entering its critical section – or if s differs from 0 – implying that the competitor process P_1 performed its assignment to y_1 after p_0 assigned 1 to y_0.

Questions concerning this mutual exclusion protocol:

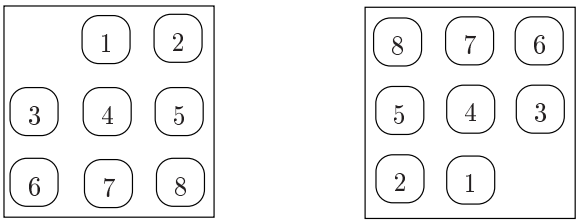
1. Model this protocol in SMV, formulate the property of mutual exclusion in CTL and check this property.
2. Check whether Pnueli's protocol ensures absence of unbounded overtaking, i.e.,

when a process wants to enter its critical section, it eventually will be able to do so. Provide a counterexample (and an explanation thereof) in case this property is violated.

3. Add the fairness constraint `FAIRNESS running` to the process specification in your SMV model of the mutual exclusion program, and check again the property of absence of unbounded overtaking. Compare the obtained results with the results obtained in the previous question without using the fairness constraint.
4. Express in CTL that each process will occupy its critical section infinitely often. Check the property (use again the `FAIRNESS running`).
5. A practical problem with this mutual exclusion protocol is that it is too demanding in the sense that it enforces to perform the assignments to y_i and s (in line l2) in a single step. Most existing hardware systems cannot perform such assignments in one step. Therefore, it is requested to investigate whether any of the four possible realizations of this protocol – in which the aforementioned assignments are not atomic anymore – is a correct mutual exclusion protocol.
   (a) Report for each possible implementation your results, including possible counterexamples and their explanation.
   (b) Compare your results with the results of your PROMELA experiments with this exercise in the previous exercise series.

EXERCISE 7.10.

In this exercise you are confronted with a non-standard example for model checking. The purpose of this exercise is to present the model checker as a solver for combinatorial problems rather than a tool for correctness analysis. These problems involve a search (involving backtracking) of optimal or cost-minimizing strategies such as schedulers or puzzle solutions. The exercise is concerned with Loyd's puzzle that consists of a $N \cdot K$ grid in which there are $N \cdot K - 1$ numbered tiles and a single blank space. The goal of the puzzle is to achieve a predetermined order on the tiles. The initial and final configuration of the tiles for $N = 3$ and $K = 3$ is as follows:



initial configuration final configuration

Note that there are approximately $4 \cdot (N \cdot K)!$ possible moves in this puzzle. For $N = 3$ and $K = 3$ this already amounts to about $1.45\ 10^6$ possible configurations.

Questions concerning Loyd's puzzle:

1. Complete the (partial) model of Loyd's puzzle in SMV that is given below. In this model, there is an array `h` that keeps track of the horizontal positions of the

tiles and an array `v` that records the vertical positions of the tiles such that the position of tile `i` is given by the pair `h[i], v[i]`. Tile 0 represents the blank tile. Position `h[i] = 1` and `v[i] = 1` is the lowest left corner of the puzzle.

```
MODULE main

DEFINE N := 3; K := 3;

VAR move: {u, d, l, r};     -- the possible tile-moves
    h: array 0..8 of 1..3;  -- the horizontal positions of all tiles
    v: array 0..8 of 1..3;  -- .... and their vertical positions

ASSIGN -- the initial horizontal and vertical positions of all tiles

init(h[0]) := 1;  init(v[0]) := 3;
init(h[1]) := 2;  init(v[1]) := 3;
init(h[2]) := 3;  init(v[2]) := 3;
init(h[3]) := 1;  init(v[3]) := 2;
init(h[4]) := 2;  init(v[4]) := 2;
init(h[5]) := 3;  init(v[5]) := 2;
init(h[6]) := 1;  init(v[6]) := 1;
init(h[7]) := 2;  init(v[7]) := 1;
init(h[8]) := 3;  init(v[8]) := 1;

ASSIGN

-- determine the next positions of the blank tile

next(h[0]) :=      -- horizontal position of the blank tile
       case
                              -- one position right
                              -- one position left
              1 : h[0];       -- keep the same horizontal position

       esac;

next(v[0]) :=      -- vertical position of the blank tile
       case
                              -- one position down
                              -- one position up
              1 : v[0];       -- keep the same vertical position
       esac;

-- determine the next positions of all non-blank tiles

next(h[1]) :=      -- horizontal position of tile 1
       case


       esac;

next(v[1]) :=      -- vertical position of tile 1
       case


       esac;

-- and similar for all remaining tiles
```

A possible way to proceed is as follows:

(a) First, consider the possible moves of the blank tile (i.e., the blank space). Notice that the blank space cannot be moved to the left in all positions. The same applies to moves upwards, downwards and to the right.

(b) Then try to find the possible moves of tile `[1]`. The code for tiles `[2]` through `[8]` are obtained by simply copying the code for tile `[1]` while replacing all references to `[1]` to references of the appropriate tile number. (Unfortunately, SMV does not support arrays to be indexed by variables.)

(c) Test the possible moves by running a simulation.

2. Define an atomic proposition `goal` that describes the desired goal configuration of the puzzle. Add this definition to your NuSMV specification by incorporating the following line(s) in your NuSMV model:

```
DEFINE goal :=   .............. ;
```

where the dotted lines contain your description of the goal configuration.

3. Find a solution to the puzzle by imposing the appropriate CTL-formula to the NuSMV specification, and running the model checker on this formula.

This exercise has been taken from [54].

EXERCISE 7.11. Consider the mutual exclusion algorithm by the Dutch mathematician Dekker. There are two processes P_1 and P_2, two boolean-valued variables b_1 and b_2 whose initial values are false, and a variable k which may take the values 1 and 2 and whose initial value is arbitrary. The i-th process (i=1, 2) may be described as follows, where j is the index of the other process:

while true **do**
begin $b_i :=$ true;
  while b_j **do**
    if $k = j$ **then begin**
      $b_i :=$ false;
      while $k = j$ **do skip**;
      $b_i :=$ true
      end;
  $\langle$ critical section $\rangle$;
  $k := j$;
  $b_i :=$ false
end

Questions:

1. Model Dekker's algorithm in SMV.
2. Verify whether this algorithm satisfies the following properties:
3. Mutual exclusion: two processes cannot be in their critical section at the same time.

4. Absence of individual starvation: if a process wants to enter its critical section, it is eventually able to do so.

Hint: use the `FAIRNESS running` statement in your SMV specification for proving the latter property in order to prohibit unfair executions (that might trivially violate these requirements).

EXERCISE 7.12. In the original mutual exclusion protocol by Dijkstra in 1965, another Dutch mathematician, it is assumed that there are $n \geqslant 2$ processes, and global variables b, c : **array** $[1 \ldots n]$ of **boolean** and an integer k. Initially all elements of b and of c have the value true and the value of k belongs to $1, 2, \ldots, n$. The i-th process may be represented as follows:

```
var j : integer;
while true do
begin b[i] := false;
      L_i : if k ≠ i then begin c[i] := true;
                                if b[k] then k := i;
                                goto L_i
                          end;
            else begin c[i] := false;
                       for j := 1 to n do
                       if (j ≠ i ∧ ¬ (c[j])) then goto L_i
                 end
      ⟨ critical section ⟩;
      c[i] := true;
      b[i] := true
end
```

Questions:

1. Model this algorithm in SMV and
2. Check the mutual exclusion property (at most one process can be in its critical section at any point in time) in two different ways: by means of a CTL-formula using `SPEC` and by using invariants. Try to check this property for n=2 through n=5, by increasing the number of processes gradually and compare the sizes of the state spaces and the run time needed for the two ways for verifying the mutual exclusion property.
3. Check the absence of individual starvation property: if a process wants to enter its critical section, it is eventually able to do so.

EXERCISE 7.13. In order to find a fair solution for N processes, Peterson proposed in 1981 the following protocol. Let $Q[1 \ldots N]$ (Q for Queue) and $T[1 \ldots N]$ (T for Turn), be two shared arrays which are initially 0 and 1, respectively. The variables i and j are local to the process with i containing the process number. The code of process i is as follows:

```
while true do
for  j := 1 to  N − 1 do
begin
        Q[i] := j;
        T[j] := i;
        wait until  (T[j] ≠ i  ∨  (forall k ≠ i. Q[k] < j))
end;
⟨ critical section ⟩;
Q[i] := 0
end
```

Questions:

1. Model Peterson's algorithm in SMV.
2. Verify whether this algorithm satisfies the following properties:
   (a) Mutual exclusion.
   (b) Absence of individual starvation.

Chapter 8

Timed Automata

This chapter introduces timed automata, a model aimed at representing continuous-time systems in a symbolic, i.e., mostly finite, way. The ingredients of timed automata, the semantics in terms of infinite timed transition systems, and the composition of timed automata are presented. An industrial example illustrates the use of timed automata to model real-time systems.

8.1 Introduction

The logics we have encountered so far are interpreted over Kripke structures. These automata describe how a reactive system evolves from one state to another. Timing aspects are, however, not covered. That is, no indications are given about how long a system will stay in a state, and there are no possibilities to specify that, for instance, a certain transition may only be taken at a particular time point. However, reactive systems such as device drivers, coffee machines, communication protocols and automatic teller machines, must react in time – they are *time critical*. The behavior of time-critical systems is typically subject to rather stringent timing constraints. For a train crossing it is essential that once the approach of a train is detected, it is closed within a certain time bound in order to halt car and pedestrian traffic before the train reaches the crossing. For a radiation machine the time period during which a cancer patient is subject to a high dosis of radiation is extremely important; a small exceed of this period is dangerous and can cause the patient's death.

To put it in a nutshell:

> *Time-critical systems are systems in which correctness depends not only on the logical result of the computation but also on the time at which the results are produced.*

As timeliness is of vital importance to reactive systems, it is essential that the timing constraints of the system are guaranteed to be met. Checking whether timing constraints are met is the subject of model-checking real-time systems.

The essential ingredients for model-checking time-critical systems are (i) a model description language for such systems, (ii) a property specification language, and (iii) model-checking algorithms. This chapter is concerned with *timed automata*, a model description language for real-time systems. Timed automata are an extension of finite-state automata with clocks that are used to measure the elapse of time. Since their conception, timed automata have been used for the specification of various types of time-critical systems, ranging from communication protocols to safety-critical systems. In addition, several powerful model-checking tools have been developed for timed automata. Chapter 9 treats an extension of the branching temporal logic CTL, called timed CTL, to specify timing properties over timed automata. Chapter 10 deals with algorithms for model-checking timed automata against timed CTL-formulas.

This chapter is organized as follows. Section 8.2 introduces timed automata. Section 8.3 defines the formal interpretation of a timed automaton in terms of a timed labelled transition system. Section 8.4 deals with the parallel composition of timed automata. Section 8.5 presents a model of a file transfer protocol using a network of timed automata.

8.2 Timed Automata

Timed automata are used to model finite-state real-time systems. A timed automaton is in fact a finite-state automaton equipped with a finite set of real-valued *clock variables*, called clocks for short.

Definition 8.1. (**Clock**)
A clock is a variable ranging over $\mathbb{R}^+$.

In the sequel we will use x, y and z as clocks. A "state" of a timed automaton consists of the current *location* of the automaton plus the current values of all clock variables.[1]

Clocks may be initialized to zero when the system makes a transition. Once initialized, they start incrementing their value implicitly. All clocks proceed at the same rate. More precisely, when time advances a single time-unit, all clocks will have progressed by a single time-unit. The value of a clock thus denotes the amount of time that has been elapsed since it has been initialized. Clocks are used to measure the elapse of time when the timed automaton remains in a

[1] Note the deliberate distinction between the terminology location and state.

location. They can thus be considered as stop-watches that can be started and checked independently of each other, but they all refer to the same global time frame (i.e., global clock).

The system may evolve by remaining in a location while time progresses, or may move from one location to another by taking a transition. Transitions are assumed to take no time, i.e., transitions are instantaneous. Conditions on the values of clocks are used as enabling conditions (or *guards*) of transitions: only if the clock constraint is fulfilled, the transition is enabled, and may be taken; otherwise, the transition is blocked. For simplicity, we assume that enabling conditions only depend on clocks; in real applications, though, they may also depend on other system variables that are not clocks, cf. the model in Section 8.5. *Invariants* on clocks are used to limit the amount of time that may be spent in a location. Enabling conditions and invariants are constraints over clocks:

Definition 8.2. (**Clock constraints**)
Let C be a set of clocks with $x \in C$ and c be a natural number. The *clock constraints* over C satisfy the following rules:

1. $x < c$ and $x \leqslant c$ are clock constraints
2. If α is a clock constraint, then $\neg\,\alpha$ is a clock constraint
3. If α and β are clock constraints, then $\alpha \wedge \beta$ is a clock constraint
4. Anything else is not a clock constraint.

The set of clock constraints over C is denoted by $\mathsf{Constraints}(C)$.

Throughout this chapter we use abbreviations such as $x \geqslant c$ for $\neg\,(x < c)$ and $x = c$ for $x \leqslant c \ \wedge \ x \geqslant c$, and so on. The choice of legal clock constraints is an important one. One could, for instance, allow addition of constants like $x \leqslant c{+}d$ for $d \in \mathbb{N}$ without problems, but addition of clock variables like in $x{+}y < 3$, would make the model-checking problem undecidable. Also if c could be a real number like in $x \leqslant 2\pi$, then model-checking a real-time temporal logic that is interpreted in a dense time domain – as for timed CTL – becomes undecidable. Therefore, c is required to be a natural. Decidability of the model-checking problem is not affected if the constraint is relaxed such that c is allowed to be a rational number. In this case the rationals in each formula can be converted into naturals by suitable scaling. For instance, the clock constraint $x \leqslant \frac{2}{7}$ can be changed into $x \leqslant 2$ by multiplying all constants in enabling conditions and invariants in the timed automaton by a factor seven. In general, we can multiply each constant by the least common multiple of denominators of all constants appearing in the enabling conditions and invariants of the timed automaton.

Definition 8.3. (**Timed automaton**)
A *timed automaton* $\mathcal{A}$ is a tuple $(L, I, C, \rightarrow, Label, inv)$ with

- L, a non-empty, finite set of locations
- $I \subseteq L$, the set of initial locations
- C, a finite set of clocks
- $\rightarrow \subseteq L \times (\mathsf{Constraints}(C) \times 2^C) \times L$, the transition relation
- $Label : L \longrightarrow 2^{AP}$, an interpretation function on L
- $inv : L \longrightarrow \mathsf{Constraints}(C)$, an *invariant*-assignment function.

A timed automaton is a Kripke structure equipped with a set C of clocks. Transitions are labelled with tuples (α, C') where α is a clock constraint on the clocks of the timed automaton, and $C' \subseteq C$ is a set of clocks. For $(l, \alpha, C, l') \in \rightarrow$ we write $l \xrightarrow{\alpha, C'} l'$. The intuitive interpretation of $l \xrightarrow{\alpha, C'} l'$ is that the timed automaton can move from location l to location l' when clock constraint α holds. Besides, when moving from location l to l', any clock in C' will be reset to 0. The function *Label* has the same role as for Kripke structures and associates to a location the set of atomic propositions that are valid in that location. As we will see in the next chapter, this function is only relevant for defining the satisfaction of atomic propositions for timed CTL. Finally, function *inv* assigns to each location an *invariant* that specifies how long the timed automaton may stay there. For location l, $inv(l)$ constrains the amount of time that may be spent in l. That is to say, once the invariant $inv(l)$ becomes invalid, the location l must be left immediately. If this is not possible – as there is no outgoing transition enabled – no further progress is possible. As time progress is no more possible, this situation is also known as a *timelock*.

The precise interpretation of timed automata will be defined later on. We first give some simple examples.

For depicting timed automata we adopt the following conventions. Circles denote locations and transitions are represented by arrows. Invariants are indicated inside locations, unless they equal 'true', i.e., if no constraint is imposed on delaying. Arrows are equipped with labels that consist of an optional clock constraint and an optional set of clocks to be reset, separated by a straight horizontal line. If the clock constraint equals 'true' and if there are no clocks to be reset, the arrow label is omitted. The horizontal line is omitted if either the clock constraint is 'true' or if there are no clocks to be reset. We will omit the labelling *Label* in this chapter. This function will play a role when discussing the model-checking algorithm only, and is of no importance for the other concepts to be introduced.

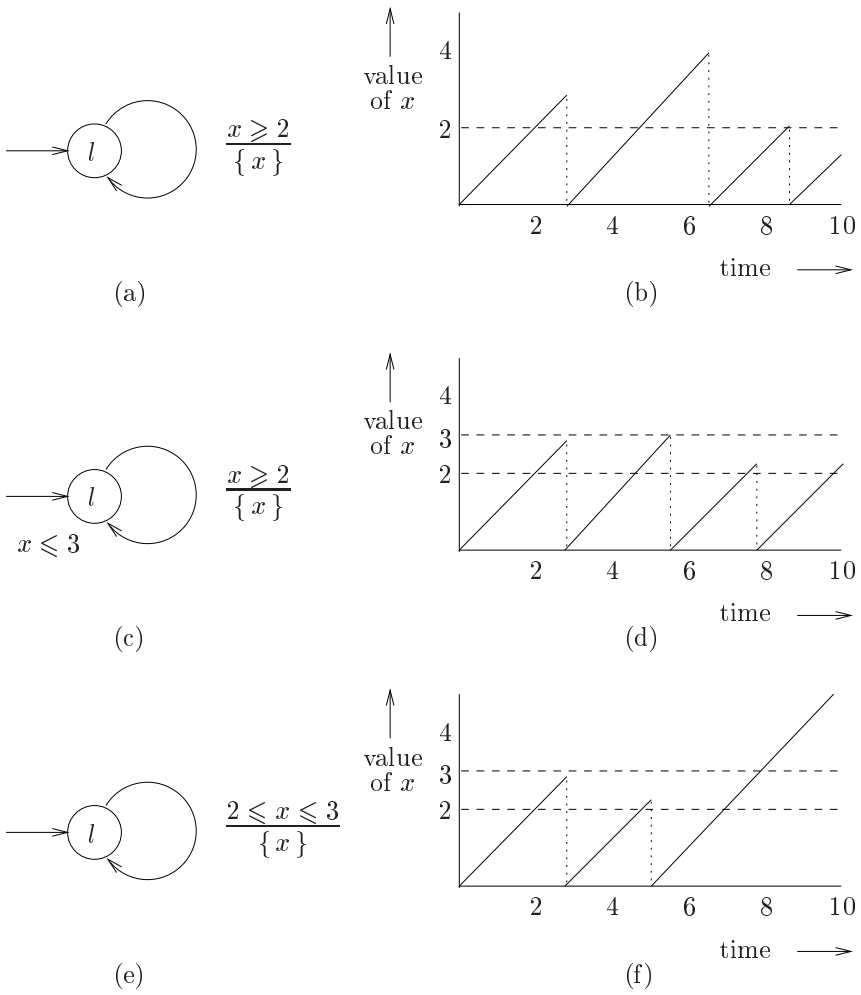


Figure 8.1: Some timed automata with a single clock and one of their evolutions

Example 8.1. Figure 8.1(a) depicts a simple timed automaton with one clock x and one location l equipped with a self-loop. The self-transition can be taken if clock x has at least the value 2, and when being taken, clock x is reset. Initially, clock x has the value 0. Figure 8.1(b) gives an example execution of this timed automaton, by depicting the value of clock x versus the elapsed time. Each time the clock is reset to 0, the automaton moves from location l to location l. Due to the invariant 'true' in l, time can progress without bound while being in l. In particular, a legal behavior of this automaton is to stay in location l ad infinitum. Formally, $L = I = \{ l \}$, $C = \{ x \}$, $l \xrightarrow{x \geqslant 2, \{ x \}} l$, and $inv(l) = true$.

Changing the timed automaton of Figure 8.1(a) slightly by incorporating an invariant $x \leqslant 3$ in location l, leads to the effect that x cannot progress without bound anymore. Rather, if $x \geqslant 2$ (enabling constraint) and $x \leqslant 3$ (invariant) the outgoing transition must be taken. This is illustrated in Figure 8.1(c) and (d).

Observe that the same effect is not obtained when strengthening the enabling constraint in Figure 8.1(a) into $2 \leqslant x \leqslant 3$ *while keeping the invariant 'true' in location* l*. In that case, the outgoing transition can only be taken when* $2 \leqslant x \leqslant 3$ *– as in the previous scenario – but is not forced to be taken, i.e., it can simply be ignored by letting time pass while staying in* l*. This is illustrated in Figure 8.1(e) and (f).* *(End of example.)*

From these examples it is clear that clocks are piece-wise continuous real-valued functions of time where discontinuities may occur when a transition is taken, cf. Figures 8.1(b), (d) and (f).

Different clocks can be started at different times, and hence there is no lower bound on their difference. This is, for instance, not possible in a discrete-time model, where the difference between two concurrent clocks is always a multiple of one unit of time. Having multiple clocks thus allows to model multiple concurrent delays. This is exemplified in the following example.

