Concepts, Algorithms, and Tools

for

Model Checking

Joost-Pieter Katoen

Lehrstuhl für Informatik VII

Friedrich-Alexander Universität Erlangen-Nürnberg

Lecture Notes of the Course

"Mechanised Validation of Parallel Systems"

(course number 10359)

Semester 1998/1999

Dedicated to Ulrich Herzog

on the occasion of his 60th birthday

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.

On the relevance of system validation

In daily life we are more and more confronted with information technology, either explicitly (by using PCs, Internet, Personal Digital Assistants, etc.) or implicitly by using devices like TVs, electric razors, mobile phones, cars, public transportation and so forth. In 1995 it has been estimated that people are in contact with about 25 "information processing devices" per day. Due to the high integration of information technology in all kinds of applications — it is estimated that 20% of the total development costs of transportation devices such as cars, trains and air-planes are for computer-based components — we increasingly rely on the reliability (or should we say unreliability?) of software and hardware components. Clearly, we do not accept that our mobile phone is malfunctioning or that our video recorder reacts unexpectedly and wrongly to commands we issue via the remote control unit. And these errors do in a sense still have little impact: however, errors in safety-critical systems such as nuclear power plants or flight control systems are unacceptable and have a huge impact. A major problem though is that the complexity of systems grows rapidly, and hence their vulnerability for errors.

Apart from (important) safety aspects, errors can be very expensive, in particular if they are demonstrated during system operation, i.e. after the product has been launched on the market. There are several impressive recent stories about those experiences. For instance, the error in Intel's Pentium floating-point division unit is estimated to have caused a loss of about 500 million US dollars. A similar loss has been estimated for the crash of the Ariane-5 missile, which seems to be due to an error in the software that controls the flight direction.

All in all, it is fair to say that *system validation*, the process of determining the correctness of specifications, designs and products, is becoming an increasingly important activity. System validation is a technique to support the quality control of the system design. Current practice in software engineering shows that system designs are checked to a large extent by humans (so-called "peer reviewing") and by dynamic testing (design tests, system tests, field tests etc.). Little tool-support is involved, let alone the application of techniques and tools with a sound mathematical basis. Due to the increasing magnitude and complexity of systems and the pressure to reduce system design and development time ("time to market"), the support of the system validation process through techniques and tools that facilitate the *automated* analysis of the correctness is essential. As Wolper has stated in the context of software verification by humans[1]:

"Manual verification is at least as likely to be wrong as the program itself"

Techniques for system validation

Important validation techniques are peer reviewing, (simulation and) testing, formal verification, and model checking. Testing is an operational way to check whether a given system realization — an entity consisting of soft- and/or hardware —- conforms to an abstract specification. By nature, testing can be applied

[1]P. Wolper. Verification: dreams and reality. Inaugural lecture of the course *"The algorithmic verification of reactive systems"*.

only after a prototype implementation of the system has been realized. Formal verification, as opposed to testing, works on models (rather than implementations) and amounts to a mathematical proof of the correctness of a system (i.e., its model). Both techniques can (and partially are) supported by tools. For instance, in the field of testing interest is increasing in the development of algorithms and tools for the automated generation and selection of tests, starting from a formal system specification. In formal verification, theorem provers and proof checkers have been shown to be an important support, though quite some expertise is usually required to handle them.

Model checking

In these lecture notes we concentrate on a different validation technique that is known as *model checking*. To put it in a nutshell, model checking is an automated technique that, given a finite-state model of a system and a property stated in some appropriate logical formalism (such as temporal logic), systematically checks the validity of this property. Model checking is a general approach and is applied in areas like hardware verification and software engineering. Due to its success in several projects and the high degree of support from tools, there is an increasing interest in industry in model checking — various companies have started their own research groups and sometimes have developed their own in-house model checkers. For instance, Intel has started 3 validation laboratories for the verification of new chip designs. An interesting aspect of model checking is that it supports partial verification: a design can be verified against a partial specification by considering only a subset of all requirements.