Example 8.2. Figure 8.2(a) depicts a timed automaton with two clocks, x *and* y*. Initially, both clocks start running from value 0 on, until two time-units have passed. From this point in time, both transitions are enabled and can be taken non-deterministically. As a result, either clock* x *or clock* y *is reset, while the other clock continues. It is not difficult to see that the difference between clocks* x *and* y *is arbitrary. An example evolution of this timed automaton is depicted in part (b) of the figure.* *(End of example.)*

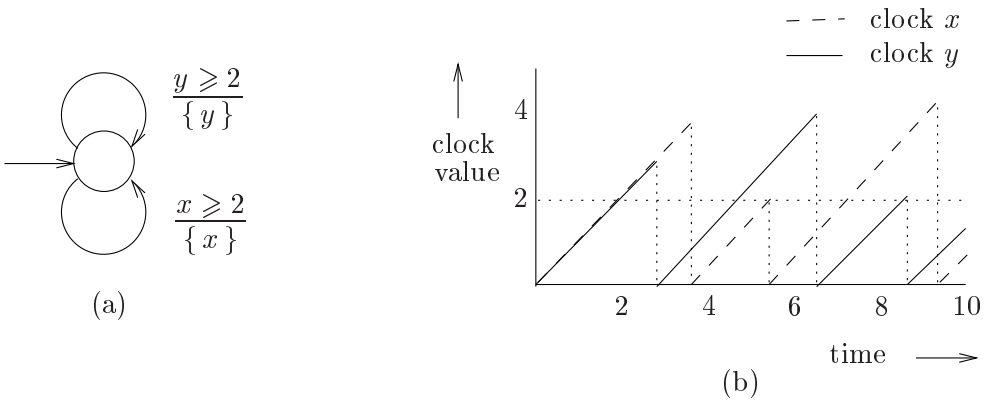


Figure 8.2: A timed automaton with two clocks and a sample evolution of it

Example 8.3. Figure 8.3 shows a timed automaton with two locations, named off *and* on*, and two clocks* x *and* y*. All clocks are initialized to 0 if the initial location* off *is entered. The timed automaton in Figure 8.3 models a switch that controls a light. The switch may be turned on at any time instant since the light has been switched on for at least two time units, even if the light is still on. It may switch automatically off exactly 9 time-units after the most recent time the*

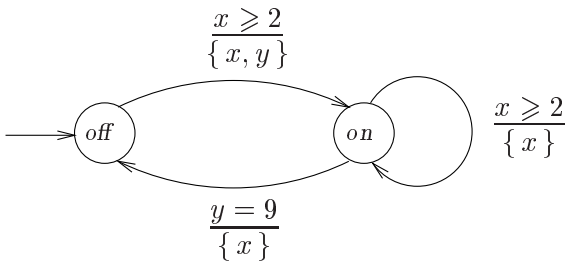


Figure 8.3: A timed automaton example: the switch

light has turned from off to on. Clock x is used to keep track of the delay since the last time the light has been switched on. The transitions labelled with the clock constraint $x \geqslant 2$ model the on-switching transitions. Clock y is used to keep track of the delay since the last time that the light has moved from location off to on, and controls switching the light off. *(End of example.)*

8.3 Semantics of Timed Automata

The previous examples indeed suggest that the state of a timed automaton is determined by (i) its current location and (ii) the current values of all clocks. In fact, timed automata represent labelled transition systems with infinitely many states, and possibly infinitely many outgoing transitions of a state (i.e., infinite branching).

8.3.1 Clock Valuations

The values of clocks are formally defined by *clock valuations*.

Definition 8.4. (**Clock valuation**)
A *clock valuation* v for a set C of clocks is a function $v : C \longrightarrow \mathbb{R}^+$, assigning to each clock $x \in C$ its current value $v(x)$.

Let $\mathrm{V}al(C)$ denote the set of all clock valuations over C. A *state* of $\mathcal{A}$ is a pair (l, v) with l a location in $\mathcal{A}$ and v a valuation over C, the set of clocks of $\mathcal{A}$.

Example 8.4. Consider the timed automaton of Figure 8.3. Some states of this timed automaton are the pairs (off, v) with $v(x) = v(y) = 0$ and (off, v') with $v'(x) = 4$ and $v'(y) = 13$ and (on, v'') with $v''(x) = 2.7$ and $v''(y) = 13$. Note that the latter state is not reachable. *(End of example.)*

Let v be a clock valuation on set of clocks C. For positive real d, clock valuation

$v{+}d$ denotes that all clocks are increased by d with respect to valuation v. It is defined by $(v{+}d)(x) = v(x){+}d$ for all clocks $x \in C$. Clock valuation $\mathsf{reset}\ x\ \mathsf{in}\ v$, valuation v with clock x reset, is defined by

$$(\mathsf{reset}\ x\ \mathsf{in}\ v)(y) = \begin{cases} v(y) & \text{if } y \neq x \\ 0 & \text{if } y = x. \end{cases}$$

Nested occurrences of reset are typically abbreviated. For instance, $\mathsf{reset}\ x\ \mathsf{in}$ $(\mathsf{reset}\ y\ \mathsf{in}\ v)$ is simply denoted $\mathsf{reset}\ x, y\ \mathsf{in}\ v$.

Example 8.5. Consider the clock valuations v and v' of the previous example. Valuation $v{+}9$ is defined by $(v{+}9)(x) = (v{+}9)(y) = 9$. In clock valuation $\mathsf{reset}\ x\ \mathsf{in}\ (v{+}9)$*, clock x has value 0 and clock y reads 9. Clock valuation v' now equals* $(\mathsf{reset}\ x\ \mathsf{in}\ (v{+}9)) + 4$. *(End of example.)*

We can now formally define what it means for a clock constraint to be valid or not. This is done in a similar way as characterizing the semantics of a temporal logic, namely by defining a satisfaction relation. In this case the satisfaction relation $\models$ is a relation between clock valuations (over set of clocks C) and clock constraints (over C).

Definition 8.5. (**Evaluation of clock constraints**)
For $x \in C$, $v \in \mathit{Val}(C)$, natural c and $\alpha, \beta \in \mathsf{Constraints}(C)$ we have:

$$\begin{array}{ll} v \models x \leqslant c & \text{iff } v(x) \leqslant c \\ v \models x < c & \text{iff } v(x) < c \\ v \models \neg\, \alpha & \text{iff } v \not\models \alpha \\ v \models \alpha\ \wedge\ \beta & \text{iff } v \models \alpha\ \wedge\ v \models \beta. \end{array}$$

For negation and conjunction, the rules are identical to those for propositional logic. In order to check whether $x \leqslant c$ is valid in v, it is simply checked whether $v(x) \leqslant c$. The same applies to $x < c$.

Example 8.6. Consider clock valuation v, $v{+}9$ and $\mathsf{reset}\ x\ \mathsf{in}\ (v + 9)$ *of the previous example and suppose we want to check the validity of $\alpha = x \leqslant 5$. It follows $v \models \alpha$, since $v(x) = v(y) = 0$. We have $v{+}9 \not\models \alpha$ since $(v{+}9)(x) = 9 \not\leqslant 5$. It follows that* $\mathsf{reset}\ x\ \mathsf{in}\ (v{+}9) \models \alpha$. *(End of example.)*

8.3.2 Timed Transition Systems

The interpretation of timed automata is defined in terms of an infinite transition system $(S, \longrightarrow)$ where S is a set of states, i.e. pairs of locations and clock

valuations, and $\longrightarrow$ is the transition relation that defines how to evolve from one state to another. This transition relation should not be confused with the transitions in the timed automaton itself! There are two possible ways in which a timed automaton can proceed: by traversing a transition in the automaton, or by letting time progress while staying in the same location. The former is called a discrete transition, the latter a delay transition. A single labelled transition relation $\longrightarrow$ is used; for discrete transitions the label equals $*$ while delay transitions are labelled with a positive real number indicating the amount of time that has elapsed.

Definition 8.6. (**Transition system underlying a timed automaton**) Let $\mathcal{A} = (L, I, C, \rightarrow, Label, inv)$ be a timed automaton. The labelled transition system associated to $\mathcal{A}$, denoted $[\![\,\mathcal{A}\,]\!]$, is defined as $(S, I', \longrightarrow)$ where:

- $S = \{\, (l, v) \in L \times Val(C) \mid v \models inv(l) \,\}$
- $I' = \{\, (l_0, v_0) \mid l_0 \in I \,\}$ where $v_0(x) = 0$ for all $x \in C$
- the transition relation $\longrightarrow \subseteq S \times (\mathbb{R}^+ \cup \{\, * \,\}) \times S$ is the smallest relation defined by the rules:
  1. $(l, v) \xrightarrow{*} (l', \mathsf{reset}\ C'\ \mathsf{in}\ \ v)$ if the following conditions hold:
     (a) $l \xrightarrow{\alpha, C'} l'$
     (b) $v \models \alpha$, and
     (c) $(\mathsf{reset}\ C'\ \mathsf{in}\ \ v) \models inv(l')$
  2. $(l, v) \xrightarrow{d} (l, v{+}d)$, for positive real d, if the following condition holds:
     $$\forall\, d' \leqslant d.\ v{+}d' \models inv(l).$$

The set of states is the set of pairs (l, v) such that l is a location of $\mathcal{A}$ and v is a clock valuation over C, the set of clocks in $\mathcal{A}$, such that v does not invalidate the invariant of l. Note that this set may include states that are unreachable. For a transition that corresponds to (a) traversing transition e in the timed automaton it must be that (b) v satisfies the clock constraint of $l \xrightarrow{\alpha, C'} l'$ (otherwise the transition is disabled), and (c) the new clock valuation, that is obtained by resetting all clocks C' in v, satisfies the invariant of the target location l' (otherwise it is not allowed to be in l'). Idling in a location (second clause) for some positive amount of time is allowed if the invariant is respected while time progresses. Notice that it does not suffice to only require $v{+}d \models inv(l)$, since this could invalidate the invariant for some $d' < d$. For instance, for $inv(l) = (x \leqslant 2) \vee (x > 4)$ and state (l, v) with $v(x) = 1.5$ it should not be allowed to let time pass with 3 time-units: although the resulting valuation $v{+}3 \models inv(l)$, there is some intermediate valuation, e.g., $v{+}2$, that invalidates $inv(l)$.

For a state (l, v) such that $v \models inv(l)$, there are infinitely many outgoing delay transitions of the form $(l, v) \xrightarrow{d}$ as d can be selected from a dense domain.

An execution of a timed automaton corresponds to a path through its timed transition system.

Definition 8.7. (**Path**)
A *path* σ is an infinite sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ such that $s_i \xrightarrow{a_i} s_{i+1}$ for all $i \geqslant 0$.

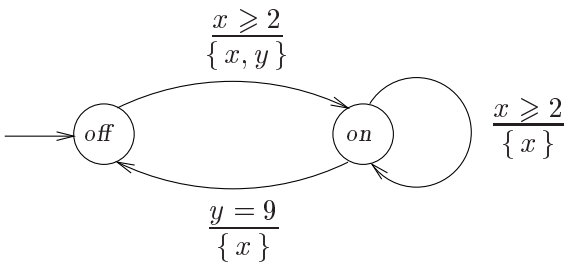
The set of paths starting in a given state is defined as before. Recall a_i equals * for a discrete transition, and a (positive) real number for a delay transition. In order to "measure" the amount of time that elapses on a path, we introduce:

Definition 8.8. (**Elapsed time on a path**)
For path $\sigma = s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots$ and natural i, the time elapsed from s_0 to s_i, denoted $\Delta(\sigma, i)$, is defined by:

$$\begin{array}{rcl} \Delta(\sigma, 0) & = & 0 \\ \Delta(\sigma, i{+}1) & = & \Delta(\sigma, i) + \left\{ \begin{array}{ll} 0 & \text{if } a_i = * \\ a_i & \text{if } a_i \in \mathbb{R}^+. \end{array} \right. \end{array}$$

$\Delta(\sigma, i)$ is thus the accumulated time that has elapsed since i (delay or discrete) transitions have taken place.

Example 8.7. Recall the light switch from Example 8.3:



A prefix of an example path of the switch is

$$\begin{array}{rcl} \sigma & = & (off, v_0) \xrightarrow{3} (off, v_1) \xrightarrow{*} (on, v_2) \xrightarrow{4} (on, v_3) \xrightarrow{*} (on, v_4) \\ & & \xrightarrow{1} (on, v_5) \xrightarrow{2} (on, v_6) \xrightarrow{2} (on, v_7) \xrightarrow{*} (off, v_8) \ldots \end{array}$$

with $v_0(x) = v_0(y) = 0$, $v_1 = v_0{+}3$, $v_2 = \mathsf{reset}\ x, y\ \mathsf{in}\ v_1$, $v_3 = v_2{+}4$, $v_4 = \mathsf{reset}\ x\ \mathsf{in}\ v_3$, $v_5 = v_4{+}1$, $v_6 = v_5{+}2$, $v_7 = v_6{+}2$ *and* $v_8 = \mathsf{reset}\ x\ \mathsf{in}\ v_7$. *These*

clock valuations are summarized by the following table:

	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
x	0	3	0	4	0	1	3	5	0
y	0	3	0	4	4	5	7	9	9

The transition $(\mathit{off}, v_1) \xrightarrow{*} (\mathrm{on}, v_2)$ *is, for instance, possible since (a) there is an transition from* off *to* on*, (b)* $v_1 \models x \geqslant 2$ *since* $v_1(x) = 3$*, and (c)* $v_2 \models \mathit{inv}(\mathit{on})$*. We have, for instance,* $\Delta(\sigma, 3) = 7$ *and* $\Delta(\sigma, 6) = 12$*.*

Another possible evolution of the switch is to stay infinitely long in location off *by making infinitely many delay transitions. Although at some point, i.e. if* $v(x) \geqslant 2$ *the transition to location* on *is enabled, it can be ignored continuously. Similarly, the switch may stay arbitrarily long in location* on*. These behaviors are caused by the fact that* $\mathit{inv}(\mathit{off}) = \mathit{inv}(\mathit{on}) = \mathit{true}$*.*

If we modify the switch such that $\mathit{inv}(\mathit{off})$ *becomes* $y \leqslant 9$ *while* $\mathit{inv}(\mathrm{on})$ *remains true, the aforementioned path* σ *is still legal. In addition, the light may stay infinitely long in location* off *– while awaiting a person that pushes the button – it* must *switch off automatically if during 9 time-units the automaton has not switched from* off *to* on*.* *(End of example.)*

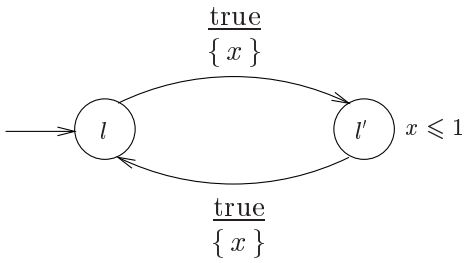


Figure 8.4: An example Zeno timed automaton

It is possible that subsequent discrete transitions in a path take place at the same time. For instance, the timed automaton depicted in Figure 8.4 allows a path $(l, 0.7) \xrightarrow{*} (l', 0) \xrightarrow{*} (l, 0) \xrightarrow{\pi} (l, \pi) \ldots$ where moving from location l to l' and back takes place in zero time-units. Although this may be convenient for certain purposes, e.g., when a certain delay is negligible, one typically wants time to advance at some point since it is counterintuitive, and impossible to realize, a path in which time never progresses. That is to say, a path like $(l, 0) \xrightarrow{*} (l', 0) \xrightarrow{*} (l, 0) \xrightarrow{*} (l, 0) \ldots$ is not considered to be realistic. A timed automaton that allows such behavior is called *Zeno*.

Path σ is called *time-divergent* if $\lim_{i \to \infty} \Delta(\sigma, i) = \infty$. The set of time-divergent paths starting at state s is denoted $\mathit{Paths}^{\infty}(s)$. An example of a non time-divergent path is a path that visits an infinite number of states in a bounded

amount of time. For instance, the path

$$s_0 \xrightarrow{2^{-1}} s_1 \xrightarrow{2^{-2}} s_2 \xrightarrow{2^{-3}} s_3 \ldots s_k \xrightarrow{2^{-k+1}} s_{k+1} \ldots$$

is not time-divergent, since an infinite number of states is visited in the bounded interval $[\frac{1}{2}, 1]$. In non-Zeno timed automata such paths are explicitly excluded.

Definition 8.9. (**Non-Zeno timed automaton**)
A timed automaton $\mathcal{A}$ is called *non-Zeno* if from any of its states some time-divergent path can start.

8.4 Composing Timed Automata

To enable the composition of timed automata we slightly change the transition relation of a timed automaton, and allow – besides an enabling condition and a set of clocks to be reset – an action as label of a transition. Actions are assumed to be taken from an alphabet Σ. Transitions are now thus labelled by a triple (a, α, C') with action a, enabling condition α and set of clocks C'. The addition of actions as labels allows us to specify timed systems in a compositional manner. The elementary operation on timed automata that allows to do so is – like for labelled transition systems – parallel composition.

Definition 8.10. (**Composition of timed automata**)
Let $\mathcal{A}_i$ be the timed automaton $(\Sigma_i, L_i, I_i, C_i, \rightarrow_i, Label_i, inv_i)$ for $i = 1, 2$ with $C_1 \cap C_2 = \varnothing$. The composition of $\mathcal{A}_1$ and $\mathcal{A}_2$, denoted $\mathcal{A}_1 \,||\, \mathcal{A}_2$, is the timed automaton $(\Sigma_1 \cup \Sigma_2, L_1 \times L_2, I_1 \times I_2, C_1 \cup C_2, \rightarrow, Label, inv)$ with

- $\rightarrow$ is the smallest transition relation defined by the following rules:
  - for $a \in \Sigma_1 \cap \Sigma_2$:
$$\frac{l_1 \xrightarrow{a,\alpha_1,C_1}_1 l_1', \;\; l_2 \xrightarrow{a,\alpha_2,C_2}_2 l_2'}{(l_1, l_2) \xrightarrow{a,\alpha_1 \wedge \alpha_2, C_1 \cup C_2} (l_1', l_2')}$$
  - for $a \in \Sigma_1 \setminus \Sigma_2$:
$$\frac{l_1 \xrightarrow{a,\alpha_1,C_1}_1 l_1'}{(l_1, l_2) \xrightarrow{a,\alpha_1,C_1} (l_1', l_2)}$$
  - for $a \in \Sigma_2 \setminus \Sigma_1$:
$$\frac{l_2 \xrightarrow{a,\alpha_2,C_2}_2 l_2'}{(l_1, l_2) \xrightarrow{a,\alpha_2,C_2} (l_1, l_2')}$$

- $Label(l_1, l_2) = Label_1(l_1) \cup Label_2(l_2)$
- $inv(l_1, l_2) = inv_1(l_1) \ \wedge \ inv_2(l_2)$

The invariant of a composed location is simply the conjunction of the invariants of its components. A few remarks on the transition relation are in order; the other components are defined in a straightforward manner. Two scenarios are distinguished. If an action a occurs in the alphabet of both timed automata, then a transition in one automaton can only be taken when both automata are able to do so. That is to say, in the resulting automaton a transition is obtained guarded by the conjunction of the enabling conditions of the individual timed automata (as both transitions need to be enabled). This scenario is covered by the first item. Besides, the clocks reset in the individual automata are all reset. Alternatively, if the action a is not in the common alphabet, then the automaton can take the enabled transition autonomously, i.e., without the need to synchronize with the other automaton. The cases $a \in \Sigma_1 \setminus \Sigma_2$ and $a \in \Sigma_2 \setminus \Sigma_1$ are fully symmetric and are covered by the last two items.

Example 8.8. *The parallel composition of the two timed automata at the top of Figure 8.5 results in the timed automaton at the bottom of that figure. Transitions are separately labelled with actions and an enabling condition together with a set of clocks to be reset. The reader is invited to check the construction.* *(End of example.)*

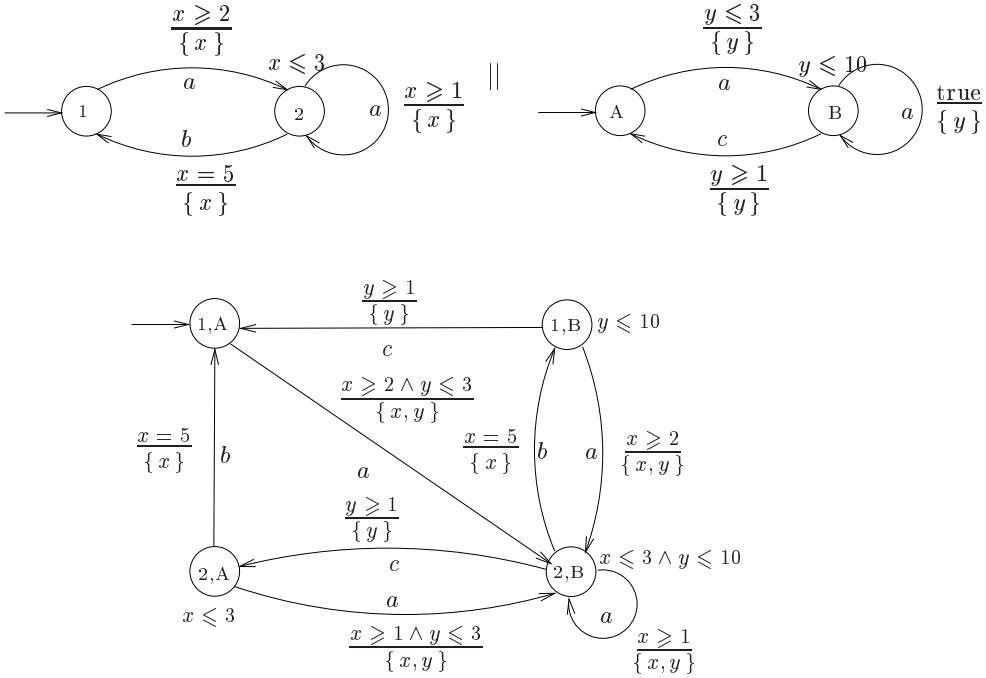


Figure 8.5: Parallel composition of two timed automata

8.5 Philips' Bounded Retransmission Protocol

To illustrate the use of timed automata we treat the compositional modeling of part of an infra-red control system (called RC6) that has been developed by Philips in the late nineties [39].

We focus on the so-called frame exchange protocol which is used to transfer bulks of data (files) such as Teletext pages between audio/video equipment and a remote control unit. Since the infra-red communication medium is rather vulnerable, data may easily be lost, e.g., if an obstacle is positioned in front of the television set. In order to avoid the (rather expensive) retransmission of entire files, files are chopped into small units, called chunks. The successful transmission of a chunk is notified to the sender by means of an acknowledgement (ack). In absence of an ack, the chunk will be retransmitted. In order to distinguish retransmitted chunks and new chunks, an alternating bit accompanies each chunk at transmission. This ensures that a receiver can ignore duplicated chunks that are transmitted by the sender due to the loss of an ack. If the number of retransmissions exceeds a certain threshold, it is assumed that there is a serious communication problem, and the sender aborts the transmission of the file. Due to this bounded number of retransmissions, the protocol is called the *Bounded Retransmission Protocol* (BRP, for short).

The timing intricacies of the BRP are twofold: (i) the sender needs to decide to retransmit a chunk if its ack has not been received in time, and (ii) the receiver needs to decide that the transmission has been aborted by the remote sender if an expected chunk is not received in time.

8.5.1 Service of the BRP

As for many data transfer protocols, the service delivered by the BRP behaves like a buffer, i.e., it reads data from one client to be delivered at another one. There are two features that make the behavior more complicated than a simple buffer. Firstly, the input is a *large file* (which is modeled as a sequence), which is delivered in small chunks. Secondly, there is a *limited amount of transmissions* for each chunk to be delivered, so we cannot guarantee its eventual successful delivery. It is assumed that during transmission, chunks will not be garbled and their order will not be changed. Thus, either an initial part of the file or the whole file is delivered. Both clients obtain an indication whether the entire file has been delivered successfully or not.

The file is input via port S_{in} as a sequence of chunks $\langle d_1, \ldots, d_n \rangle$ (cf. Figure 8.6). We assume that $n > 0$, i.e., the transmission of empty files is not considered. Ideally, each d_i is delivered on the "output" port R_{out}. Each chunk is accompanied by an indication. Thus, pairs (d_j, i_j) of chunks and indications

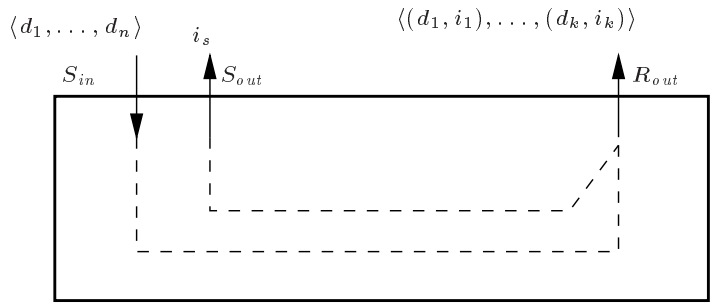


Figure 8.6: Schematic view of the external view of the BRP

are communicated via R_{out}. These indications can be either I_{FST}, I_{INC}, I_{OK}, or I_{NOK}. I_{OK} is used if d_i is the last element of the file, and I_{FST} if it is the first element *and more will follow*. All other chunks are accompanied by I_{INC}, except that a "not OK" indication (I_{NOK}) is delivered without datum if something goes wrong. Note that the receiving client does not need a "not OK" indication before delivery of the first chunk nor after delivery of the last one.

The sending client is informed via S_{out} after transmission of the whole file, or when the transmission has been aborted. Possible indications here are I_{OK}, I_{NOK}, or I_{DK}. After an I_{OK} or an I_{NOK} indication, the sender can be sure, that the receiver has the corresponding indication. A "don't know" indication I_{DK} occurs if the last chunk has not been acknowledged. In this case, either the last chunk may be lost, or the ack of the last chunk was lost. Thus, there is no way to know whether the last chunk d_n has been delivered correctly or not.

8.5.2 Modeling the BRP

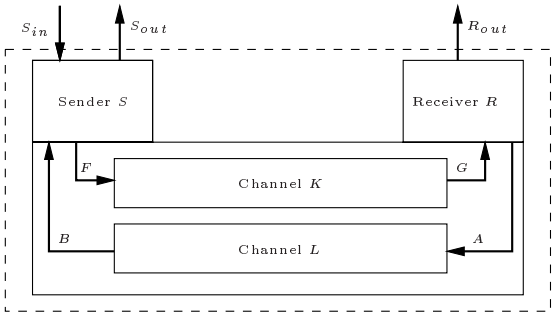


Figure 8.7: Schematic view of the internal structure of the BRP

The BRP consists of a sender S and a receiver R communicating through the lossy channels K and L (cf. Figure 8.7). The actions used for synchronizing the several components are F, G, A and B. The signatures of the parameters that are exchanged at synchronization are as follows:

Synchronization	Signature
S_{in}	$\langle d_1, \ldots, d_n \rangle$ for $n > 0$
S_{out}	$i_s \in \{ I_{OK}, I_{NOK}, I_{DK} \}$
R_{out}	$\langle (d_1, i_1), \ldots, (d_k, i_k) \rangle$ for $0 \leqslant k \leqslant n$
	$i_j \in \{ I_{FST}, I_{INC}, I_{OK}, I_{NOK} \}$ for $0 < j \leqslant k$
F, G	(b, b', ab, d_i) with $b, b', ab \in \{ 0, 1 \}$, and $0 < i \leqslant n$
A, B	ack

Modeling the Channels

Channels K and L are modeled as queues of capacity one with possible loss of messages. The timed automata of the channels are depicted in Figure 8.8. Channel K has a single clock u that is reset once a frame (i.e., a chunk and three additional bits) is received by synchronizing with the sender S (over F). Two possibilities now arise: the frame is either lost (upper transition from location *in_transit* to *start*) or is passed onto the receiver R (lower transition). The maximum latency of the channel is TD time units, for some fixed constant TD. By the invariant $u \leqslant$ TD it is guaranteed that location *in_transit* is left before the maximum latency has expired. Channel L is modeled in an analogous way.

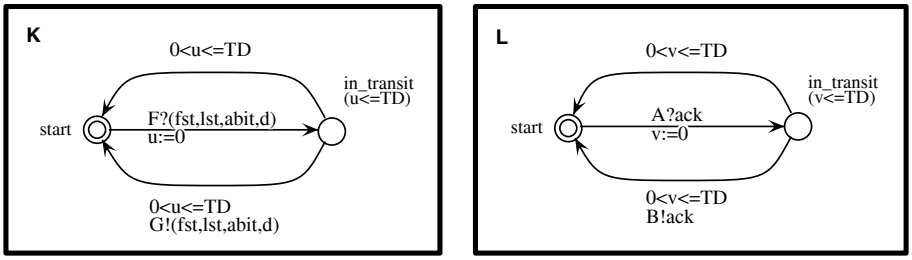


Figure 8.8: Timed automata for channels K and L.

Modeling the Sender

The sender S is modeled by the timed automaton depicted in Figure 8.9. It has four variables: clock x to keep track of the waiting time for an ack, an alternating bit $ab \in \{ 0, 1 \}$ to accompany the next chunk to be sent, the subscript i ($0 \leqslant i \leqslant n$) of the current chunk processed by S, and the number rc ($0 \leqslant rc \leqslant$ MAX) of retransmissions of the current chunk so far. Constant T1 is the maximum amount of time the sender is willing to wait for an ack, and MAX is the maximum number of retransmissions allowed.

In location *idle*, a new file is awaited to be received via S_{in}. On its receipt, i is set to one and clock x is reset. Then it starts sending the chunks one-by-

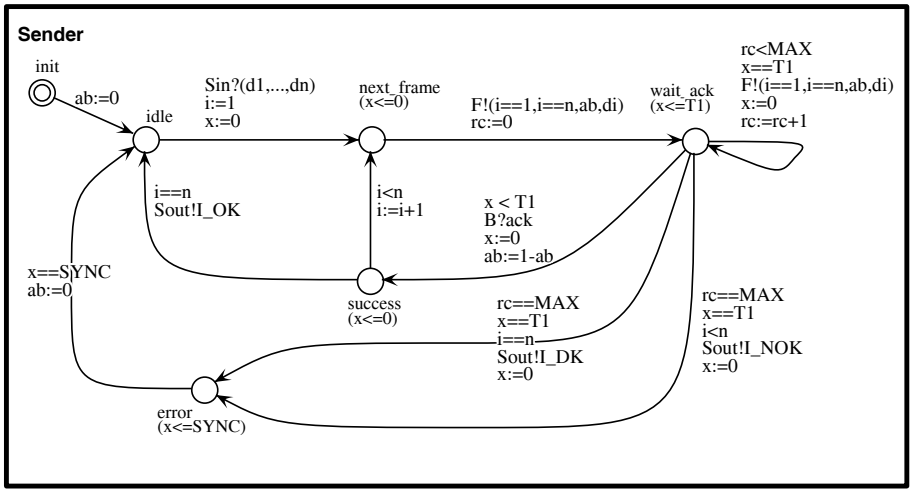


Figure 8.9: Timed automaton for sender S.

one over K (via synchronizing over F) to the receiver R. A frame (b, b', ab, d_i) consists of three bits and a datum (= chunk). Bit b indicates whether chunk d_i is the first chunk of the file (i.e., $i = 1$); bit b' indicates whether the datum is the last item of the file (i.e., $i = n$). The retransmission counter rc is reset as it is the first transmission of d_i. In location *wait_ack* there are two possibilities:

- an *ack* is received (via B) within time (i.e., $x < \mathsf{T1}$) and the sender moves to the *success* location while flipping ab
- timer x expires (i.e., $x = \mathsf{T1}$); then
  - if a retransmission is still allowed (i.e., $rc < \mathsf{MAX}$), a retransmission is initiated with the same alternating bit and counter rc is incremented
  - if the maximum number of retransmissions has been reached (i.e., $rc = \mathsf{MAX}$), the *error* location is reached and an I_{DK} or I_{NOK} indication is emitted (via S_{out}) depending on whether d_i is the last chunk or not.

  For simplicity, it is assumed that the timer x only expires if indeed no ack is still to come[2].

In all cases clock x is reset. If the last chunk has been acknowledged, S moves from location *success* to location *idle* indicating the successful transmission of the file by emitting I_{OK}. Otherwise, i is incremented while moving to location *next_frame* from which the next chunk is transmitted.

[2]This assumption requires that $\mathsf{T1} > 2 \times \mathsf{TD} + \delta$ where δ denotes the processing time in the receiver R.

Note that in most locations (such as *next_frame*, *error* and *success*) it is not allowed to delay at all. This is established by resetting clock x on all incoming transitions, and imposing the invariant $x \leqslant 0$. Secondly, we remark that after aborting the transmission of a file (i.e., location *error*) an additional delay of SYNC time units is incorporated. This delay is introduced in order to ensure that S does not start transmitting a new file before the receiver has properly reacted to the abortion. In case of abortion the alternating bit scheme is restarted.

Modeling the Receiver

The receiver is depicted in Figure 8.10. System variable $exp\_ab \in \{0, 1\}$ models the expected alternating bit. Clock z is used to keep track of the delay between the receipt of successive frames, and auxiliary clock w is used to avoid delaying in locations. Clock z is (re)set by the receiver at the arrival of each new frame. It times out if the transmission of a file has been interrupted by the sender S.

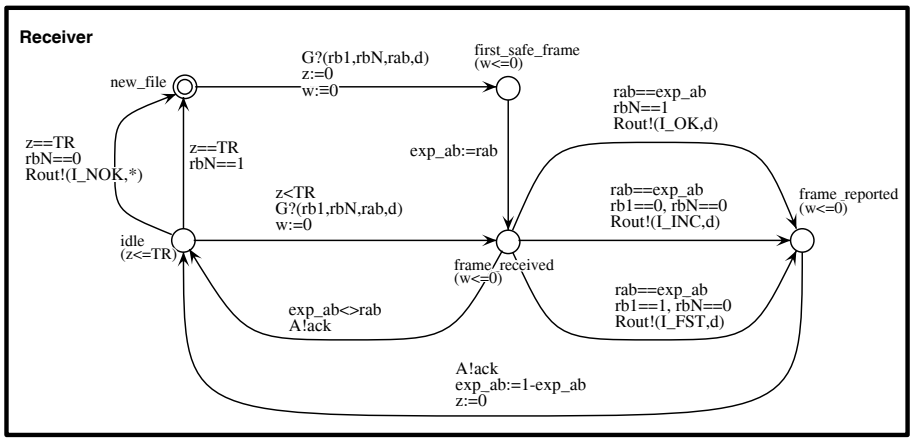


Figure 8.10: Timed automaton for receiver R.

In location *new_file*, R is waiting for the first chunk of a new file to arrive. Immediately after the receipt of such chunk, *exp_ab* is set to the just received alternating bit and R enters the location *frame_received*. If the expected alternating bit agrees with the just received alternating bit (which, due to the former assignment to *exp_ab* is always the case for the first chunk) then an appropriate indication is sent to the receiving client, an *ack* is sent via A, *exp_ab* is toggled, and clock z is reset. R is now in location *idle* and waits for the next frame to arrive. If such frame arrives in time (i.e., $z < \mathsf{TR}$) then it moves to the location *frame_received* and the above described procedure is repeated; if timer z expires (i.e., $z = \mathsf{TR}$) then in case R did not just receive the last chunk of a file an indication I_{NOK} (accompanied with an arbitrary chunk "$*$") is sent via R_{out} indicating a failure, and in case R just received the last chunk, no failure

is reported.

In most locations, no delay is allowed. This is done ensure that the sender S does not timeout if an ack is still to come, and that the receiver R does not timeout if the sender did not abort the transmission. For example, if we would allow an arbitrary delay in location *frame_received* then the sender S could generate a timeout (since it takes too long for an ack to arrive at S) while an ack generated by R is possibly still to come.

8.6 Bibliographic Notes

Timed Automata. Incorporating real-time aspects in formal modeling techniques has received a considerable attention in the last two decades. The distinction between instantaneous activities (like changing a state) and delays, which model the passage of time, originates from timed extensions of process algebras such as timed CSP [163], timed CCS [193] and ATP [144]. The resulting two-phase behaviour in which discrete phases – a state change due to some activity – and continuous phases – passage of time – has also been adopted by Alur and Dill in their original timed automata paper in 1990 [5] (journal version is published as [6]). Henzinger *et al.* [92] introduced the idea of invariants into timed automata and called this safety timed automata. Since the observation by Alur, Courcoubetis and Dill [4] that reachability properties for timed automata can be checked in a symbolic manner (cf. Chapter 9), timed automata have been the central focus of specification and verification approaches for real-time systems. Various extensions of timed automata have been considered such as, e.g., timed automata with edges equipped with deadlines (as opposed to invariants that are associated with locations) by Bornot and Sifakis [28], timed automata with drifting clocks – associating an interval to each clock that specifies the relative speed with respect to an exact reference clock – by Olivero, Sifakis and Yovine [145], and hybrid automata for describing a mixture of discrete and continous behaviour of a more general nature [128] by Maler, Manna and Pnueli. Concerning specification, for instance, D'Argenio and Brinksma [61] defined a compositional framework based on process algebra for describing safety timed automata.

Theoretical Results. Like the close link between finite-state automata and language theory – the expressive power of finite automata is equivalent to regular expressions – there is a strong connection between timed automata and timed languages. Timed languages consist of words that are infinite sequences of symbols where a real value is associated to each symbol representing its time of occurrence. Alur and Dill [6] proved that the class of languages accepted by timed automata is not closed under complementation, and hence no simple logical charaterisation of this class exists. Alur, Fix and Henzinger [8] consider a subclass of timed automata, so-called event recording automata, that

is closed under all Boolean operations including complementation. An event-recording automaton is a timed automaton that contains for every event a clock that records the time of the last occurrence of the event. Two-way timed automata [9], timed automata that can move back and forth for reading a word, are (under the restriction that an input symbol cannot be read infinitely often) also closed under all Boolean operations. Recently, Asarin, Caspi and Maler [17] defined *timed regular expressions* and proved that, à la Kleene's theorem for finite automata, its expressive power is equivalent to timed automata (in the sense of [6], i.e., without location invariants). Interesingly, intersection and renaming are essential operators for timed regular expressions to obtain this result. As in classical automata theory, the construction of automata from expressions is rather straightforward, while the other direction, from automata to expressions, is much more involved. Asarin *et al.* extend their results to timed ω-regular expressions. Alur and Dill [7] present an algorithm to check the emptiness of the language accepted by a timed automaton.

8.7 Exercises

EXERCISE 8.1. Consider a simple multi-media communication system that consists of a *Sender*, a *Receiver* and two unidirectional channels, a *VideoChannel* and an *AudioChannel*, directed from the *Sender* to the *Receiver*. For this exercise we ignore the *AudioChannel* and concentrate on the communication of video frames. If the *Receiver* receives a video frame, it processes the frame, and then plays the frame on a remote TV-set. The processing time of the *Receiver* is exactly 5 time units.

Model the simple video-transfer system for the following types of *VideoChannel*s in terms of a network of timed automata:

1. a perfect "optimal delay" channel that does neither garble nor lose a frame, and imposes a fixed latency of 10 time units to each frame;
2. a perfect "anchored jitter" channel that does neither garble nor lose a frame, and that imposes a latency to each frame such that each frame is delivered to the *Receiver* at earliest 2 time units before the perfect delay of 10 time units, and at the latest 3 time units after the time point of a perfect delay;
3. a perfect "non-anchored jitter" channel that does neither garble nor lose a frame, and that imposes a latency to each frame such that the time distance between any two successive frames that are delivered to the *Receiver* is between 8 and 13 time units;
4. an imperfect "non-anchored jitter" channel with the same timing characteristics as the previous question, but with the possibility to either lose a frame or garble a frame;
5. (More involved.) take your model of the perfect "optimal delay" channel and suppose the clock that regulates the delay of a frame does not have a fixed rate, but has a fluctuating rate between a minimal rate of $\frac{10}{13}$ ticks per time unit and

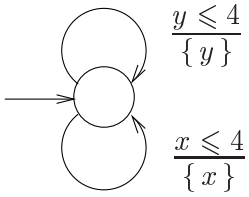
a maximal rate of $\frac{10}{8}$ ticks per time unit. Compare the behavior of this model with the behavior of your model of a perfect "non-anchored jitter" channel.

6. suppose you are now given a video communication channel that is known to never garble or lose a message, but has an unknown timing behavior; i.e., it may be one of the types perfect "optimal delay", perfect "anchored jitter" or perfect "non-anchored jitter". The only available timing information is that frames have a timing distance of 10 time units if they do not incur any jitter. Give (a set of) real-time temporal logic formulae that you could use to determine the type of channel. Assume that you have the model of the *Sender* and *Receiver* at your disposal.

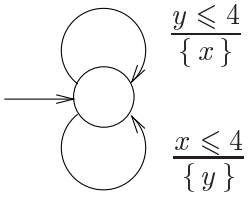
EXERCISE 8.2. Consider the following simple railway system that consists of three components: a *Train*, a *Gate* and a *Controller*. If the *Train* gets close to the *Gate* it signals this by emitting an approach signal. After a while, it passes the crossing and when it has passed the crossing it informs the gate that it has exited. The *Train* takes 5 minutes between signalling its approach and its exiting of the crossing. The *Gate* is controlled by the *Controller*. It may be lowered when it is open or raised when it is closed. Closing the *Gate* takes at most 1 minute and opening takes between 1 and 2 minutes. The *Controller* indicates to lower the *Gate* when a *Train* is approaching, and once the *Train* has exited the crossing, the *Controller* will raise the *Gate*. The *Controller* takes exactly 1 minute to respond to the approach signal, and responds to an exit signal within 1 minute.

Question: model this system as a network of timed automata. Explain your model.

EXERCISE 8.3. Consider the two timed automata depicted below.



(a)



(b)

As these automata have a single location only, the *state* of these automata can be considered as just a point in the real plane. A point (d, e) (with $d, e \geqslant 0$) then means that clock x equals d and clock y equals e.

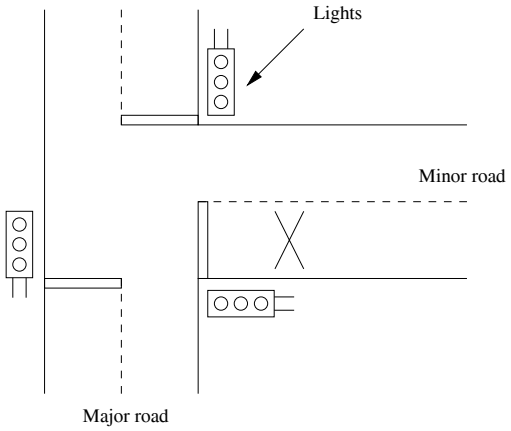
Question: determine what is the reachable state space of each of these automata. Justify your answers.

EXERCISE 8.4. Consider a system that consists of two processes: *Main* and *Int*. Process *Main* increments the counter *count* as long as the global boolean variable *flag* is false. When *flag* is true, the process *Main* decrements *count*. When *count* reaches zero, process *Main* jumps to its final location. Process *Main* performs its actions once every $[L, U]$ time-units with $U \geqslant L$. The only purpose of process *Int* is to set the variable *flag* to true (once) within maximally W time-units.

Questions:

1. Model the informally specified system as a network of timed automata.
2. Determine the maximal amount of time it takes before process *Main* reaches its final location. Justify your answer.

EXERCISE 8.5. A control system must ensure the safe and correct functioning of a set of traffic lights at a T-junction between a major and a minor road. The lights will be set on green on the major road and red on the minor road unless a vehicle is detected by a sensor in the road just before the lights on the minor road. In this case the lights will be switchable in the standard manner and allow traffic to leave the minor road. After a suitable interval the lights will revert to their default position to allow traffic to flow on the major road again. Once a vehicle is detected the sensor will be disabled until the minor-road lights are set to red again. A sketch of the T-junction is provided below.

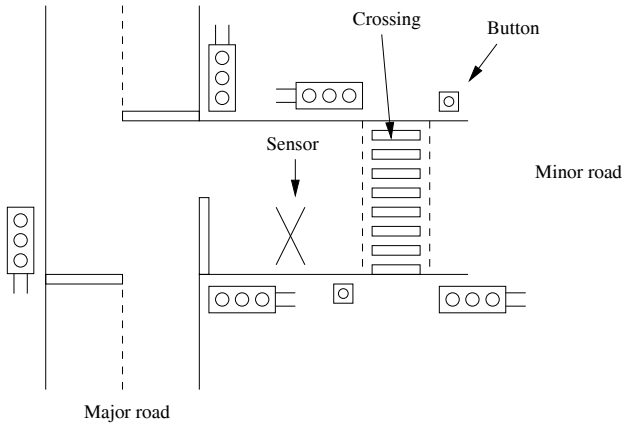


Questions:

1. First we ignore all timing issues involved and concentrate on the qualitative aspects of the behavior of the traffic lights. Model the above system as a network of (timed) automata. For convenience, you may assume that the two major-road lights are fully synchronized and can be modeled as a single light. Complement your system model by adding a process that regulates the arrival of cars in the minor road.
2. Adopt your model so as to incorporate the following timing constraints. Deal with each timing constraint separately so as reduce the complexity. Indicate for each timing constraint the necessary adaptations to your untimed model:
   (a) a minor-road light stays on green for 30 seconds
   (b) all interim lights stay on for 5 seconds
   (c) there is a one second delay between switching one light off and another on (e.g. switching from red to amber)
   (d) the major-road lights must be on green for at least 30 seconds in each cycle
   (e) (More involved.) but must respond to the sensor immediately after that.

(*Hint:* use urgent locations and urgent channels whenever appropriate; in case of urgent channels, make sure there are no timing constraints on the communication via that channel, and that the location invariant of locations where such transitions emanate are non-restrictive.)

3. We extend the T-junction in the following way. Suppose there is a pedestrian crossing a short distance down the minor road but beyond the sensor. There is a button on each side of the road for pedestrians to indicate they wish to cross. The crossing should only allow people to cross when the 'minor lights' are set to red in order to minimize waiting times for traffic on the minor road. The new situation is sketched below.



Extend your timed model of the previous question in order to cope with this new situation.