Scope of these notes

These lecture notes are concerned with the concepts, algorithms, and tools for model checking. The concepts of model checking have their roots in sound mathematical foundations such as logic and automata theory, data structures, graph algorithms. Although some basic knowledge (such as that obtained in an undergraduate course) on these topics is required, these concepts will be extensively treated in these notes. We will deal with three different types of model checking: model checking of linear, branching, and real-time (branching) temporal logic. The first two types concentrate on the functional or qualitative aspects of systems, whereas the latter type of model checking allows in addition some quantitative analysis. For each of these forms of model checking we deal with the concepts, algorithms that will be developed systematically on the basis of the mathematical concepts, and tools whose functionality is illustrated by a small case study. In presenting the concepts and algorithms we concentrate on the basic ingredients and do not treat efficiency-improving techniques that — as we firmly believe — would unnecessarily distract the reader from the essence of model checking. Instead, we devote a separate chapter to techniques for state space reduction. We do not underestimate the role of these techniques — they have been (and still are) the key for the success of model checking — but we support Dijkstra's principle of separation of concerns, and therefore treat these techniques separately.

Acknowledgements

These lecture notes have been developed while teaching the graduate course on "mechanized validation of parallel systems" at the University of Erlangen-Nürnberg, Germany (winter seminars in 1997/1998 and 1998/1999). All chapters of these notes were covered by 15 lectures of 2 hours each. The part on real-time model checking has in addition been used in a course on "model checking" at the University of Pisa, Italy (winter of 1997). I would like to thank Ulrich Herzog for supporting me in setting up this course in Erlangen, for giving me the time to work on the development of the course and the writing of these notes. I dedicate these notes to him with great pleasure. Stefania Gnesi (CNR/I.E.I) is kindly acknowledged for giving me the opportunity to contribute to her course in Pisa. During the courses I received several suggestions for improvements from my colleagues and by students. Thanks to all! In particular, I would like to thank Holger Hermanns, Markus Siegle and Lennard Kerber for their critical remarks and discussion. Detailed comments on the various chapters were provided by Ulrich Klehmet and Mieke Massink (C.N.R./CNUCE) and were very helpful. I thank them for their efforts. Clearly, the remaining errors and unclarities are

fully my responsibility. Finally, I would like to thank Robert Bell who checked my English of the first three chapters and suggested several improvements. In case you have suggestions for improvements or find any mistakes, I would be happy to be informed. My e-mail address is `katoen@cs.utwente.nl`.

Further reading

Personally I consider these notes as an introduction to the concepts, algorithms, and tools for model checking. Although a few (good) books on model checking are currently available there does not exist, at least to the best of my knowledge, an introductory work that covers model checking of linear, branching and real-time model checking and provides an introduction to system validation. Due to lack of space and time, there are various aspects of model checking that are not covered in these lecture notes. An important aspect is the usage of these techniques in industrial case studies. For those readers who are interested in more information on specific model checking techniques I suggest the following books:

1. G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.
2. K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
3. R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, 1994.
4. G. Bruns. *Distributed Systems Analysis with CCS.* Prentice-Hall, 1997.
5. M. Huth and M.D. Ryan. *Logic in Computer Science – Modelling and Reasoning about Systems.* Cambridge University Press, 1999 (to appear).

In addition each chapter contains a list of references that can be consulted in case a deeper understanding is desired.

March 1999

Joost-Pieter Katoen
Pretzfeld (Frankonia, Germany)

Chapter 1

System Validation

1.1 Purpose of system validation

Introduction and motivation

Systems that — in one way or another — are based on information processing increasingly provide critical services to users. Mobile telephone systems, for instance, are used in various circumstances (as in ambulances) where malfunctioning of the software can have disastrous consequences. Information processing plays a significant role in process control systems, as in nuclear power plants or in chemical industry where, evidently, errors (in software) are highly undesirable. Other typical examples of such safety-critical systems are radiation machines in hospitals and storm surge barriers.