4. Does the crossing indeed only allows pedestrians to cross when the 'minor lights' are set to red?

Chapter 9

Timed CTL

Chapter 10

Model-Checking Timed CTL

Chapter 11

Symbolic Model Checking

state-space explosion?

This chapter deals with CTL model checking using symbolic representations of boolean functions. Reduced ordered binary decision diagrams are introduced as such representations, and various operations on these data structures are discussed. It is shown how these data structures can be employed for CTL model checking.

11.1 Introduction

The various algorithms for model checking, like for PLTL and CTL, all are based on a system description in terms of a Kripke structure $\mathcal{K}$. This structure needs to be kept in memory in order to perform the necessary computations. Different types of data structures can be employed to store the model $\mathcal{K}$ in computer memory. Consider $\mathcal{K} = (S, I, R, Label)$ with $S = \{ s_1, \ldots, s_N \}$ such that states are labeled with atomic propositions $a_1, \ldots, a_K$, say. A simple representation of this Kripke structure is to identify the states by numbers, represent the set of initial states I as boolean vector $\underline{i}$, use a boolean matrix $\mathbf{R}$ of cardinality $N{\cdot}N$ to represent the transition relation R, and a boolean matrix $\mathbf{L}$ of cardinality $N{\cdot}K$ to represent the state-labeling *Label*. Bit-vector $\underline{i}$ represents the *characteristic function* for I, i.e., $\underline{i}(k) = 1$ if state $k \in I$, and equals 0 otherwise. Similarly, $\mathbf{R}$ and $\mathbf{L}$ are the characteristic functions of R and *Label*, respectively:

$$\mathbf{R}(s_i, s_j) \quad = \quad \begin{cases} 1 & \text{if } s_j \in R(s_i) \\ 0 & \text{otherwise} \end{cases}$$

and

$$\mathbf{L}(s_i, a_j) \quad = \quad \begin{cases} 1 & \text{if } a_j \in Label(s_i) \\ 0 & \text{otherwise} \end{cases}$$

As the matrices $\mathbf{R}$ and $\mathbf{L}$ typically contain a relatively high number of elements that are equal to zero, data structures for sparse matrices such as (arrays of) linked lists can be used. The same applies to the bit-vector to represent I. Since all components of a Kripke structure are explicitly stored, this scheme is referred to as an *explicit* state-space representation.

Example 11.1. *Consider the Kripke structure depicted in Figure 11.1. It*

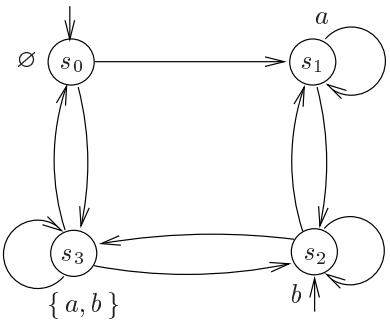


Figure 11.1: An example Kripke structure

consists of four states, named s_0 through s_3, and has as labels propositions of the set $\{a, b\}$. Suppose we identify state s_i by number i, proposition b by 0, and a by 1. The vector $\underline{i}$ representing the set of initial states, and the matrices $\mathbf{R}$ *and* $\mathbf{L}$ *are given by:*

$$\underline{i} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \quad \textit{and} \quad \mathbf{R} = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad \textit{and} \quad \mathbf{L} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}$$

For instance, $\underline{i}(0) = 1$ represents that s_0 is an initial state, $\mathbf{R}(1,2) = 1$ represents that state s_2 is a direct successor of s_1. $\mathbf{L}(2,0) = 1$ represents the fact that state s_2 is labeled by b while $\mathbf{L}(2,1) = 0$ means that s_2 is not labeled with a. *(End of example.)*

The total space consumption of this state space representation is $N{\cdot}(N{+}K){+}N$ bits. The basic concept that underlies symbolic model checking is to replace the explicit state-space representation by a symbolic, and, in most cases, more compact representation. Although the term "symbolic" in principle refers to any technique to store states and transitions symbolically, often a particular method is referred to, namely the use of so-called *binary decision diagrams*, BDDs for short. These data structures have originally been used by Akers [2, 3]

and Lee [121] to represent boolean functions – such as hardware circuits – in a compact way. As decision diagrams are a very natural data structure, it is difficult to track their origin. Observe that the representation of Kripke structures above is based on boolean functions, in particular, boolean vectors (functions with one parameter) and matrices (functions with two parameters). In addition, it turns out that all operations needed for CTL model checking such as set manipulations and fixed-point computations can efficiently be supported by BDDs. In particular, the set $Sat(\Phi)$ of states satisfying Φ that is central to verifying CTL-formulae can be represented in a (mostly) very compact way.

Although BDDs cannot guarantee to avoid state-space explosion in all cases, they provide a compact representation for several systems, allowing these systems to be verified — systems that would be impossible to handle using explicit state enumeration methods. Experience shows that in particular synchronous hardware circuits are well-suited to be represented using BDDs in a compact way. All basic operations (such as set manipulations or projection) on BDDs have a low time complexity, and are relatively easy to implement. In addition, BDDs are applicable to all finite Kripke structures, and are not tailored to a specific set of such structures.

11.2 Representing Boolean Functions

11.2.1 Kripke Structures as Boolean Functions

To introduce the use of binary decision diagrams, let us consider how Kripke structures can be encoded using boolean functions. Consider again $(S, I, R, Label)$ with N states. Assume that each state is uniquely labeled, i.e., for any states s, s' if $Label(s) = Label(s')$ then s equals s'. Note that this assumption does not impose any restriction: each Kripke structure can be modified by, for instance, extending the set of atomic proposition with proposition at_s for any state s, and adding the new atomic proposition at_s to the labeling of state s only, i.e., $at_s \in Label(s)$, and $at_s \notin Label(s')$ for any state s' different from s. Clearly, then for any states s, s', $Label(s) = Label(s')$ implies $s = s'$. Suppose that after this possible transformation, states are uniquely labeled with the atomic propositions $a_1, \ldots, a_K$, and assume there is a fixed total ordering on these propositions such that $a_1 < a_2 < \ldots < a_K$.

A state will be represented by a bit-vector of length K. More precisely, state s is represented by $[\![s]\!] = \langle b_1, b_2, \ldots, b_K \rangle$ such that b_i equals 1 if atomic proposition $a_i \in Label(s)$ and 0 otherwise. Alternatively, each state is represented by a *boolean function* over the boolean variables x_1 through x_K as follows:

$$x_1^* \ \wedge \ x_2^* \ \wedge \ \ldots \ \wedge \ x_K^*$$

where the term x_i^* equals x_i if $a_i \in \mathit{Label}(s)$, and $\neg\, x_i$ otherwise. Formally, each state s is described by a boolean function f_s say, that takes K boolean variables and K boolean values as argument such that:

$$f_s(x_1, \ldots, x_K; b_1, \ldots b_K) = x_1^* \ \wedge \ \ldots \ \wedge \ x_K^*$$

where x_i^* equals x_i if b_i equals one, and $\neg\, x_i$ otherwise. For example, when we assume that $b < a$ in the previous example, then for state s_2 we have $[\![\, s_2 \,]\!] = \langle 1, 0 \rangle$, or $f_{s_2}(x_1, x_2; 1, 0) = x_1 \wedge \neg\, x_2$.

The set of initial states I can be considered as an unary relation over the set of states, returning 1 if a state is initial, and 0 otherwise. The *characteristic function* of I, denoted f_I, may be represented by a simple truth table, or, alternatively, by a boolean function. For instance, the following boolean expression represents the characteristic function corresponding to $I = \{\, s_1, \ldots, s_n \,\}$:

$$f_{s_1}(x_1, \ldots, x_K; b_1, \ldots, b_K) \ \vee \ \ldots \ \vee \ f_{s_n}(x_1, \ldots, x_K; b_1, \ldots, b_K)$$

Note that this expression is in disjunctive normal form.

The successor relation R is a binary relation over $\{\, 0, 1 \,\}^K$, i.e., over bit-vectors of length K, such that $[\![\, s \,]\!]$ is related to $[\![\, s' \,]\!]$ if and only if $(s, s') \in R$. The characteristic function of R, denoted f_R, may, for instance, be described by a truth table, or, alternatively, by a boolean function in disjunctive or conjunctive normal form. For instance, the following boolean expression represents that s_2 and s_3 are both direct successors of s_1 in our running example, i.e., $R = \{\, (s_1, s_2), (s_1, s_3) \,\}$:

$$\begin{array}{ll} & (f_{s_1}(x_1, x_2; b_1, b_2) \ \wedge \ f_{s_2}(x_1', x_2'; b_1, b_2)) \\ \vee & (f_{s_1}(x_1, x_2; b_1, b_2) \ \wedge \ f_{s_3}(x_1', x_2'; b_1, b_2)) \end{array}$$

Here, the boolean variables x_1 and x_2 are used to encode the source state of a single transition while their primed variants are used to encode the target state. Note that this expression is also in disjunctive normal form.

There is no need to represent the function *Label* explicitly anymore as the encoding of states as bit-vectors, or equivalently, as boolean functions, is implicitly based on this labeling.

Example 11.2. Consider again the Kripke structure of Figure 11.1. First observe that in this example all states are uniquely labeled, so no additional atomic propositions have to be considered. Assume the atomic propositions in $\{\, a, b \,\}$ *are ordered as* $b < a$*. The states are then represented by:*

state	*bit-vector*	*boolean function*
s_0	$\langle 0,0 \rangle$	$\neg x_1 \wedge \neg x_2$
s_1	$\langle 0,1 \rangle$	$\neg x_1 \wedge x_2$
s_2	$\langle 1,0 \rangle$	$x_1 \wedge \neg x_2$
s_3	$\langle 1,1 \rangle$	$x_1 \wedge x_2$

The set of initial states $I = \{ s_0, s_2 \}$ *is represented by the truth table such that* $f_I(\langle 0,0 \rangle) = 1$ *and* $f_I(\langle 1,0 \rangle) = 1$ *and 0 otherwise, or equivalently, by* $f_I(x_1, x_2) = (\neg x_1 \wedge \neg x_2) \vee (x_1 \wedge \neg x_2)$. *The latter can be simplified to* $\neg x_2$ *by straightforward formula manipulation.*

The characteristic function of the successor relation R *is listed by its truth table:*

f_R	$\langle 0,0 \rangle$	$\langle 0,1 \rangle$	$\langle 1,0 \rangle$	$\langle 1,1 \rangle$
$\langle 0,0 \rangle$	*0*	*1*	*0*	*1*
$\langle 0,1 \rangle$	*0*	*1*	*1*	*0*
$\langle 1,0 \rangle$	*0*	*1*	*1*	*1*
$\langle 1,1 \rangle$	*1*	*0*	*1*	*1*

For example, the fact that state s_1 *is a successor of* s_2 *follows from* $f_R([\![s_2]\!], [\![s_1]\!]) = 1$, *that is,* $f_R(\langle 1,0 \rangle, \langle 0,1 \rangle) = 1$. *A representation of* f_R *in disjunctive normal form can simply be obtained from the above truth table by "listing" all entries equal to one in the table, e.g., in a row-wise fashion starting at the uppermost row:*

$$
\begin{aligned}
f_R(x_1, x_2, x_1', x_2') = \quad & (\neg x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge x_2') \\
\vee \quad & (\neg x_1 \wedge \neg x_2 \wedge x_1' \wedge x_2') \\
\vee \quad & (\neg x_1 \wedge x_2 \wedge x_1' \wedge \neg x_2') \\
\vee \quad & \ldots \\
\vee \quad & (x_1 \wedge x_2 \wedge x_1' \wedge x_2')
\end{aligned}
$$

We have $f_R(x_1, x_2, x_1', x_2') = 1$ *if and only if* $(s, s') \in R$ *with* $[\![s]\!] = \langle x_1, x_2 \rangle$ *and* $[\![s']\!] = \langle x_1', x_2' \rangle$. *The first disjunct represents that* s_1 *is a successor of* s_0, *the second disjunct that* s_3 *is a successor of* s_0, *and so on.* *(End of example.)*

11.2.2 Binary Decision Trees

Boolean functions like the transition relation before can be either represented by a truth table, listing all possible inputs and corresponding output, or a boolean expression in conjunctive or disjunctive normal form. Alternatively, a rooted *binary decision tree* (BDT, for short) can be used. A BDT is a binary tree in which each leaf v is labeled with a boolean constant *val*(v) which is either 0 or 1. Other vertices are augmented with a boolean variable *var*(v) and have two children: *left*(v) and *right*(v). The labeling of vertices with boolean variables

is done in such a way that on each path from root to a leaf, the variables are encountered in the same order.[1] The left child of a non-leaf corresponds to the case that the value of $var(v)$ equals zero; the right to the case that $var(v)$ equals one:

if $var(v) = 0$ **then** goto $left(v)$ **else** goto $right(v)$ **fi**

Leafs are often called terminals and other vertices are referred to as nonterminals. In the representation of a BDT, edges to left children (zero) are drawn as dotted lines, while edges to right children (one) are represented by solid lines. As BDTs are drawn from top to the leafs, the direction from edges is clear and therefore edges are drawn as lines rather than as arrows.

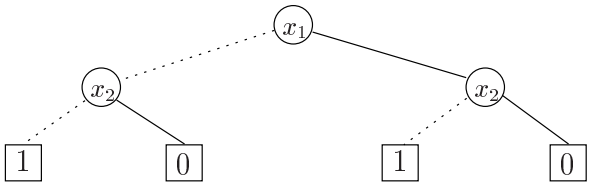


Figure 11.2: A binary decision tree representing f_I

Example 11.3. The binary decision tree that represents the set of initial states of the Kripke structure in Figure 11.1 is depicted in Figure 11.2. The ordering of the variables on any path from the root to any leaf is $x_1 < x_2$. The binary decision tree of the characteristic function $f_R(x_1, x_2, x'_1, x'_2)$ of the previous example is given in Figure 11.3. The ordering of the variables on any path from root to any leaf is $x_1 < x_2 < x'_1 < x'_2$. (End of example.)

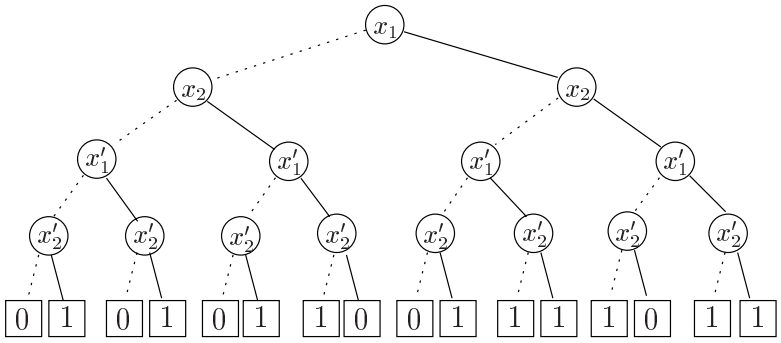


Figure 11.3: A binary decision tree representing f_R

The function value $f(x_1, \ldots, x_n)$ for a given assignment to the arguments x_1 through x_n is determined by traversing the BDT starting from the root, branch-

[1] Strictly speaking, we should call these data structures *ordered* BDTs due to this property. Originally, BDTs do not obey this ordering requirement.

ing at each vertex based on the assigned value of the variable that labels the vertex. This process is continued until a leaf is reached. The value of the thus encountered leaf is the function value. For instance, to determine $f_R(\langle 1,0\rangle, \langle 0,1\rangle)$ in Figure 11.3, we instantiate the variables $x_1 := 1$, $x_2 := 0$, $x_1' := 0$ and $x_2' := 1$ and traverse the tree accordingly from root to leaf. The resulting path is: go right, go left, go left, and, finally, go right. This results in a terminal vertex with value 1. We thus establish that $\langle 0,1\rangle$ (state s_1) is a successor of $\langle 1,0\rangle$ (state s_2) as the function value $f_R(\langle 1,0\rangle, \langle 0,1\rangle)$ equals one. This is also denoted as

$$f_R[x_1 := 1, x_2 := 0, x_1' := 0, x_2' := 1] = 1.$$

In a similar way, we can establish in Figure 11.2 that $f_I(\langle 0,0\rangle)$ equals one confirming that $\langle 0,0\rangle$ (state s_0) is an initial state.

A few remarks on BDTs are in order. BDTs are not very compact; in fact the number of leafs is identical to the size of the truth table of f. That is, a BDT for function f with n arguments has 2^n leafs. A major reason for this exponential size is that BDTs contain quite some redundancy. For instance, rather than representing all leafs separately, all leafs with value one (zero) could be collapsed into a single leaf, and the pointers of vertices to these leafs (in Figure 11.2 the vertices labeled with x_2) adjusted accordingly. A similar scheme can be employed for other subtrees, e.g., the vertices labeled with x_2 in Figure 11.2 have isomorphic subtrees and could be collapsed. In the next section, we will see that the avoidance of such redundancies is one of the key ideas behind BDDs. Finally, remark that the size of a BDT does not change if we would change the order of the variables occurring in the tree when traversing from the root to a leaf. For instance, one could change the current variable ordering $x_1 < x_2 < x_1' < x_2'$ in Figure 11.3 into $x_1 < x_1' < x_2 < x_2'$ without affecting the size of the BDT. A similar effect occurs for the truth table representation. It is left to the reader to verify this.

Given a BDT, one may wonder which boolean function it represents. Each vertex in a BDT in fact represents a boolean function:

Definition 11.1. (**Function represented by a BDT-vertex**)
Let B be a BDT and v a vertex in B. The boolean function $f_{\mathsf{B}}(v)$ represented by vertex v is defined as follows:

- for v a terminal vertex $f_{\mathsf{B}}(v) = \mathit{val}(v)$, and
- for v a nonterminal vertex:

$$f_{\mathsf{B}}(v) = (\mathit{var}(v) \ \wedge \ f_{\mathsf{B}}(\mathit{right}(v))) \ \vee \ (\neg \mathit{var}(v) \ \wedge \ f_{\mathsf{B}}(\mathit{left}(v))).$$

The function represented by a BDT is $f_{\mathsf{B}}(v)$ where v is the root of the BDT, i.e., the top-vertex without incoming edges. In the sequel we denote the function

denoted by B simply by f_{B}. The value of f_{B} for assignment $x_1 := b_1, \ldots, x_n := b_n$, for $b_i \in \{0, 1\}$, equals $f_{\mathsf{B}}[x_1 := b_1, \ldots, x_n := b_n]$.

Example 11.4. The boolean function represented by the BDT in Figure 11.2 is defined by:

$$(\neg x_1 \wedge \neg x_2 \wedge 1) \vee (\neg x_1 \wedge x_2 \wedge 0) \vee (x_1 \wedge \neg x_2 \wedge 1) \vee (x_1 \wedge x_2 \wedge 0)$$

It is straightforward to see that this represents $f_I(x_1, x_2)$. The boolean function represented by the BDT in Figure 11.3 is obtained in a similar way:

$$\begin{array}{ll} & (\neg x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge \neg x_2' \wedge 0) \\ \vee & (\neg x_1 \wedge \neg x_2 \wedge \neg x_1' \wedge x_2' \wedge 1) \\ \vee & (\neg x_1 \wedge \neg x_2 \wedge x_1' \wedge \neg x_2' \wedge 0) \\ \vee & (\neg x_1 \wedge \neg x_2 \wedge x_1' \wedge x_2' \wedge 1) \\ \vee & \ldots\ldots \\ \vee & (x_1 \wedge x_2 \wedge x_1' \wedge x_2' \wedge 1) \end{array}$$

which corresponds to the characteristic function $f_R(x_1, x_2, x_1', x_2')$ of the successor relation R. (End of example.)

Definition 11.1 is inspired by the so-called *Shannon expansion* for boolean functions. This expansion allows to rewrite function $f : \{0, 1\}^n \longrightarrow \{0, 1\}$ by:

$$f(x_1, \ldots, x_n) = (x_1 \wedge f[x_1 := 1]) \vee (\neg x_1 \wedge f[x_1 := 0])$$

where $f[x_1 := 1]$ stands for the function $f(1, x_2, \ldots, x_n)$ and $f[x_1 := 1]$ is a shorthand for $f(0, x_2, \ldots, x_n)$. Alternatively,

$$f(x_1, \ldots, x_n) = \textbf{if } x_1 \textbf{ then } f[x_1 := 1] \textbf{ else } f[x_1 := 0] \textbf{ fi}$$

It is, of course, possible to expand f not only with respect to x_1, but with respect to any variable x_1 through x_n. In general, expansion with respect to variable x_i is:

$$f(x_1, \ldots, x_n) = (x_i \wedge f[x_i := 1]) \vee (\neg x_i \wedge f[x_i := 0])$$

11.3 Reduced Ordered Binary Decision Diagrams

11.3.1 Binary Decision Diagrams

In a binary decision *diagram* (BDD) the redundancy that is present in a decision tree is reduced. A binary decision diagram is a BDT in which isomorphic subtrees are collapsed and redundant vertices – also called "don't care" vertices – are omitted. Stated in terms of boolean expressions, BDDs are representations in which equivalent boolean sub-expressions are uniquely represented. A vertex is considered redundant if the truth variable of its boolean variable is irrelevant for the truth value of the function represented by the BDD. For instance, the vertex labeled with x_1 in Figure 11.2 is a redundant vertex: irrespective of selecting its left or right branch, the function value is only determined by x_2. This conforms to the fact that the boolean function it represents, $(\neg x_1 \ \wedge \ \neg x_2) \ \vee \ (x_1 \ \wedge \ \neg x_2)$ is equivalent to $\neg x_2$.

The resulting structure is no longer a tree, but a directed acyclic graph.

Definition 11.2. (**Rooted directed acyclic graph**)
A *directed acyclic* graph (dag, for short) G is a pair (V, E) where V is a set of vertices and $E \subseteq V \times V$ a set of edges, such that G does not contain any cycles [2]. A dag is *rooted* if it contains a single vertex $v \in V$ without incoming edges, i.e., $\{ v' \in V \mid (v', v) \in E \} = \varnothing$.

Let $X = \{ x_1, \ldots, x_n \}$ be a set of boolean variables and $<$ a fixed total order on X such that $x_i < x_j$ or $x_j < x_i$ for all i, j $(i \neq j)$. The pair $\langle X, < \rangle$ is a totally ordered set.