These are obvious examples where reliability is a key issue and where the correctness of a system is of vital importance. However, information technology is more and more entering our daily life. In "non-information processing" systems, such as electric razors, audio equipment, and TV-sets (where high-end TVs are nowadays equipped with 2 Megabyte of software), the amount of soft- and hardware is increasing rapidly. For transport systems such as cars, trains and airplanes, information technology is coming to play a dominant role; expectations are that about 20% of the total development costs of those vehicles will be needed for information technology. Given this increasing integration of information processing into various kinds of (critical) applications, it is fair to state that:

> *The reliability of information processing is a key issue in the system design process.*

What is common practice on maintaining the reliability of, for instance, software systems? Usually the design is started with a requirements analysis where the desires of the client(s) are extensively analyzed and defined. After traversing several distinct design phases, at the end of the design process a (prototype) design is obtained. The process of establishing that a design fulfills its requirements is referred to as *system validation*. Schematically this strategy is depicted in Figure 1.1. What is common practice on when to perform validation? Clearly,

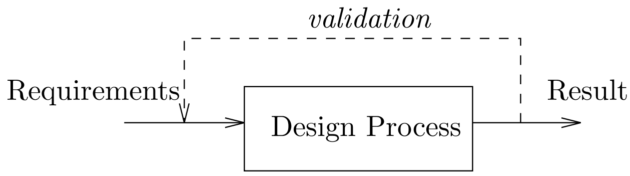


Figure 1.1: Schematic view of (a posteriori) system validation

checking for faults only at the end of the design trajectory is not acceptable: if an error is found it takes too much effort to repair it, since the whole design trajectory needs to be re-traversed in order to see where and how the error could arise. This is rather costly. It is therefore more common practice to check on-the-fly, i.e. while the system is being designed. If an error is determined in this setting, ideally only the design phase up to the previous validation step needs to be re-examined. This reduces the costs significantly. This is depicted schematically in Figure 1.2.

Two techniques are widely applied in practice to ensure that the final result does what it originally is supposed to do, *peer reviewing* and *testing*. Investigations in the software engineering branch, for instance, show that 80% (!) of all projects use peer reviewing. Peer reviewing is a completely manual activity

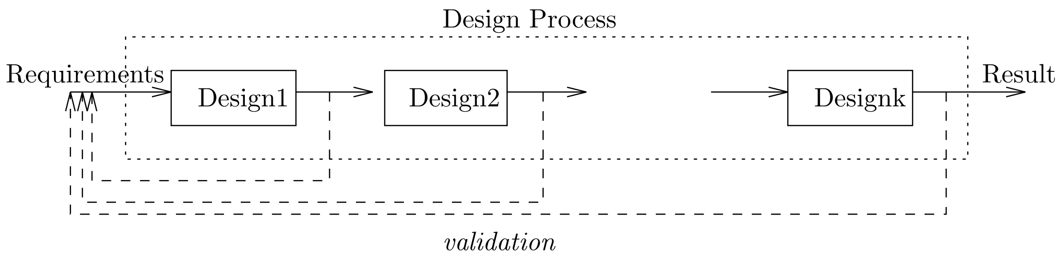


Figure 1.2: Schematic view of on-the-fly system validation

in which (part of) a design is reviewed by a team of developers that (in order to increase the effect of reviewing) have not been involved in the design process of that particular part. Testing is an operational way to check whether a given system realization conforms to the original abstract specification. Although testing is an (the) essential validation technique for checking the correctness of real implementations, it is usually applied in an ad-hoc manner — tests are, for instance, generated by hand — and tool-support has received only scant attention. (Testing is covered more extensively later on in this chapter.)

System validation is an important step in system design:

> *In most designs more time and effort is spent on validation than on construction!*

This is not so surprising, since errors can be very expensive. The error in Intel's Pentium floating-point division unit is estimated to have caused a loss of about 500 million US dollars. Consider also the (software) mistake in the missile Ariane-5 (costs of about 500 million dollars) or the mistake in Denver's baggage handling system that postponed the opening of the new airport for 9 months (at 1.1 million dollar per day).