Definition 11.3. (**Ordered binary decision diagram**)
An *ordered binary decision diagram* (OBDD, for short) over $\langle X, < \rangle$ is a rooted dag with non-empty vertex-set V containing two disjoint types of vertices:

- each *nonterminal* vertex v is labeled by a boolean variable $\mathit{var}(v) \in X$ and has two children $\mathit{left}(v), \mathit{right}(v) \in V$
- each *terminal* vertex v is labeled by a boolean value $\mathit{val}(v)$,

such that for each nonterminal vertex $v \in V$ and each vertex w:

$$w \in \{ \mathit{left}(v), \mathit{right}(v) \} \ \Rightarrow \ (\mathit{var}(v) < \mathit{var}(w) \ \vee \ w \text{ is a terminal}).$$

[2] A cycle is a finite path $v_1 \ldots v_n$ with $v_i \in V$, $n > 1$, $v_n = v_1$, and $(v_i, v_{i+1}) \in E$ for $i < n$.

The functions *left* and *right* define the edges of the graph, *left* defines the dotted edges while *right* defines the solid edges. The (ordering) constraint on the labeling of the nonterminals requires that on any path from the root to a terminal vertex, the variables respect the ordering $<$. This constraint also guarantees that an OBDD is a directed acyclic graph: as the order strictly decreases when going from a nonterminal vertex to its descendants, there cannot be a cycle. For nonterminal v, the edge from v to $\mathit{left}(v)$ represents the case that $\mathit{var}(v) = \text{false}$ (=0); the edge from v to $\mathit{right}(v)$ represents the case $\mathit{var}(v) = \text{true}$ (=1). Note that, by definition, each BDT is an OBDD.

Example 11.5. *Let $X = \{ x_1, x_2, x_1', x_2' \}$ with ordering $x_1 < x_2 < x_1' < x_2'$. An example OBDD of the characteristic function f_R over $\langle X, < \rangle$ of our running example is given in Figure 11.4. Observe that compared to the BDT of Figure 11.3 several isomorphic subtrees rooted at vertices labeled with x_1' are collapsed.* *(End of example.)*

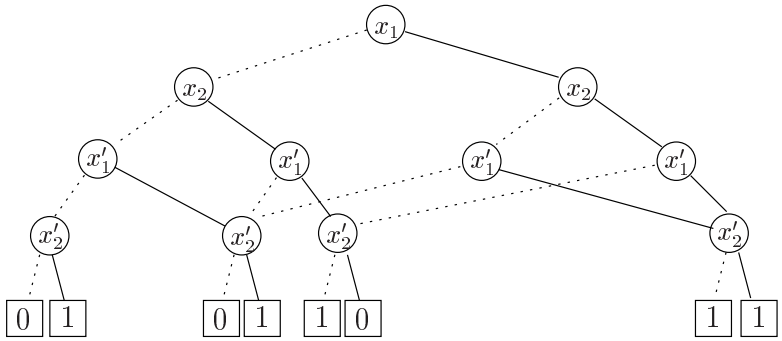


Figure 11.4: A binary decision diagram representing f_R

Typically, OBDDs that are identical up to, e.g., mirroring of certain subtrees are not distinguished. Such structures are called isomorphic. OBDDs B and B' over $\langle X, < \rangle$ are *isomorphic* if and only if their root vertices are isomorphic. Vertices v in B and w in B' are isomorphic, denoted $v \cong w$, if and only if there exists a bijection, H, say, from the vertices of B to the vertices of B' such that:

1. if v is a terminal, then $H(v) = w$ is a terminal with $\mathit{val}(v) = \mathit{val}(w)$
2. if v is a nonterminal, then $H(v) = w$ is a nonterminal such that $\mathit{var}(v) = \mathit{var}(w)$, $H(\mathit{left}(v)) = \mathit{left}(H(v))$ and $H(\mathit{right}(v)) = \mathit{right}(H(v))$.

Testing whether two ROBDDs are isomorphic can be done in linear time due to the labels (0 and 1) of the edges.

11.3.2 Reduced OBDDs

An ordered BDD may still contain some redundancy. For instance, the two subtrees of the leftmost nonterminal labeled with x'_1 in the OBDD in Figure 11.4 are isomorphic and could be collapsed. In fact, the value of variable x'_1 in this case is irrelevant, and the nonterminal vertex can be safely removed. In order to obtain more compact representations, *reduced* OBDDs are considered.

Definition 11.4. (**Reduced OBDD**)
OBDD B over $\langle X, < \rangle$ is called *reduced* if the following conditions hold:

1. for each terminal v, w: $(\mathit{val}(v) = \mathit{val}(w)) \Rightarrow v = w$
2. for each nonterminal v: $\mathit{left}(v) \neq \mathit{right}(v)$
3. for each nonterminal v, w:

$$(\mathit{var}(v) = \mathit{var}(w) \wedge \mathit{right}(v) \cong \mathit{right}(w) \wedge \mathit{left}(v) \cong \mathit{left}(w)) \Rightarrow v = w$$

The first constraint does not allow identical terminal vertices, thus at most two terminal vertices are allowed. The second constraint states that no nonterminal vertex has identical left and right children and the last constraint forbids vertices to denote isomorphic sub-dags. A reduced OBDD is abbreviated as ROBDD. The boolean function represented by a ROBDD is obtained in the same way as for BDTs, cf. Definition 11.1.

Example 11.6. The BDD of Figure 11.4 is an OBDD over $\langle X, < \rangle$, but not an ROBDD. For instance, the two subtrees rooted at the vertex labeled with x'_1 in the leftmost subtree are isomorphic, and the top-vertices of these subtrees violate the last constraint of Definition 11.4. (End of example.)

Definition 11.4 suggests three possible ways to transform a given OBDD into a reduced form:

- Removal of duplicate terminals: if an OBDD contains more than one terminal labeled 0 (or 1), redirect all edges that point to such vertex to one of them, and remove the obsolete terminal vertices, cf. Figure 11.5(a). Accordingly, a single terminal vertex with 0 (and 1) remains.
- Removal of redundant ("don't care") nonterminals: if both outgoing edges of a nonterminal vertex v point to the same vertex w, then eliminate v and redirect all its incoming edges to w, cf. Figure 11.5(b).

- Removal of duplicate nonterminals: if two distinct nonterminal vertices v and w are equally labeled and are roots of isomorphic ROBDDs, then eliminate v (or w), and redirect all incoming edges to the other one, cf. Figure 11.5(c).

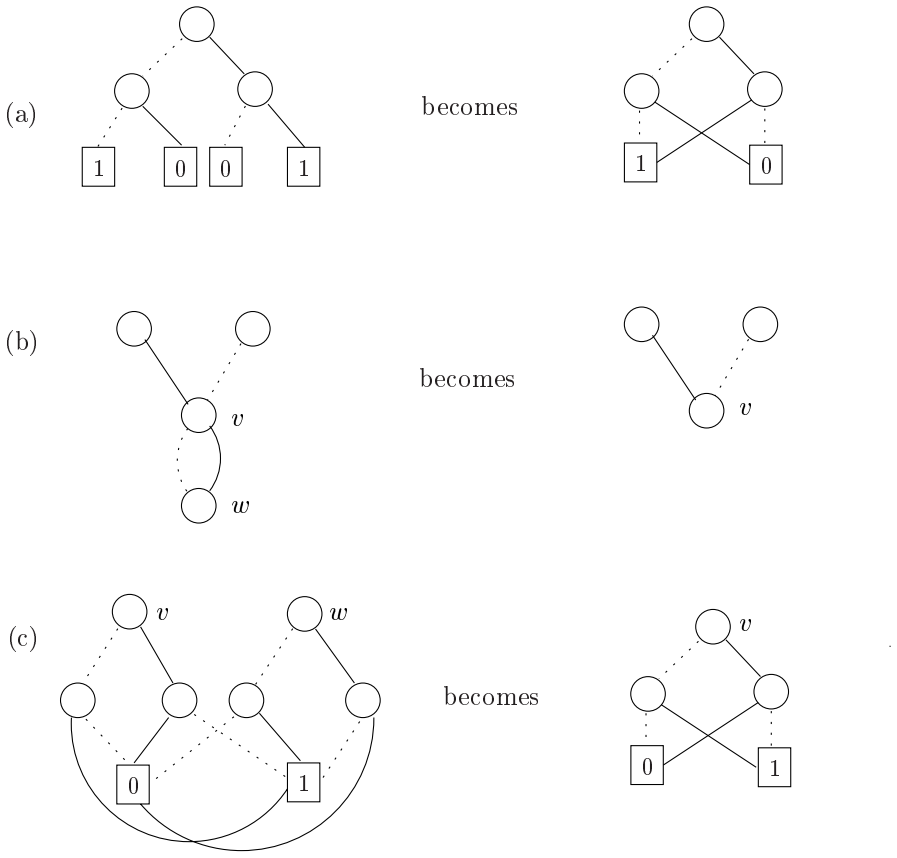


Figure 11.5: Steps to transform an OBDD into reduced form

For convenience, sometimes the convention is adopted to omit the 0-terminals and all edges pointing to it. An OBDD is reduced if none of the possible reduction steps above can be applied anymore. The transformation of an OBDD into an ROBDD can be done by a bottom-up traversal of the directed graph in time that is linear in the number of vertices in the OBDD. This transformation is further described below. A natural question that arises when applying the above transformations is whether the resulting ROBDD will be unique, or whether there will be several possible resulting ROBDDs representing the same boolean function? This question can be answered affirmative by the following result which is due to Bryant [35]. Given a total ordering on the boolean variables, there exists a unique ROBDD (up to isomorphism) that represents a boolean function:

Theorem 11.1.
Let $X = \{ x_1, \ldots, x_n \}$ *be a set of boolean variables and* $<$ *a total ordering on*

X. For ROBDDs B *and* B' *over* $\langle X, < \rangle$ *we have:*

$$(f_{\mathsf{B}} = f_{\mathsf{B}'}) \Rightarrow \mathsf{B} \text{ and } B' \text{ are isomorphic.}$$

Proof: This result is proven by induction on the number of boolean variables n. Let $f\mathsf{B} = f_{\mathsf{B}'} = f$ and assume $x_1 < \ldots < x_n$.

Base case: if n=0, then either $f = 0$ or $f = 1$. It is not difficult to see that the only ROBDD representing $f = 0$ consists of a single vertex labeled with value 0. Similarly, the only ROBDD representing $f = 1$ consists of a single vertex labeled with value 1. Clearly, if B and B' represent the same constant function, they are isomorphic.

Induction step: Suppose that for any boolean function with less than k arguments, $k > 0$, the result holds, and consider $f(x_1, \ldots, x_k)$. Let i be the minimum index such that f depends on x_i. Let $f_0 = f[x_i := 0]$ and $f_1 = f[x_i := 1]$. As f_0 and f_1 both depend on less than k arguments, they are – by the induction hypothesis – represented by isomorphic ROBDDs. Let B and B' both represent f, v be a nonterminal in B such that $\mathit{val}(v) = x_i$ and v' be a nonterminal in B' with $\mathit{val}(v') = x_i$. Note that $f_{\mathsf{B}}(v) = f_{\mathsf{B}'}(v') = f$, as f is not depending on x_1 through x_{i-1}. The subgraphs rooted by $\mathit{left}(v)$ and $\mathit{left}(v')$ both denote f_0, and, by the induction hypothesis, there is an isomorphism, H_0, say, relating their vertices. By a symmetric argument, H_1 is an isomorphism relating $\mathit{right}(v)$ and $\mathit{right}(v')$. Let H be a mapping from vertices in B to vertices in B' defined by:

$$H(w) = \begin{cases} v' & \text{if } w = v \\ H_0(w) & \text{if } w \text{ is a vertex in } \mathit{left}(v) \\ H_1(w) & \text{if } w \text{ is a vertex in } \mathit{right}(v) \end{cases}$$

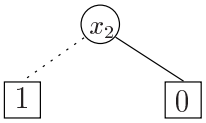
It is straightforward to see that H is a function that maps terminals in B onto isomorphic terminals in B', and nonterminals in B onto isomorphic nonterminals in B'. It remains to be checked that B and B' have isomorphic roots. Note that B (and B') contain only one vertex labeled with x_i: if there would be another vertex than v (and v') then the subgraphs rooted at these vertices are isomorphic, contradicting that B (and B') are reduced. Suppose there is some vertex w in B with $\mathit{var}(w) < \mathit{var}(v) = x_i$, and no other vertex u, say, such that $\mathit{var}(w) < \mathit{var}(u) < \mathit{var}(v)$. As x_i is the smallest variable (in the ordering $<$) on which f depends, f does not depend on $\mathit{var}(w)$, and therefore, the subgraphs rooted at $\mathit{left}(w)$ and $\mathit{right}(w)$ are isomorphic to the subgraph rooted at v. This, however, contradicts the fact that B is reduced. So, such vertex w does not exist, and v is the root of B. BY a symmetric argument it follows that v' is the root of B'. As H (defined above) relates v and v' it follows that the roots are isomorphic. *qed.*

As a result, several computations on boolean functions can be easily decided using their ROBDD representation. For instance, to decide the equivalence of

two boolean functions it suffices to check whether their ROBDDs are similar, i.e., isomorphic. Thus, the check whether a boolean function is always true (or false) for any variable assignment of boolean values to its variables, amounts to simply check equality of its ROBDD with a single terminal vertex labeled 1 (or 0), the ROBDD that represents true (or false). Furthermore, to test the satisfiability of a boolean function f – does $f(x_1, \ldots, x_n) = 1$ for some of the assignments to its variables? – amounts to check whether its ROBDD is not equal to the ROBDD with a single terminal vertex labeled 0. Such check can be performed rather efficiently. For boolean expressions this satisfiability problem is NP-complete by Cook's theorem. Note that it is assumed that an ROBDD-representation is at our disposal and obtianing this might not be easy!

Another consequence of this theorem is that when we apply the three reduction rules listed before to an OBDD until no further reductions are possible, then we are guaranteed that always the *same* resulting reduced OBDD is obtained. In particular, the order in which the several reductions are applied is irrelevant.

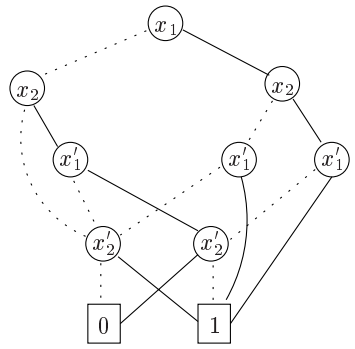
Example 11.7. Consider again our running example and $X = \{ x_1, x_2 \}$ *with* $x_1 < x_2$. *The BDT over* $\langle X, < \rangle$ *representing* f_I, *the characteristic function of the set of initial states, consists of 7 vertices, cf. Figure 11.2. It is evident that only the value of boolean variable* x_2 *is relevant to decide the value of* f_I. *The ROBDD over* $\langle X, < \rangle$ *thus consists of just a vertex labeled* x_2 *that has a one-leaf as left child, and a zero-leaf as right child:*



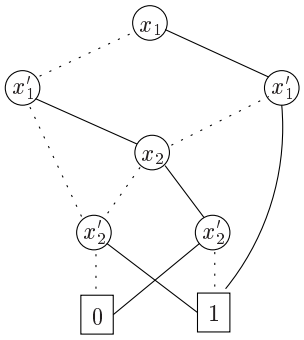
Let $X = \{ x_1, x_2, x_1', x_2' \}$ *with* $x_1 < x_2 < x_1' < x_2'$. *The BDT over* $\langle X, < \rangle$ *that represents the characteristic function* f_R *consists of* $2^5 - 1 = 31$ *vertices, cf. Figure 11.3. The ROBDD over* $\langle X, < \rangle$ *representing* f_R *is depicted in Figure 11.6(a). Since the BDT contains a substantial degree of redundancy (as argued before), the size of the ROBDD is significantly smaller; it only consists of 10 vertices. Notice that for some evaluations of* f_R, *the value of a variable might be irrelevant, e.g.,* $f_R(\langle 0, 0 \rangle, \langle 0, 1 \rangle)$ *is determined without using the fact that variable* x_1' *equals 0. This is reflected in the ROBDD by the fact that no vertex labeled with* x_1' *is encountered on determining the value* $f_R(\langle 0, 0 \rangle, \langle 0, 1 \rangle)$.
(End of example.)

11.3.3 Constructing a Reduced Ordered BDD

There are basically two ways to generate an ROBDD from a boolean expression: either an OBDD is constructed and subsequently reduced by applying the reduction rules listed before, or an OBDD is constructed and reduced on-the-fly,



(a) ordering $x_1 < x_2 < x'_1 < x'_2$

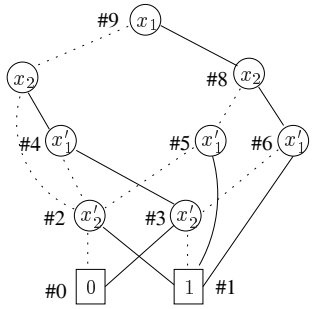


(b) ordering $x_1 <' x'_1 <' x_2 <' x'_2$

Figure 11.6: Two ROBDDs (for different variable orderings) representing a transition relation

i.e., reduced during construction.

In the latter approach, typically, a hash table, called "unique table", is maintained which contains all currently existing OBDD vertices. A table entry for a vertex stores all necessary information such as the identifier of its boolean variable, and references to (the identities of) its two children vertices, cf. Table 11.1. Reference counters are used to keep track of the number of references to a vertex. This information is used to remove (non-root) vertices from memory once their reference counter equals zero. On creating a new BDD vertex it is checked by a simple table look-up whether the unique combination of its variable identifier together with the identifiers of its children already appears in the table. If not, the corresponding item is inserted into the hash table; otherwise, it is not. In the sequel, we will assume that this implementation scheme is used.



vertex	*var*	*left*	*right*
#0	n/a	n/a	n/a
#1	n/a	n/a	n/a
#2	3	#0	#1
#3	3	#1	#0
#4	2	#2	#3
#5	2	#2	#1
#6	2	#3	#1
#7	1	#0	#4
#8	1	#5	#6
#9	0	#7	#8

Table 11.1: Example of a unique table for an ROBDD where variable x_1, x_2, x'_1 and x'_2 are indicated by 0 through 3, respectively

An alternative approach is to generate an OBDD and then reduce it. This is

done using the operation REDUCE. For the sake of completeness, we describe this operation here. Given OBDD B over $\langle X, < \rangle$, the operation REDUCE(B) returns a reduced OBDD B' over $\langle X, < \rangle$ such that $f_{\mathsf{B}} = f_{\mathsf{B}'}$. Suppose $X = \{ x_1, \ldots, x_n \}$. The algorithm REDUCE traverses the OBDD B in a bottom-up fashion starting from the terminal vertices. Each terminal vertex v with $val(v) = 0$ is labeled with $id(v) = \#0$; similarly, all one-terminals are labeled by $\#1$. On encountering a nonterminal vertex, v, say, an identifier $id(v)$ is assigned in such a way that vertices representing the same boolean function are equally labeled. For vertices at level k, this works as follows. Suppose we have labeled all vertices up to level k, i.e., all terminals and all nonterminals that are at level $m > k$. The label of vertex v is now determined as follows:

- If $id(left(v)) = id(right(v))$, then v is a redundant vertex – as it represents the same boolean function as its children – and thus obtains the same identifier.
- If there is another vertex w that is labeled with $var(v)$ and, moreover, $id(left(w)) = id(left(v))$ and $id(right(w)) = id(right(v))$, then v and w represent the same boolean function, and v gets the same identifier as w.
- If none of these cases apply, we assign the next unused identifier to v.

The worst case time-complexity of REDUCE is $\mathcal{O}(|\,\mathsf{B}\,| \cdot \log |\,\mathsf{B}\,|)$ where $|\,\mathsf{B}\,|$ denotes the number of vertices in OBDD B.

Example 11.8. The left part of Figure 11.7 depicts an ordered BDD that is not reduced. The identifiers assigned to vertices in the way explained above (where at each level vertices are labeled from left to right) are indicated next or beneat the vertex. The corresponding reduced OBDD is depicted in the right part of Figure 11.7. *(End of example.)*

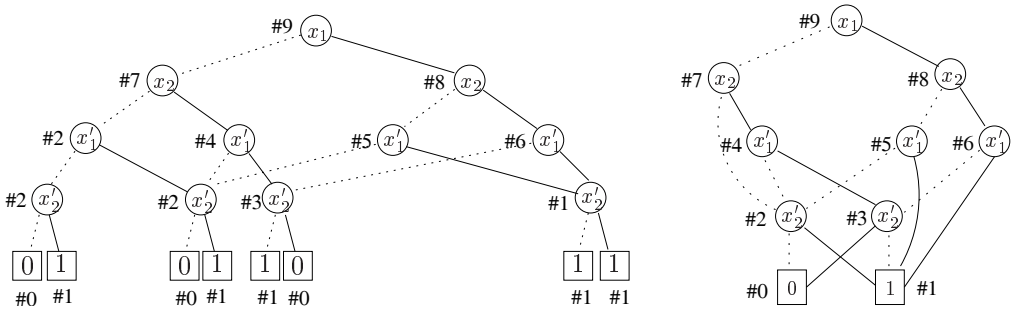


Figure 11.7: Reducing an ordered BDD (left) into a reduced OBDD (right) using the procedure REDUCE

11.3.4 Variable Ordering

The size of an ROBDD strongly depends on the ordering of the boolean variables. This is illustrated by the following example.

Example 11.9. Figure 11.6(b) depicts the ROBDD representing the characteristic function f_R of our running example, using a different variable ordering: $x_1 <' x_1' <' x_2 <' x_2'$. This ROBDD over $\langle X, <' \rangle$ consists of 8 vertices and thus exploits the redundancy in the BDT of Figure 11.3 in a more optimal way than the ROBDD in Figure 11.6(a). This is, for instance, illustrated by considering the path in order to determine the value of $f_R(\langle 1,1\rangle, \langle 1,1\rangle)$: in the ROBDD of Figure 11.6(a) three vertices are visited in order to determine this function value, whereas in the ROBDD of Figure 11.6(b) only two vertices are needed. (End of example.)

For representing the transition relation of Kripke structures, experiments have shown that an ordering on the variables in which the bits representing the source and target of a transition are alternated, provides rather compact ROBDDs. Thus, for $[\![s]\!] = (b_1, \ldots, b_n)$ and $[\![s']\!] = (b_1', \ldots, b_n')$ an effective ordering of the boolean variables is:

$$x_1 < x_1' < x_2 < x_2' < \ldots < x_n < x_n'$$

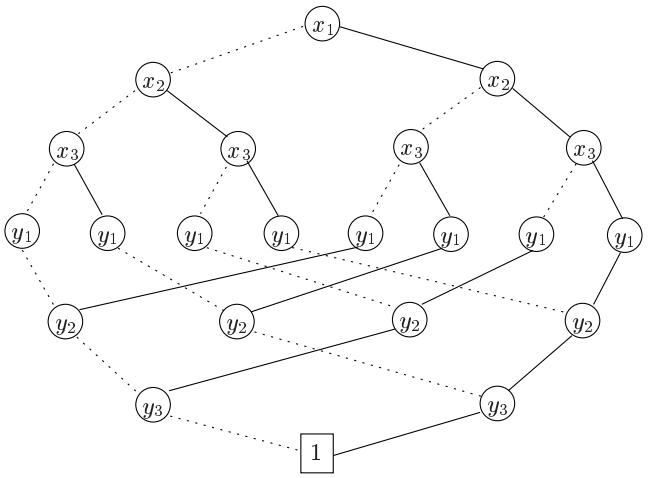
or its symmetric counterpart $x_1' < x_1 < \ldots < x_n' < x_n$. This ordering scheme has been applied in Figure 11.6(b). The more naive ordering in which all bits encoding the source state precede the bits encoding the target state is:

$$x_1 < \ldots < x_n < y_1 < \ldots < y_n$$

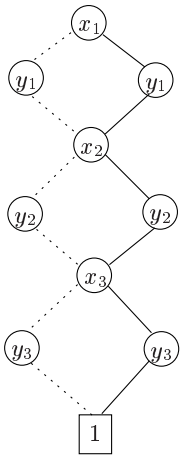
and is applied in Figure 11.6(a).

For larger examples, the differences in BDD-size can be substantially larger than in the previous example. Examples do exist for which under a given ordering an exponential number of vertices are needed, whereas under a different ordering of the variables only a linear number of vertices is needed. Moreover, there do exist boolean functions that always have exponential-size ROBDDs: regardless of the variable ordering, the ROBDD representing the function $H_n : \{0,1\}^n \to \{0,1\}$ defined by $H_n(x_1, \ldots, x_n) = x_k$ with $0 < k \leqslant n$ where k is the number of ones in the input and $x_0 = 0$, is exponential in n. $H_n(x_1, \ldots, x_n)$ yields a bit position where x_k is the number of ones in the input. For instance, for $n = 4$ we have $H_4(0,0,1,0) = 0$, as $k = 1$ and $x_1 = 0$ and $H_4(1,1,1,0) = 1$ as $k = 3$ and $x_3 = 1$.

Example 11.10. Consider the following boolean function that takes $2n$ param-



(a) $x_1 < x_2 < x_3 < y_1 < y_2 < y_3$



(b) $x_1 < y_1 < x_2 < y_2 < x_3 < y_3$

Figure 11.8: Two reduced OBDDs representing the same boolean function

eters:

$$f(x_1, \ldots, x_n, y_1, \ldots, y_n) = (x_1 \iff y_1) \ \wedge \ \ldots \ \wedge \ (x_n \iff y_n)$$

This function compares two n-bit input vectors. Under the variable ordering $x_1 < \ldots < x_n < y_1 < \ldots < y_n$*, the ROBDD representing f consists of* $3 \cdot 2^n - 1$ *vertices. That is, the size of the ROBDD is exponential in the number of boolean variables. Under the variable ordering* $x_1 < y_1 < \ldots < x_n < y_n$*, however, the ROBDD representing f has a size that is proportional to n, i.e., more precisely, it has* $3 \cdot n + 2$ *vertices. This is illustrated for* $n = 3$ *in Figure 11.8 where it should be noted that the 0-terminal is omitted to simplify the picture.*

An intuitive explanation for this dramatic difference in size is given by the matrix representation of $f(\langle x_1, x_2, x_3 \rangle, \langle y_1, y_2, y_3 \rangle)$ *and* $f(\langle x_1, y_1, x_2 \rangle, \langle y_2, x_3, y_3 \rangle)$*, where the order of the six arguments corresponds to the variable orderings. For the former function, we have that all diagonal elements equal one whereas all other elements are zero, yielding no possibilities for reductions. For the latter, however, the one-values are more scattered around the matrix, allowing possible reductions. These effects are illustrated by the following matrices:*

	000	*001*	*010*	*100*	*011*	*110*	*101*	*111*
000	1	0	0	0	0	0	0	0
001	0	1	0	0	0	0	0	0
010	0	0	1	0	0	0	0	0
100	0	0	0	1	0	0	0	0
011	0	0	0	0	1	0	0	0
110	0	0	0	0	0	1	0	0
101	0	0	0	0	0	0	1	0
111	0	0	0	0	0	0	0	1

$x_1 < x_2 < x_3 < y_1 < y_2 < y_3$

	000	*001*	*010*	*100*	*011*	*110*	*101*	*111*
000	1	0	0	0	1	0	0	0
001	0	0	0	1	0	0	0	1
010	0	0	0	0	0	0	0	0
100	0	0	0	0	0	0	0	0
011	0	0	0	0	0	0	0	0
110	1	0	0	0	1	0	0	0
101	0	0	0	0	0	0	0	0
111	0	0	0	1	0	0	0	1

$x_1 < y_1 < x_2 < y_2 < x_3 < y_3$

(End of example.)

Can we not just determine the variable ordering that gives rise to the smallest ROBDD? Unfortunately, this optimal variable ordering problem is NP-complete:

Theorem 11.2.
The optimal variable ordering problem for ROBDDs is NP-complete.

Most model checkers based on a symbolic representation of the state space using ROBDDs apply heuristics. There are two kinds of heuristics for finding a sub-optimal variable ordering. Static approaches try to derive an ordering from the system model (a Kripke structure or a hardware circuit) a priori to the construction of the ROBDD by inspecting the mutual dependencies between the boolean variables. Dynamic approaches changes the variable ordering during or after the construction of the ROBDD to find a compact representation. The latter schemes are mostly used in model checkers.

11.3.5 OBDDs and Automata on Finite Words

There is a strong analogy between OBDDs and deterministic finite-state automata. Recall from Chapter 4 that a finite-state automaton (FSA, for short) is defined over a set of input symbols (its alphabet) and consists of a set of states, a transition function that given a symbol from the alphabet and the current state determines the possible next states, a distinguished set of initial states, and a set of accept states. An OBDD B is in fact a deterministic FSA over the alphabet $\{0, 1\}$. The states of the FSA correspond to the nonterminals of B, and the edges to the children of a nonterminal are considered as transitions to next states labeled with 0 (for left children) and 1 (for right children), respectively. The root acts as the single initial state, whereas the terminal labeled with value 1 is the accept state. The terminal labeled with 0 is a non-accept state. Formally:

Definition 11.5. (**Mapping of an OBDD onto a deterministic FSA**)
For OBDD B let the FSA $A_{\mathsf{B}} = (\Sigma, S, I, \longrightarrow, F)$ be defined as follows:

- $\Sigma = \{0, 1\}$
- $S = V$, the set of vertices of B
- $I = \{ v \in V \mid v \text{ is the root of } \mathsf{B} \}$
- $\longrightarrow$ is defined as the smallest relation defined as follows:

  - $v \xrightarrow{0} w$ if and only if $w = \mathit{left}(v)$
  - $v \xrightarrow{1} w$ if and only if $w = \mathit{right}(v)$
- $F = \{\, v \in V \mid v \text{ is terminal and } \mathit{val}(v) = 1 \,\}$

The OBDD B for function $f_{\mathsf{B}} : \{0,1\}^n \to \{0,1\}$ can be considered as a deterministic FSA for the language $f^{-}(1)$ of words of length n, where the FSA may skip – like redundant vertices – bits from the input.

Theorem 11.3.
For any OBDD B*:* $\mathcal{L}(A_{\mathsf{B}}) = f_{\mathsf{B}}^{-1}(1)$.

This result can be justified as follows. If for a certain sequence of zeroes and ones, the FSA ends in an accept state, then f_{B} applied to this input yields 1; if the FSA ends in a non-accept state, then f_{B} yields 0.

Example 11.11. The deterministic FSA that corresponds to the OBDD of Figure 11.6(b) is depicted in Figure 11.9. (End of example.)

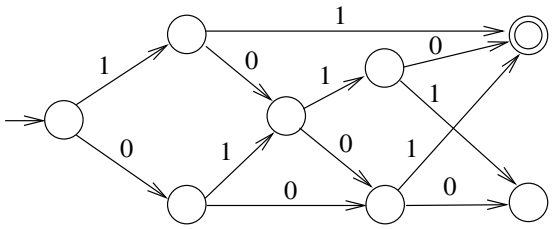


Figure 11.9: A deterministic FSA that corresponds to an OBDD

The relationship between an OBDD B and its reduced variant is almost identical to the relationship between the deterministic FSA representing B and its minimized FSA that accepts the same language. Moreover, the following analogies do exist between reduced OBDDs and minimised deterministic FSA:

- A minimised FSA for a given accepted language is unique up to isomorphism. An ROBDD for a given boolean function is unique up to isomorphism given a variable ordering.
- Two languages can be tested for equivalence – do they contain the same words? – by checking whether the minimised automata that accept these languages are isomorphic. The equivalence of two boolean functions can be tested by checking whether their ROBDD representations are isomorphic. In both cases, efficient algorithms do exists for checking isomorphism.

- The question whether an automaton is non-empty – does there exist an accepting word? – is decidable by the (efficient) test whether there exists a path from the initial state to an accept state. This is analogous to testing the satisfiability of a boolean function by checking whether its ROBDD has a reachable terminal with value one.
- The standard operations on languages, such as union, intersection and complementation, can efficiently be performed on the corresponding accepting minimised automata. Similarly, disjunction, conjunction and negation can efficiently be performed on ROBDDs (see below).

11.4 Operations on ROBDDs

11.4.1 Negation

Let B be an ROBDD over $\langle X, < \rangle$ representing the function f_{B}. The ROBDD representing the negation of this function, denoted $\textsc{Not}(\mathsf{B})$, is obtained from B by simply swapping the two terminal vertices. By this operation it is evident that whenever $f_{\mathsf{B}}(x_1, \ldots, x_n)$ equals one, then $f_{\textsc{Not}(\mathsf{B})}(x_1, \ldots, x_n)$ equals zero, and similar for the other case. It is also not difficult to see that $\textsc{Not}(\mathsf{B})$ is a reduced OBDD given that B is a reduced OBDD.

11.4.2 Variable Renaming

Let B be an ROBDD over $\langle X, < \rangle$ representing the function f_{B}, and let y be a fresh boolean variable, i.e., $y \notin X$, such that $x_{i-1} < y < x_{i+1}$. The function $\textsc{Rename}(\mathsf{B}, x_i, y)$ yields an ROBDD over $(X - \{ x_i \}) \cup \{ y \}$ ordered under $<$, that results from B by replacing the label of any x_i-labeled vertex into y. This operator can be generalized towards the renaming of several variables in a straightforward manner. Let $\textsc{Rename}(\mathsf{B}, x_{i_1}, \ldots, x_{i_k}, y_{i_1}, \ldots, y_{i_k})$ be a shorthand for the successive renaming of variable x_{i_1} by y_{i_1}, x_{i_2} by y_{i_2}, and so on.

11.4.3 Binary Operations

The operator $\textsc{Apply}$ performs a point-wise application of the binary operator op (e.g., conjunction $\wedge$ or disjunction $\vee$) to two ROBDDs. Formally, for ROBDDs B_1 and B_2 over $\langle X, < \rangle$, $\textsc{Apply}(op, \mathsf{B}_1, \mathsf{B}_2)$ yields an ROBDD over $\langle X, < \rangle$ representing the function $f_{\mathsf{B}_1}\ op\ f_{\mathsf{B}_2}$. In order to determine $\textsc{Apply}(op, \mathsf{B}_1, \mathsf{B}_2)$, the function listed in Table 11.2 is invoked where vertex v_i is the root of ROBDD B_i. In general, this function is invoked with vertices of B_1 and B_2.

```
function Apply(op : BinOp, v1, v2 : BDDvertex) : BDDvertex;
(* pre: v1 and v2 are vertices of ROBDDs over ⟨X, <⟩ *)
begin var v : BDDvertex;
      if (v1 and v2 are terminals)
      then val(v) := val(v1) op val(v2);
      else if (v1 and v2 are nonterminals  ∧  var(v1) = var(v2))
           then var(v) := var(v1);
                left(v) := Apply(op, left(v1), left(v2));
                right(v) := Apply(op, right(v1), right(v2));
           else if (var(v1) < var(v2));
                then var(v) := var(v1);
                     left(v) := Apply(op, left(v1), v2);
                     right(v) := Apply(op, right(v1), v2);
                else var(v) := var(v2);
                     left(v) := Apply(op, v1, left(v2));
                     right(v) := Apply(op, v1, right(v2));
                fi
           fi
      fi
return v;
(* post: v is a BDD-vertex representing f_B(v1) op f_B(v2) *)
end
```

Table 11.2: Algorithm to apply a binary operator to two ROBDDs

The algorithm for *Apply* follows a recursive descent scheme using Shannon's expansion. More concretely, for boolean functions f and g over $x_1, \ldots, x_n$:

$$\begin{aligned} f \ op \ g \quad = \quad & (x_1 \ \wedge \ (f[x_1 := 1] \ op \ g[x_1 := 1])) \\ \vee \ & (\neg x_1 \ \wedge \ (f[x_1 := 0] \ op \ g[x_1 := 0])) \end{aligned}$$

Stated in words, this means that starting from the root of the ROBDDs representing f and g, the ROBDD representing $f \ op \ g$ can be determined by recursively constructing the left- and right-branches (the 0- and 1-expressions) and then form an ROBDD from there. During this recursive scheme, four cases are distinguished. The simplest case occurs when vertices v_1 and v_2 are both terminals; then a terminal node is returned with value *op* applied to the values of the terminals v_1 and v_2. In the remaining cases, at least one of the vertices is a nonterminal. If both vertices are equally labeled, then *Apply* is applied recursively according to the above expansion. If $val(v_1) < val(v_2)$ then there is no vertex in B_2 labeled with $val(v_1)$, because both B_1 and B_2 are over the same variable ordering. Thus f_{B_2} is independent of $val(v_1)$, and *Apply* needs to be applied recursively only to the children of v_1. The case $val(v_1) > val(v_2)$ is handled symmetrically.

Example 11.12. (Taken from [10]). The result of calling $\textsc{Apply}(\wedge, \mathsf{B}_1, \mathsf{B}_2)$ *on the ROBDDs* B_1 *(cf. Figure 11.10(a)) and* B_2 *(cf. Figure 11.10(b)) is given in Figure 11.10(c). To simplify the drawings, the 0-terminal vertices have been*

omitted. *(End of example.)*

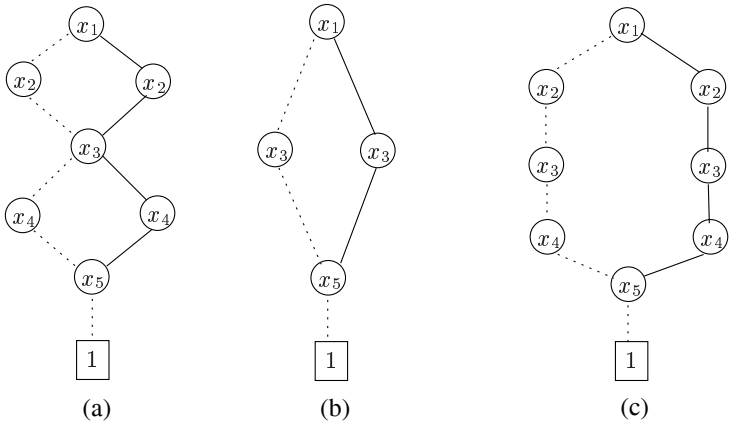


Figure 11.10: Two ROBDDs and their conjunction

To improve its efficiency, the function *Apply* may avoid recursive invocations if, depending on the binary operator *op* at hand, one of the two values of the formal parameters has a value that fully determines the resulting vertex. For instance, if *op* is conjunction, and v_1 is a terminal vertex with $val(v_1) = 0$, a terminal vertex with value zero can be returned without any recursive invocation.

A more important optimisation is the following. Due to the recursive nature of Apply it may happen that several invocations with the same formal parameters, i.e., the same BDD vertices, occur. In fact, Apply has an exponential time complexity, since the processing of each nonterminal gives rise to two recursive invocations. In order to avoid re-computation, in most implementations *dynamic programming* is employed. Using this technique, computed results are stored once they are determined and prior to each recursive invocation it is checked whether the requested result is stored. If so, then the recursive call is not made and the stored result is used instead, otherwise Apply is invoked recursively. The table storing the intermediate results is called "computed table". Using dynamic programming, there can be at most $|\,\mathsf{B}_1\,|\cdot|\,\mathsf{B}_2\,|$ recursive invocations, each adding at most one vertex to the resulting ROBDD. Given a hash table implementation, each step can be performed in constant time on average. Thus, both the time complexity and the size of the resulting ROBDD are in $\mathcal{O}(|\,\mathsf{B}_1\,|\cdot|\,\mathsf{B}_2\,|)$.

11.4.4 Replacement by Constants

Let B be an ROBDD over the variables x_1 through x_n. Restrict(B, x_i, b) returns the ROBDD which results from replacing (also called "restricting") the variable x_i by the boolean constant b. That is to say, Restrict(B, x_i, b) returns

the ROBDD representing the function $f_{\mathsf{B}}[x_i := b]$. Note that RESTRICT(B, x_i, b) is an ROBDD which no longer depends on variable x_i, i.e., it only depends on $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$. The ROBDD RESTRICT$(\mathsf{B}, x_i, b)$ can be obtained from B by replacing any edge from a vertex v to a vertex w labeled with x_i by an edge to *left*(w) if b equals zero, and to *right*(w) otherwise. Thus, all x_i-labeled vertices are "bypassed". Subsequently, all vertices labeled with x_i are removed. Finally, the resulting OBDD is reduced to bring it into reduced form. (This is typically guaranteed by constructing RESTRICT(B, x_i, b) using the unique table described before.) The time complexity and the size of the resulting ROBDD are proportional to $|\,\mathsf{B}\,|$.

Example 11.13. Figure 11.11 depicts an example application of RESTRICT. *The variable x_2 in ROBDD B (on the left) is restricted to one. The resulting ROBDD* RESTRICT$(\mathsf{B}, x_2, 1)$ *is depicted on the right. As explained, the edges to the vertex labeled with x_2 are redirected to its right child. (End of example.)*

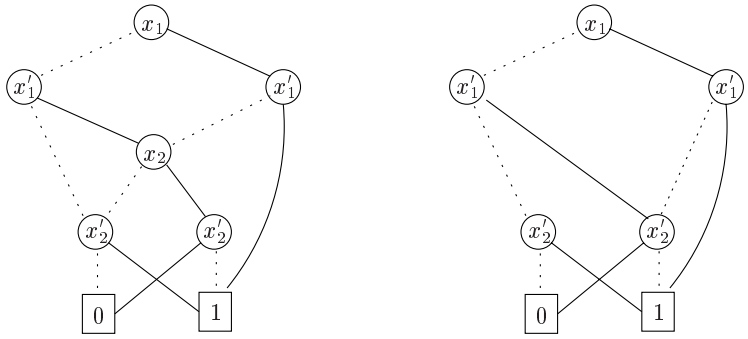


Figure 11.11: Example of replacing the variable x_2 by the constant 1

11.4.5 Abstraction

Let B be an ROBDD over $\langle X, < \rangle$ with the boolean variables $X = x_1, \ldots, x_n$, and *op* be an associative binary boolean operator, such as conjunction ($\wedge$) or disjunction ($\vee$). The ROBDD over $\langle X, < \rangle$ that represents the abstraction of B with respect to variable x_i and operator *op*, denoted ABSTRACT(B, x_i, op), represents the function $f_{\mathsf{B}}[x_i := 0]$ *op* $f_{\mathsf{B}}[x_i := 1]$. For disjunction (i.e., *op* equals $\vee$), abstraction with respect to variable x_i amounts to existential quantification over x_i, as:

$$\exists x_i.\, f(x_1, \ldots, x_n) = f[x_i := 1] \;\vee\; f[x_i := 0]$$

A naive implementation of existential quantification over x_i is to simply return

$$\text{APPLY}(\vee, \text{RESTRICT}(\mathsf{B}, x_i, 1), \text{RESTRICT}(\mathsf{B}, x_i, 0))$$

As the ROBDDs RESTRICT($\mathsf{B}, x_i, 0$) and RESTRICT($\mathsf{B}, x_i, 1$) are very similar in structure, the efficiency of this operation can be improved substantially by replacing each node in B labeled with x_i by the result of applying $\vee$ on its children, while keeping the rest of the ROBDD B in tact.

Due to the associativity of the operator *op*, abstraction can be generalized with respect to several variables: the ROBDD ABSTRACT($\mathsf{B}, (x_{i_1}, \ldots, x_{i_n}), op$) represents the boolean function

$$f_{\mathsf{B}}[x_{i_1} := 0, \ldots, x_{i_n} := 0] \; op \; \ldots \; op \; f_{\mathsf{B}}[x_{i_1} := 1, \ldots, x_{i_n} := 1]$$

That is, it represents a combination by the operator *op* of all possible restrictions of B with respect to the variables x_{i_1} through x_{i_n}.

Example 11.14. Abstraction is illustrated in Figure 11.12 by abstracting from variable x_2 in the ROBDD in Figure 11.6(b). *(End of example.)*

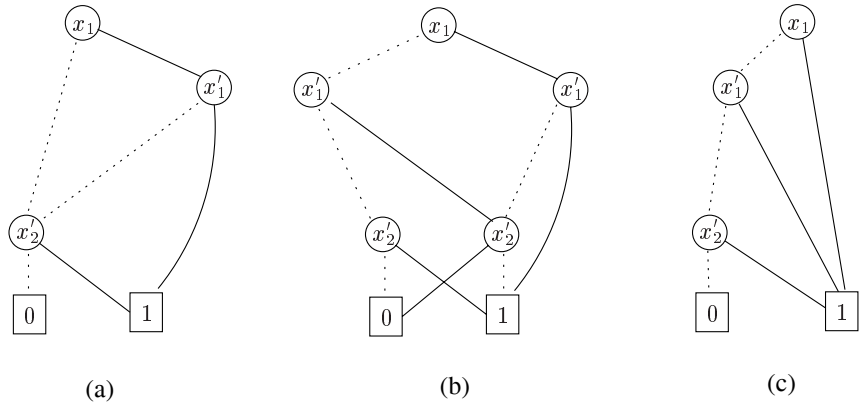


Figure 11.12: (a) RESTRICT($\mathsf{B}, x_2, 0$) and (b) RESTRICT($\mathsf{B}, x_2, 1$), and (c) ABSTRACT($\mathsf{B}, x_2, \vee$)

11.5 Symbolic Model Checking

Given the operations on ROBDDs we are now in a position to explain in detail how CTL model checking works based on a symbolic representation of the Kripke structure under consideration. The first question, of course, is: how do we obtain an ROBDD representation from a Kripke structure? Assume that $|S| = 2^n$ and let $a_1, \ldots, a_n$ be an enumeration of the atomic propositions that occur in the Kripke structure. This provides enough information to obtain the binary encodings of states. The algorithm to construct an ROBDD representation from a given Kripke structure is straightforward and works as

```
function bddSat (Φ : Formula) : ROBDD;
(* pre: Φ is a base CTL-formula *)
begin
  switch(Φ) :
      case Φ = true then return Const(1)
      case Φ = false then return Const(0)
      case Φ = a_i ∈ AP then return ROBDD for f(x_1, ..., x_n) = x_i
      case Φ = ¬ Φ_1 then return Not(bddSat(Φ_1))
      case Φ = Φ_1 ∨ Φ_2 then return Apply(∧, bddSat(Φ_1), bddSat(Φ_2))
      case Φ = EX Φ_1 then return bddEX(Φ_1);
      case Φ = E (Φ_1 U Φ_2) then return bddEU(Φ_1, Φ_2)
      case Φ = EG Φ_1 then return bddEG(Φ_1)
  end switch
(* post: ROBDD bddSat(Φ) represents Sat(Φ) = { s | s |= Φ } *)
end
```

Table 11.3: The recursive algorithm for symbolic model checking of CTL

follows: starting from the ROBDD representing 0, the Kripke structure is traversed, e.g., in a depth-first search manner, and on encountering an edge, an ROBDD for this edge is constructed and combined with the "or" operation with the ROBDD representing all previously processed transitions. This yields the ROBDD R that represents the successor relation R. Similarly, the ROBDD I representing the set of initial states is constructed in a setp-wise fashion on encountering an initial state.

Given this symbolic representation of the Kripke structure, model-checking the CTL-formula Φ takes place in the same manner as in the non-symbolic setting (cf. Chapter 7). That is, the main algorithm consists of a bottom-up traversal of the parse tree of the formula Φ. For each node of the parse tree, i.e., for each sub-formula Ψ of Φ, the set $Sat(\Psi)$ of states is computed for which Ψ holds. This computation is carried out level-wise, starting from the leafs of the parse tree – the nodes that correspond to the atomic propositions – and finishing at the root of tree, the (only) node in the parse tree that corresponds to Φ. At an intermediate node, the results of the computions of its children are used and combined in an appropriate way to establish the states of its associated sub-formula. The type of computation at such node depends on the operator (e.g., $\wedge$, EX or EU) that is at the "top level" of the sub-formula treated. By following this bottom-up procedure, one obtains the set of states for which the requested formula Φ holds at the root of the parse tree. The main difference with the non-symbolic procedure is that the set $Sat(\Phi)$ is not stored explicitly anymore, but as an ROBDD. The recursive algorithm is shown in Table 11.3. The ROBDDs consisting of just a single terminal vertex labeled with 0 or 1 are indicated by Const(0) and Const(1), respectively. The ROBDD representing $f(x_1, \ldots, x_n) = x_i$ consists of a single nonterminal v labeled with x_i such that the terminals $left(v)$ and $right(v)$ are labeled with 0 and 1, respectively.

For formulae of the form EX Φ the function *bddEX* (cf. Table 11.4) is in-

```
function bddEX(Φ : Formula) : ROBDD;
(* pre: Φ is a base CTL-formula *)
begin var B, B′, N : ROBDD;
        B := bddSat(Φ);
        B′ := RENAME(B, x_1, ..., x_n, x′_1, ..., x′_n);
        N := APPLY(∧, R, B′);
return ABSTRACT(N, (x′_1, ..., x′_n), ∨);
(* post: bddEX(Φ) represents Sat(EX Φ) = { s | s ⊨ EX Φ } *)
end
```

Table 11.4: Algorithm to symbolically compute $Sat(\mathsf{EX}\,\Phi)$

voked that computes the ROBDD representing the characteristic function of $Sat(\mathsf{EX}\,\Phi)$ as follows. First, the ROBDD B representing $Sat(\Phi)$ is determined. All state variables x_i are renamed into their primed variants, yielding a primed version of $Sat(\Phi)$, stored as B'. The ROBDD $\textsc{Apply}(\wedge, \mathsf{R}, \mathsf{B}')$ is constructed, representing the set of pairs (s, s') in R such that $s' \in Sat(\Phi)$. Finally, all boolean variables x'_1 through x'_n are deleted by abstraction to obtain the ROBDD representing

$$\{\, s \mid \exists s'.\, R(s, s') \;\wedge\; s' \in Sat(\Phi) \,\}$$

Note that in the last step of the construction, the boolean variables x_1 through x_n are retained.

The function *bddEU* (cf. Table 11.5) is based on the fixed-point characterization of $\varphi = \mathsf{E}\,(\Phi\,\mathsf{U}\,\Psi)$. In fact, it iteratively computes the least fixed point of

$$\mathsf{E}\,(\Phi\,\mathsf{U}\,\Psi) = \Psi \;\vee\; (\Phi \;\wedge\; \mathsf{EX}\,\mathsf{E}\,(\Phi\,\mathsf{U}\,\Psi))$$

in a symbolic manner. As all Ψ-states satisfy φ, the computation starts with constructing the ROBDD N for $Sat(\Psi)$. The states satisfying Φ are recursively computed and stored as ROBDD B. P is initialised to $\textsc{Const}(0)$. An iterative procedure is subsequently started that can be considered to systematically check the state space in a "backwards" manner. In each iteration, all Φ-states are determined that can move by a single transition to (one of) the states of which we already know to satisfy φ. Thus, in the i-th iteration of the procedure, all Φ-states are considered that can move to a Ψ-state in at most i steps. This is performed by iteratively applying the same computations on ROBDDS as in the function *bddEX* and a computation step $\textsc{Apply}(\vee, \mathsf{P}, \mathsf{B}')$ where the states satisfying φ (stored as ROBDD P) are extended with the Φ-states that can reach these states in one transition. This procedure continues as long as no new states are added, i.e., when the ROBDDs N and P are equal. Note that due to their canonicity, equality of two ROBDDs is easy to check.

The function *bddEG* is in fact a symbolic algorithm to compute the greatest fixed-point for $\mathsf{EG}\,\Phi$ (cf. Chapter 7). It is very similar to *bddEU* and is omitted

here. This completes the description of the symbolic algorithm to model-check CTL based on the fixed-point characterizations.

```
function bddEU(Φ, Ψ : Formula) : ROBDD;
(* pre: Φ and Ψ are base CTL-formulae *)
begin var N, P, B : ROBDD;
      N, P, B := bddSat(Ψ), Const(0), bddSat(Φ);
      while N ≠ P
      do P := N;
         B' := Rename(B, x_1, ..., x_n, x'_1, ..., x'_n);
         B' := Apply(∧, R, B');
         B' := Abstract(B', (x'_1, ..., x'_n), ∨);
         N := Apply(∨, P, B');
      od;
return N;
(* post: bddEU(Φ, Ψ) represents Sat(E(Φ U Ψ)) = { s | s ⊨ E(Φ U Ψ) } *)
end
```

Table 11.5: Algorithm to symbolically compute $Sat(\mathsf{E}(\Phi\,\mathsf{U}\,\Psi))$

From Chapter 7, we know that the time complexity of CTL model checking can be improved by a factor N (the number of states) when the algorithm for EU is based on detecting strongly connected components. We briefly sketch how this can be performed symbolically. The most straightforward manner to compute the strongly connected components of a Kripke structure is as follows. Suppose, as before, that R is an ROBDD representing the successor relation R. As a first step, the transitive closure of R is computed. This can be performed using standard BDD-operations in an iterative manner, e.g., using iterative squaring, yielding the ROBDD R^*. We have $\mathsf{R}^*(x_1, \ldots, x_n, y_1, \ldots, y_n) = 1$ if and only if state s' with $[\![s']\!] = (y_1, \ldots, y_n)$ is reachable from s with $[\![s]\!] = (x_1, \ldots, x_n)$. The transpose of the relation R^* represents backward reachable states. Thus, states s and s' belong to the same strongly connected component if and only if the function represented by

$$\textsc{Apply}(\wedge, \mathsf{R}^*, (\mathsf{R}^*)^T)$$

yields one on the input $([\![s]\!], [\![s']\!])$. The set of all strongly connected components can now be obtained using abstraction, i.e., existential quantification.

11.6 Bibliographic Notes

Binary decision diagrams. BDDs originate from Lee [121] and Akers [2, 3] in the late fifties as representations for combinatorial switching circuits. Ordered BDDs as well as several manipulations on these structures are due to Bryant [35]. The important result that reduced OBDDs are unique up to isomorphism is due to Fortune, Hopcroft and Schmidt [75]. The fact that the

problem of optimal variable ordering is NP-complete is due to Bollig and Wegener [27]. Overviews of BDDs and manipulations thereof have been given by Bryant [36] and Andersen [10]. For a survey of the most important theoretical achievements on OBDDs, we recommend the recent overview by Wegener [186]. An improvement of the reduction algorithm of Bryant [35] has been given by Sieling and Wegener [165]; the time complexity of this algorithm is linear in the number of parameters of the boolean function at hand. More detailed accounts of binary decision diagrams can be found in the books by Meinel and Theobald [136], Drechsler and Becker [64] and Wegener [185]. The link between OBDDs and deterministic finite-state automata has been described by, among others, Thomas [176].

Symbolic model checking. The use of BDDs as compact representations in model checking of hardware circuits has been brought up in the late eighties by the independent teams of Burch *et al.* [38, 37] and Coudert, Berthet and Madre [59]. A detailed account of "symbolic" model checking was first given in the doctoral dissertation by McMillan [133]. McMillan also showed that abstraction of variables on BDDs is NP-hard. Symbolic algorithms for detecting strongly connected components have been described by Xie and Beeler [191] and Bloem, Gabow and Somenzi [24]. An elegant and lucid description of symbolic model checking can be found in the book by Huth and Ryan [102], who also consider the relational μ-calculus and the treatment of fair CTL. The "interleaved" variable ordering for representing transition systems by BDDs has been advocated by Enders, Filkorn and Taubner [73]. Prominent symbolic model checkers are SMV [133], NuSMV [43], and VIS [31]. The use of BDDs in the verification of logic circuits has been described by Kropf [113] and Janssen [105]. An important subject in this field is to construct an ROBDD describing the input-output relation of a hardware circuit, and to compare the functionality of circuits using their ROBDD representation. As ROBDDs are unique (up to isomorphism), this results in an efficient equivalence check for hardware circuits once the ROBDD representations are given.

Implementation issues. The main issues in implementing BDD packages have been described by Brace, Rudell and Bryant [30], Janssen [105] and, more recently, by Somenzi [170]. Unique tables that contain all BDD vertices are typically kept in dynamic hash tables where hash collisions are resolved by chaining. Garbage collection is based on reference counting. As the variable order has a significant impact on the size of the BDDs, most BDD packages include *dynamic* variable reordering algorithms, such as Rudell's sifting algorithm that is based on swapping variables at consecutive levels in a BDD [157]. Such algorithms typically slow down the BDD computations, but on larger cases (that take quite some verification time) outperform schemes without this facility [192]. Computed tables (for dynamic programming in the implementation of Apply), are typically kept in caches governed by a first-in, first-out replacement strategy.

Empirical results. Empirical results of using BDDs in model checking have

extensively been described by Yang *et al.* [192]. The authors study the performance of various BDD packages for a set of benchmark problems (16 SMV models) and conclude that BDD computations in model checking and in synthesizing BDDs for combinatorial circuits have fundamentally different performance characteristics. These differences include the effects of cache sizes (model checking is more sensitive with respect to cache size), garbage collection frequency (for model checking, this should be as low as possible), and memory locality (no significant difference between breadth-first and depth-first BDD-operations). In addition, Yang *et al.* provide results on different dynamic variable reordering algorithms and the impact of initial variable orders.

11.7 Exercises

EXERCISE 11.1. Let $f(x_1, x_2, x_3) = (x_1 \wedge X_2 \wedge x_3) \vee (\neg x_1 \wedge \neg x_3)$ be a boolean function. Give the Shannon expansion of f with respect to the variable x_1. Do the same for the variable x_2.

EXERCISE 11.2. Let $f(x_1, x_2, x_3, x_4) = (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$ be a boolean function over $X = \{ x_1, x_2, x_3, x_4 \}$.

1. Give a binary decision tree representing f using the ordering $x_1 < x_2 < x_3 < x_4$
2. Transform the BDT into an ROBDD by successively applying the reduction rules
3. Repeat the former two questions using the ordering $x_1 < x_3 < x_2 < x_4$

EXERCISE 11.3. Construct an ROBDD representing the input-output relation of the hardware circuit depicted in Figure 11.13. Use the ordering $x_1 < x_2 < x_3 < x_4$.

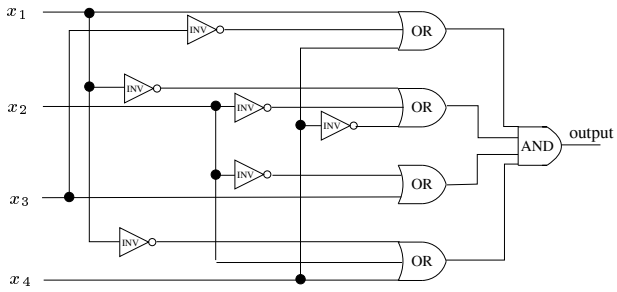


Figure 11.13: A hardware circuit with four inputs

EXERCISE 11.4. Give an ROBDD representation of the set numbers from the domain $\{ 0, 1, \ldots, 14 \}$ such that are either even or larger than 11.

EXERCISE 11.5. Consider the hidden weight function $H_n : \{0,1\}^n \to \{0,1\}$ defined by $H_n(x_1, \ldots, x_n) = x_k$ with $0 < k \leqslant n$ is a bit position where k is the number of ones in the input and $x_0 = 0$. For instance, for $n = 4$ we have $H_4(0,0,1,0) = 0$, as $x_1 = 0$ and $H_4(1,1,1,0) = 1$ as $x_3 = 1$. Give an ROBDD representation of H_n for n=4 using the variable ordering $x_1 < x_2 < x_3 < x_4$.

EXERCISE 11.6. Consider the following boolean function that has $2n$ inputs:

$$f(x_1, \ldots, x_{2n}) = (x_1 \ \vee \ x_2) \ \wedge \ \ldots \ \wedge \ (x_{2n-1} \ \vee \ x_{2n}).$$

Questions:

1. Give the ROBDD representing f for n=3 using the variable ordering $x_i < x_{i+1}$, for $0 < i < 2n$.
2. Give the ROBDD representing f for n=3 using the variable ordering $x_1 < x_3 < \ldots < x_{2n-1} < x_2 < x_4 < \ldots < x_{2n}$.
3. Give an expression for the number of vertices for representing f using the two indicated variable orderings.

EXERCISE 11.7. Consider an automaton with 7 control states s_0 through s_6, a bounded integer `var digit: 0..9` a boolean `var flag: boolean` and an indicator `var ind: 0..4`. The system is considered to be in an initial state if control is in state s_0, variable `digit`, and `ind` equal zero, and `flag` is arbitrary.

1. Find a binary encoding of the possible states of this automaton.
2. Based on this encoding, give a boolean expression that identifies the initial states.
3. Given an ROBDD representation of the set of states in which the proposition "`ready` implies `digit` exceeds three or `ind` equals two" holds.
4. Consider a transition from control state s_3 to s_4 which is enabled if `digit` and `ind` differ from zero, and which results in setting `flag` to true. Give a boolean expression representing this transition.

EXERCISE 11.8. Consider the function Mx that given the address vector $\underline{a} = (a_{k-1}, \ldots, a_0)$ describing as binary representation the address m, and the data vector $\underline{x} = (x_0, \ldots, x_{n-1})$ selects x_m. For instance for $\underline{a} = (0,1,1)$ and $\underline{x} = (0,1,0,1,0,1,1,0)$ we have that $Mx(\underline{a}, \underline{x}) = x_3 = 1$. Question: construct an ROBDD for the function Mx for n=5 using the variable ordering $a_{k-1} < \ldots < a_0 < x_0 < \ldots < x_{n-1}$.

EXERCISE 11.9. Alternatives to Shannon's expansion are negative and positive Davio expansion. For function $f : \{0,1\}^n \to \{0,1\}$ let the boolean difference with respect to x_i be defined as:

$$\frac{\delta f}{\delta x_i} = f[x_i := 0] \oplus f[x_i := 1] \text{ where } \oplus \text{ denotes "exclusive or"}$$

Prove that:

1. Davio's positive expansion: $f(x_1, \ldots, x_n) = f[x_i := 0] \oplus \left(x_i \ \wedge \ \frac{\delta f}{\delta x_i} \right)$
2. Davio's negative expansion: $f(x_1, \ldots, x_n) = f[x_i := 1] \oplus \left(\neg\, x_i \ \wedge \ \frac{\delta f}{\delta x_i} \right)$

EXERCISE 11.10. Consider the ROBDD of Figure 11.6(a), and name it B. Compute the reduced OBDDs RESTRICT$(\mathsf{B}, x_1', 0)$ and RESTRICT$(\mathsf{B}, x_1', 1)$.

EXERCISE 11.11. Define a generalization of RESTRICT towards replacing various boolean variables by constants at a time. That is, define the operation RESTRICT$(\mathsf{B},(x_{i_1}, \ldots, x_{i_n}), (b_{i_1}, \ldots, b_{i_n}))$ for ROBDD B over $\langle X, < \rangle$. You may assume that $x_{i_1} < \ldots < x_{i_n}$.

EXERCISE 11.12. Give a tree representation of the recursive invocations of the function APPLY for the example in Figure 11.10 assuming that a dynamic programming scheme is used to avoid recomputations.

EXERCISE 11.13. A BDD over $\langle X, < \rangle$ is an OBDD for which the ordering constraint is *not* required. A BDD is called read-once if each path in it contains at most one x_i-labeled vertex for each $x_i \in X$. A BDD is called oblivious with respect to x_1 through x_m, $x_i \in X$ if the set of nonterminals can be partitioned into m levels, such that level i only contains x_i-labeled vertices and all edges from level i to nodes from level j where $j > i$ are to terminals. Question: prove that an OBDD is a read-once oblivious BDD.

EXERCISE 11.14. Prove the relation between OBDDs and deterministic FSA as stated by Theorem 11.3.

Chapter 12

Partial-Order Reduction

Chapter 13

Memory Management Strategies

Chapter 14

Abstraction through (Bi-)Simulation

Bibliography

[1] Y. Abarbanel-Vinov and N. Aizenbud-Reshef and I. Beer and C. Eisner and D. Geist and T. Heymand and I. Reuveni and E. Rippel and I. Shitsevalov and Y. Wolfsthal and T. Yatzkar-Haham. On the effective deployment of functional formal verification. *Formal Methods in System Design*, **19**:35–44, 2001.

[2] S.B. Akers. On a theory of boolean functions. *Journal of the SIAM*, 1959.

[3] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, **27**(6):509–516, 1976.

[4] R. Alur and C. Courcoubetis and D. Dill. Model-checking in dense real time. *Information and Computation*, **104**:2–34, 1993.

[5] R. Alur and D. Dill. Automata for modeling real-time systems. In M.S. Paterson, editor, *Automata, Languages and Programming (ICALP)*, volume 443 of *Lecture Notes in Computer Science*, pages 332–342. Springer-Verlag, 1990.

[6] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, **126**(2):183–235, 1994.

[7] R. Alur and D. Dill. Automata-theoretic verification of timed automata. In *Formal Methods for Real-Time Computing*, pages 55–92. John Wiley & Sons, 1996.

[8] R. Alur and L. Fix and T.A. Henzinger. Event-clock automata: a determinizable class of timed automata. *Theoretical Computer Science*, **211**:253–273, 1999.

[9] R. Alur and T.A. Henzinger. Back to the future: towards a theory of timed regular languages. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 177–186, 1992.

[10] H.R. Andersen. An introduction to binary decision diagrams. Technical report, Technical University of Denmark, Dept. of Computer Science, 1994.

[11] H.R. Andersen. Model checking and Boolean graphs. *Theoretical Computer Science*, **126**(1):3–30, 1994.

[12] K.R. Apt. Ten years of Hoare's logic: a survey – part I. *ACM Transactions on Programming Languages and Systems*, **3**(4):431–483, 1981.

[13] K.R. Apt. Ten years of Hoare's logic: a survey part II: nondeterminism. *Theoretical Computer Science*, **28**:83–109, 1984.

[14] K.R. Apt and N. Francez and W.-P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, **2**:359–385, 1980.

[15] K.R. Apt and D. Kozen. Limits for the automatic verification of finite-state concurrent systems. *Information Processing Letters*, **22**:307–309, 1986.

[16] K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1997.

[17] E. Asarin and P. Caspi and O. Maler. Timed regular expressions. *Journal of the ACM*, **49**(2):172–206, 2002.

[18] T. Ball and A. Podelski and S. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer-Verlag, 2001.

[19] P. Behm and P. Benoit and A. Faivre and J.-M. Meynadier. Météor: a successful application of B in a large project. In J. Wing, J. Woodcock, and J. Davies, editors, *Formal Methods Vol. 1*, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387. Springer-Verlag, 1999.

[20] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.

[21] M. Ben-Ari and A. Pnueli and Z. Manna. The temporal logic of branching time. *Acta Informatica*, **20**:207–226, 1983.

[22] B. Bérard and M. Bidoit and A. Finkel and F. Laroussinie and A. Petit and L. Petrucci and Ph. Schnoebelen and P. McKenzie. *Systems and Software Verification*. Springer-Verlag, 2001.

[23] G. Bhat and R. Cleaveland and O. Grumberg. Efficient on-the-fly model checking for ctl*. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 388–397. IEEE Computer Science Press, 1995.

[24] R. Bloem and H.N. Gabow and F. Somenzi. An algorithm for strongly connected component analysis in $n \log n$ symbolic steps. In

W.A.Ĥunt Jr. and S.D. Johnson, editors, *Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 37–54. Springer-Verlag, 2000.

[25] B. Boehm and V.R. Basili. Software defect reduction top 10 list. *IEEE Computer*, **34**(1):135–137, 2001.

[26] B.W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1991.

[27] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, **45**(9):993–1006, 1996.

[28] S. Bornot and J. Sifakis. An algebraic framework for urgency. *Information and Computation*, **163**:172–202, 2001.

[29] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, 1986.

[30] K. Brace and R. Rudell and R.E. Bryant. Efficient implementation of a BDD package. In *ACM/IEEE Design Automation Conference (DAC)*, pages 40–45. IEEE Computer Science Press, 1990.

[31] R.K. Brayton and G.D. Hachtel and A.L. Sangiovanni-Vincentelli and F. Somenzi and A. Aziz and S.-T. Cheng and S.A. Edwards and S.P. Khatri and Y. Kukimoto and A. Pardo and S. Qadeer and R.K. Ranjan and S. Sarwary and T.R. Shiple and G. Swamy and T. Villa. Vis: a system for verification and synthesis. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer-Verlag, 1996.

[32] A. Bronstein and C. Talcott. Formal verification of synchronous circuits based on string-functional semantics: the 7 paillet circuits in Boyer-Moore. In *Proceedings of the Int. Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 317–333. Springer-Verlag, 1989.

[33] M.C. Browne and E.M. Clarke and D.L. Dill and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, **35**(12):1035–1044, 1986.

[34] J. Brunekreef and J.-P. Katoen and R. Koymans and S. Mauw. Design and analysis of dynamic leader election protocols in broadcast networks. *Distributed Computing*, **9**(4):157–171, 1996.

[35] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, **35**(8):677–691, 1986.

[36] R. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, **24**(3):293–318, 1992.

[37] J. Burch and E.M. Clarke and K.L. McMillan and D. Dill and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, **98**(2):142–170, 1992.

[38] J. Burch and E.M. Clarke and K.L. McMillan and D.L. Dill and L.J. Hwang. Sequential circuit verification using symbolic model checking. In *ACM/IEEE Design Automation Conference (DAC)*, pages 46–51. IEEE Computer Science Press, 1990.

[39] Philips Consumer Electronics B.V. Infra red remote control system RC6. Technical report, Philips, 1997.

[40] W. Chan and R.J. Anderson and P. Beame and S. Burns and F. Modugno and D. Notkin and J.D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineeering*, **24**(7):498–519, 1998.

[41] K.M. Chandy and J. Misra. *Parallel Program Design – A Foundation*. Addison-Wesley, 1988.

[42] Y. Choueka. Theories of automata on ω-tapes. *Journal of Computer and System Sciences*, **8**:117–141, 1974.

[43] A. Cimatti and E.M. Clarke and F. Giunchiglia and M. Roveri. NuSMV: a new symbolic model checker. *Journal on Software Tools and Technology Transfer*, **2**(4):410–425, 2000.

[44] E.M. Clarke and A. Biere and R. Raimi and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, **19**(1):7–34, 2001.

[45] E.M. Clarke and I.A. Draghicescu. Expressibility results for linear time and branching time logics. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time, and Partial Order in Logics and Model for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 428–437. Springer-Verlag, 1988.

[46] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic. In *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[47] E.M. Clarke and E.A. Emerson and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, **8**(2):244–263, 1986.

[48] E.M. Clarke and J. Wing et al. Formal methods: state of the art and future directions. *ACM Computing Surveys*, **28**(4):626–643, 1996.

[49] E.M. Clarke and O. Grumberg and H. Hiraishi and S. Jha and D.E. Long and K.L. McMillan and L.A. Ness. Verification of the Futurebus+ cache coherence protocol. In *Proceedings 11th Int. Symp. on Computer Hardware Description Languages and their Applications*, 1993.

[50] E.M. Clarke and O. Grumberg and K.L. McMillan and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *ACM/IEEE Design Automation Conference (DAC)*, pages 427–432. IEEE Computer Science Press, 1995.

[51] E.M. Clarke and S. Jha and Y. Lu and H. Veith. Tree-like counterexamples in model checking. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 19–29. IEEE Computer Science Press, 2002.

[52] E.M. Clarke and R. Kurshan. Computer-aided verification. *IEEE Spectrum*, **33**(6):61–67, 1996.

[53] E.M. Clarke and D. Peled and O. Grumberg. *Model Checking*. MIT Press, 1999.

[54] E.M. Clarke and H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbookof Automated Reasoning (Volume II)*, chapter 24, pages 1635–1790. Elsevier Publishers B.V., 2000.

[55] W.R. Cleaveland and J. Parrow and B. Steffen. The Concurrency Workbench: a semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, **15**(1):36–72, 1993.

[56] W.R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal μ-calculus. *Formal Methods in System Design*, **2**:121–147, 1993.

[57] R. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[58] T.H. Cormen and C.E. Leiserson and R.L. Rivest and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[59] O. Coudert and C. Berthet and J.C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag, 1990.

[60] C. Courcoubetis and M.Y. Vardi and P. Wolper and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, **1**:275–288, 1992.

[61] P.R. D'Argenio and E. Brinksma. A process algebra for timed automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*

(FTRTFT), volume 1135 of *Lecture Notes in Computer Science*, pages 110–129. Springer-Verlag, 1996.

[62] W.-P. de Roever and F.S. de Boer and U. Hannemann and J. Hooman and Y. Lakhnech and M. Poel and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[63] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[64] R. Drechsler and B. Becker. *Binary Decision Diagrams: Theory and Implementation*. Kluwer Academic Publishers, 1998.

[65] M.B. Dwyer and G.G. Avrunin and J.C. Corbett. Property specification patterns for finite-state verification. In *Proceedings Workshop on Formal Methods in Software Prractice*, pages 7–15. ACM Press, 1998.

[66] M.B. Dwyer and G.G. Avrunin and J.C. Corbett. Patterns in property specification for finite-state verification. In *21st International Conference on Software Engineering*, pages 411–420. IEEE CS Press, 1999.

[67] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol B: Formal Models and Semantics*, chapter 16, pages 995–1072. Elsevier Publishers B.V., 1990.

[68] E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, **30**(1):1–24, 1985.

[69] E.A. Emerson and J.Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM*, **33**(1):151–178, 1986.

[70] E.A. Emerson and C.S. Jutla. The complexity of tree automata and logics of programs (extended abstract). In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 328–337. IEEE Computer Science Press, 1988.

[71] E.A. Emerson and C.-L. Lei. Temporal reasoning under generalized fairness constraints. In B. Monien and G. Vidal-Naquet, editors, *Symposium on Theoretical Aspects of Computer Science (STACAS)*, volume 210 of *Lecture Notes in Computer Science*, pages 21–36. Springer-Verlag, 1985.

[72] E.A. Emerson and C.-L. Lei. Modalities for model checking: branching time logic strikes back. *Science of Computer Programming*, **8**(3):275–306, 1987.

[73] R. Enders and T. Filkorn and D. Taubner. Generating BDDs for symbolic model checking in CCS. *Distributed Computing*, **6**:155–164, 1993.

[74] R. Floyd. Assigning meaning to programs. In J.J. Schwartz, editor, *Proceedings of Symposium on Applied Mathematics 19*, pages 19–32. American Mathematical Society, 1967.

[75] S. Fortune and J.E. Hopcroft and E.M. Schmidt. The complexity of equivalence and containment for free single variable program schemes. In G. Ausiello and C. Böhm, editors, *Automata, Languages and Programming (ICALP)*, volume 62 of *Lecture Notes in Computer Science*, pages 227–240. Springer-Verlag, 1978.

[76] N. Francez. *Fairness*. Springer-Verlag, 1986.

[77] D. Gabbay and A. Pnueli and S. Shelah and J. Stavi. Completeness results for the future fragment of temporal logic. Manuscript., 1980.

[78] D. Gabbay and A. Pnueli and S. Shelah and J. Stavi. On the temporal analysis of fairness. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 163–173. ACM Press, 1980.

[79] P. Godefroid. Model checking for programming languages using Verisoft. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 174–186. ACM Press, 1997.

[80] R. Gotzhein. Temporal logic and applications–a tutorial. *Computer Networks and ISDN Systems*, **24**:203–218, 1992.

[81] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[82] J.F. Groote and F. Monin and J. van de Pol. Checking verifications of protocols and distributed systems by computer. In R. de Simone and D. Sangiorgi, editors, *Concurrency Theory (CONCUR)*, volume 1432 of *Lecture Notes in Computer Science*, pages 629–655. Springer-Verlag, 1998.

[83] C.A. Gunther and D.A. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, vol B: Formal Models and Semantics*, chapter 12, pages 633–674. Elsevier Publishers B.V., 1990.

[84] A. Gupta. Formal hardware verification methods: a survey. *Formal Methods in System Design*, **1**:151–238, 1992.

[85] B.T. Hailpern. *Verifying Concurrent Processes Using Temporal Logic*, volume 129 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.

[86] J. Hajek. Automatically verified data transfer protocols. In *Proceedings 4th Int. Conf. Computer Communication*, pages 749–756. IEEE Computer Science Press, 1978.

[87] J. Hatcliff and M. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In K.G. Larsen and M. Nielsen, editors, *Concurrency Theory (CONCUR)*, volume 2154 of *Lecture Notes in Computer Science*, pages 39–58. Springer-Verlag, 2001.

[88] K. Havelund and M. Lowry and J. Penix. Formal analysis of a space-craft controller using SPIN. *IEEE Transactions on Software Engineeering*, **27**(8):749–765, 2001.

[89] K. Havelund and T. Pressburger. Model checking Java using Java pathfinder. *Journal on Software Tools and Technology Transfer*, **2**(4):366–381, 2000.

[90] K. Heljanko. Model checking the branching temporal logic CTL. Technical Report Research Report No. 45, Helsinki University of Technology, 1997.

[91] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, **32**(1):137–161, 1985.

[92] T.A. Henzinger and X. Nicollin and J. Sifakis and S. Yovine. Symbolic model checking of real-time systems. *Information and Computation*, **111**(2):193–244, 1994.

[93] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, **12**:576–580, 583, 1969.

[94] R. Hojati and R.K. Brayton and R.P. Kurshan. BDD-based debugging of designs using language containment and fair CTL. In C. Courcoubetis, editor, *Computer-Aided Verification (CAV)*, volume 697 of *Lecture Notes in Computer Science*, pages 41–58. Springer-Verlag, 1993.

[95] G. Holzmann. The theory and practice of a formal method: NewCoRe. In *Proceedings IFIP World Congress*, pages 35–44, 1994.

[96] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[97] G.J. Holzmann. Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems*, **25**:981–1017, 1993.

[98] G.J. Holzmann and E. Najm and A. Serhrouchini. SPIN model checking: an introduction. *Journal on Software Tools and Technology Transfer*, **2**(4):321–327, 2000.

[99] G.J. Holzmann and D. Peled and M. Yannakakis. On nested depth-first search. In *2nd SPIN workshop*, pages 23–32. American Mathematical Sociey, 1996.

[100] J. Hromkovič and S. Seibert and T. Wilke. Translating regular expressions into small ε-free nondeterministic finite automata. *Journal of Computer and System Sciences*, **62**:565–588, 2001.

[101] G. Huet and G. Kahn and Ch. Paulin-Mohring. The Coq proof assistant - a tutorial. Technical Report 178, INRIA, 1995.

[102] M. Huth and M.D. Ryan. *Logic in Computer Science – Modelling and Reasoning about Systems.* Cambridge University Press, 1999.

[103] ISO/ITU-T. *Formal Methods in Conformance Testing.* International Standard, 1996.

[104] H. Iwashita and T. Nakata and F. Hirose. Ctl model checking based on forward state traversal. In *International Conference on Computer Aided Design (ICCAD)*, pages 82–87. IEEE Computer Science Press, 1986.

[105] G.L.J.M. Janssen. *Logics for Digital Circuit Verification: Theory, Algorithms and Applications.* PhD thesis, Eindhoven University of Technology, 1999.

[106] N.D. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences*, **11**:68–75, 1975.

[107] J.A.W. Kamp. *Tense Logic and the Theory of Linear Order.* PhD thesis, Michigan State University, 1968.

[108] S.C. Kleene. *Introduction to Metamathematics.* Van Nostrand, 1950.

[109] S.C. Kleene. Representation of events in nerve nets and finite automata. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, 1956.

[110] R. Koymans. Specifying message buffers requires extending temporal logic. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 191–204. ACM Press, 1987.

[111] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, **27**:333–354, 1983.

[112] S.A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, **16**:83–94, 1963.

[113] T. Kropf. *Introduction to Formal Hardware Verification.* Springer-Verlag, 1998.

[114] O. Kupferman and M.Y. Vardi and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, **47**(2):312–360, 2000.

[115] R. Kurshan. *Computer-aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[116] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineeering*, **3**(2):125–143, 1977.

[117] L. Lamport. Sometimes is sometimes "not never" – on the temporal logic of programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 174–185, 1980.

[118] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, **16**(3):872–923, 1994.

[119] M. Lange and M. Leucker and T. Noll and S. Tobies. Truth – a verification platform for concurrent systems. In *Tool Support for System Specification, Development, and Verification*, Advances in Computing Science. Springer-Verlag, 1999.

[120] F. Laroussinie and N. Markay and Ph. Schnoebelen. Temporal logic with forgettable past. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 383–392. IEEE Computer Science Press, 2002.

[121] C.Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, **38**:985–999, 1959.

[122] N. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[123] C. Lewis. Implication and the algebra of logic. *Mind* **N.F.**, **12**:522–531, 1912.

[124] O. Lichtenstein and A. Pnueli. Checking that finite-state concurrent programs satisfy their linear specification. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 97–107. ACM Press, 1985.

[125] O. Lichtenstein and A. Pnueli and L. Zuck. The glory of the past. In *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer-Verlag, 1985.

[126] P. Liggesmeyer and M. Rothfelder and M. Rettelbach and T. Ackermann. Qualitätssicherung Software-basierter technischer Systeme. *Informatik Spektrum*, **21**:249–258, 1998.

[127] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. *Software Concepts and Tools*, **17**(3):93–102, 1996.

[128] O. Maler and Z. Manna and A. Pnueli. From timed to hybrid systems. In J.W. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real-Time: Theory and Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 447–484. Springer-Verlag, 1992.

[129] Z. Manna and S. Ness and J. Vuillemin. Inductive methods for proving properties of programs. *Communications of the ACM*, **16**(8):491–502, 1973.

[130] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 377–408. ACM Press, 1990.

[131] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1992.

[132] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Safety*. Springer-Verlag, 1995.

[133] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[134] R. McNaughton. Testing and generating infinite seqeunces by a finite automaton. *Information and Control*, **9**:521–530, 1966.

[135] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, **9**(1):38–47, 1960.

[136] Ch. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design: OBDD - Foundations and Applications*. Springer-Verlag, 1998.

[137] T.F. Melham and M.J.C. Gordon. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[138] S. Merz. Model checking: a tutorial. In F. Cassez, C. Jard, B. Rozoy, and M.D. Ryan, editors, *Modelling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-Verlag, 2001.

[139] A.R. Meyer and M.J. Fischer. Economy of description by automata, grammars and formal systems. In *IEEE Symposium on Switching and Automata Theory*, pages 188–191. IEEE Computer Science Press, 1971.

[140] A.R. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential time. In *IEEE Symposium on Switchinh and Automata Theory*, pages 125–129. IEEE Computer Science Press, 1972.

[141] F.L. Morris and C.B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, **6**(2):139–143, 1984.

[142] D.E. Muller and A. Saoudi and P.E. Schupp. Weak alternating automata give a simple explanation why most temporal and dynamic logids are decidable in exponential time. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 422–427. IEEE Computer Science Press, 1988.

[143] G.J. Myers. *The Art of Software Testing*. John Wiley & Sons, 1979.

[144] X. Nicollin and J.-L. Richier and J. Sifakis and J. Voiron. ATP: an algebra for timed processes. In *IFIP TC2 Working Conference on Programming Concepts and Methods*, IFIP series, pages 402–xxx, 1990.

[145] A. Olivero and J. Sifakis and S. Yovine. Using abstractions for the verification of linear hybrid systems. In D. Dill, editor, *Computer-Aided Verification (CAV)*, volume 818 of *Lecture Notes in Computer Science*, pages 81–94. Springer-Verlag, 1994.

[146] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, **6**:319–340, 1976.

[147] S. Owre and S. Rajan and J. Rushby and N. Shankar and M.K. Srivas. PVS: combining specification, proof checking and model checking. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification (CAV)*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer-Verlag, 1996.

[148] L. Paulson. *Isabelle: a Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[149] D. Peled. *Software Reliability Methods*. Springer-Verlag, 2001.

[150] L. Pierre. Describing and verifying synchronous circuits with the Boyer-Moore theorem prover. In P.E. Camurati and H. Eveking, editors, *IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 987 of *Lecture Notes in Computer Science*, pages 35–55. Springer-Verlag, 1995.

[151] A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 46–57, 1977.

[152] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In K. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO Advanced Summer Institutes*, pages 123–144, 1985.

[153] E. Poll and J. van den Berg and B. Jacobs. Formal specification of the JavaCard API in JML: the APDU class. *Computer Networks*, **36**(4):407–421, 2001.

[154] A. Prior. *Time and Modality*. Oxford University Press, 1957.

[155] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1982.

[156] M.O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, **3**:115–125, 1959.

[157] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference on Computer Aided Design (ICCAD)*, pages 139–144. IEEE Computer Science Press, 1993.

[158] J. Rushby. Formal methods and the certification of critical systems. Technical Report CSL-93-7, SRI International, 1993. (also issued as *Formal Mtehods and Digital System Validation*, NASA CR 4551).

[159] T.C. Ruys and E. Brinksma. Managing the verification trajectory. *Journal on Software Tools and Technology Transfer*, **4**(2):246–259, 2003.

[160] S. Safra. On the complexity of ω-automata. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 319–327. IEEE Computer Science Press, 1988.

[161] A.L. Sangiovanni-Vincentelli and P.C. McGeer and A. Saldanha. Verification of electronic systems. In *ACM/IEEE Design Automation Conference (DAC)*. ACM Press, 1996.

[162] T. Schlipf and T. Buechner and R. Fritz and M. Helms and J. Koehl. Formal verification made easy. *IBM Journal of Research & Development*, **41**(4/5):549–566, 1997.

[163] S. Schneider. *Specifying Real-Time Systems in Timed CSP*. Prentice-Hall, 2000.

[164] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, **25**(3):225–262, 1993.

[165] D. Sieling and I. Wegener. Reduction of OBDDs in linear time. *Information Processing Letters*, **48**:139–144, 1993.

[166] A.P. Sistla. On the characterization of safety and liveness properties in temporal logic. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 39–48. ACM Press, 1985.

[167] A.P. Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, **6**(5):495–512, 1994.

[168] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, **32**(3):733–749, 1985.

[169] A.P. Sistla and M.Y. Vardi and P. Wolper. The complementation problem for büchi automata with applications to temporal logic. *Theoretical Computer Science*, **49**:217–237, 1987.

[170] F. Somenzi. Efficient manipulation of decision diagrams. *Journal on Software Tools and Technology Transfer*, **3**(2):171–182, 2001.

[171] J. Staunstrup and H.R. Andersen and J. Lind-Nielsen and K.G. Larsen and G. Behrmann and K. Kristoffersen and H. Leerberg and N.B. Theilgaard. Practical verification of embedded software. *IEEE Computer*, **33**(5):68–75, 2000.

[172] N. Storey. *Safety-Critical Computer Systems*. Addison-Wesley, 1996.

[173] R. Tarjan. Depth-first and linear graph algorithms. *SIAM Journal on Computing*, **1**(2):146–160, 1972.

[174] W. Thomas. Computation tree logic and ω-regular languages. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time, and Partial Order in Logics and Model for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 690–713. Springer-Verlag, 1988.

[175] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 4, pages 133–191. Elsevier Publishers B.V., 1990.

[176] W. Thomas. Boolesche Logik und Büchi-Elgot-Trakhtenbrot-Logik in der Beschreibung diskreter Systeme. In P. Horster, editor, *Angewandte Mathematik, insbesondere Informatik*, pages 282–300. Vieweg, 1999.

[177] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, **11**(6):419–422, 1968.

[178] G.J. Tretmans and K. Wijbrans and M. Chaudron. Software engineering with formal methods: the development of a storm surge barrier control system. *Formal Methods in System Design*, **19**:195–215, 2001.

[179] A.M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. Univ. Math. Lab, Cambridge, 1949.

[180] M.Y. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for Concurrency – Structure versus Automata (8th Banff Higher Order Workshop)*, volume 1043 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[181] M.Y. Vardi. Branching versus linear time: Final showdown. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer-Verlag, 2001.

[182] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, **115**(1):1–37, 1994.

[183] B. Vergauwen and J. Lewi. A linear local model checking algorithm for CTL. In E. Best, editor, *Concurrency Theory (CONCUR)*, volume 715, pages 447–461, 1993.

[184] W. Visser and H. Barringer. Practical CTL* model checking: should SPIN be extended? *Journal on Software Tools and Technology Transfer*, **2**(4):350–365, 2000.

[185] I. Wegener. *Branching Programs and Binary Decision Diagrams – Theory and Application*, volume 4 of *SIAM Monographs on Discrete Mathematics and Applications*. SIAM, 2000.

[186] I. Wegener. BDDs – design, analysis, complexity, and analysis. *Discrete Applied Mathematics*, 2003. (to appear).

[187] C.H. West. An automated technique for communications protocol validation. *IEEE Transactions on Communications*, **26**(8):1271–1275, 1978.

[188] C.H. West. Protocol validation in complex systems. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 303–312, 1989.

[189] J.A. Whittaker. What is software testing? why is it so hard? *IEEE Software*, **17**(1):70–80, 2000.

[190] P. Wolper. An introduction to model checking. Position statement for panel discussion at the Software Quality workshop, 1995.

[191] A. Xie and P.A. Beerel. Implicit enumeration of strongly connected components and an application to formal verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **19**(10):1225–1230, 2000.

[192] B. Yang and R.E. Bryant and D.R. O'Hallaron and A. Biere and O. Coudert and G. Janssen and R.K. Ranjan and F. Somenzi. A performance study of BDD-based model checking. In *Formal Methods in Computer-Aided Design*, volume 1522 of *Lecture Notes in Computer Science*, pages 255–289. Springer-Verlag, 1998.

[193] W. Yi. CCS + time = an interleaving model for real-time systems. In J.L̇each Albert, B. Monien, and M.R̃odriguez Artalejo, editors, *Automata, Languages and Programming (ICALP)*, volume 510 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1991.

[194] M. Yoeli. *Formal Verification of Hardware Design*. IEEE Computer Science Press, 1990.