System validation as part of the design process

In Germany interviews with several software engineering companies have resulted in some interesting figures for the costs of error repairment and statements about the stages in the design process where errors are introduced and detected (Liggesmeyer et. al, 1998). These investigations show — once more — that integration of system validation into the design process should take place at an early stage, particularly from an economical point of view (cf. Figure 1.3). It should not

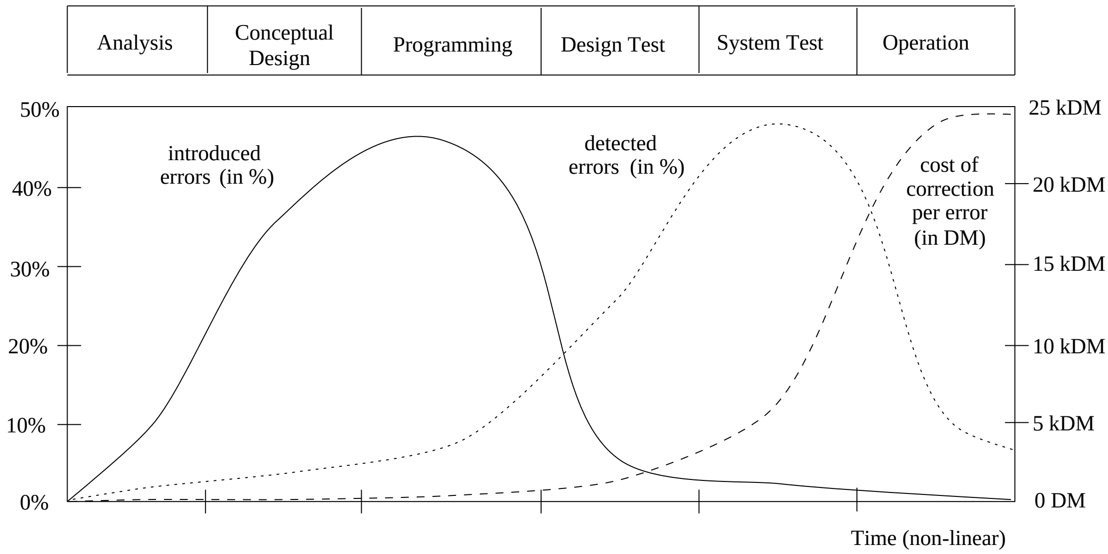


Figure 1.3: System life cycle and error introduction, detection and costs of repairment

surprise the reader that about 50% of all errors are introduced during the programming phase, i.e. the phase in which actual code is generated. The conceptual design phase also contributes significantly to the introduction of errors. Whereas only about 15% of all errors have been detected before the test phases start, it is evident that most errors are found during testing, in particular during system integration tests, where software modules are glued together and the entire system is investigated.

Errors that are detected before the test phase can on average be repaired very cheaply: about 500 German Marks (i.e. about 320 US dollar) per error. In the design test, while testing the software modules on a stand-alone basis, this cost increases to 2,000 DM, reaching a maximum of 25,000 DM per error when an error is demonstrated during system operation (field tests). It is clear that it is of vital importance to detect errors as early as possible in the system design process: the costs to repair errors are substantially lower, and the impact of errors on the rest of the design is less substantial. Without overestimating, we can fairly conclude that

> *There is a strong need for an early integration of system validation in the design process.*

Support by systematic techniques: formal methods

Nowadays, computer systems are composed of several subsystems, such as parallel and distributed systems (e.g. computer/telephony networks, chips that consist of many communicating components, or high-speed parallel computers connected by some shared memory). Due to the increase in the magnitude and complexity of information processing systems, errors can easily and unexpectedly occur in their design. Reasoning about such systems becomes more and more complicated. Thus:

> *There is a strong need for the use of advanced, systematic techniques and tools that support the system validation process.*

Whereas many validation techniques are ad-hoc and based on system specifications posed in natural language, we will concentrate in these lecture notes on validation based on *formal methods*. Using formal methods, system designs can be defined in terms of precise and unambiguous specifications that provide the basis for a systematic analysis. Important validation techniques based on formal methods are *testing*, *simulation*, *formal verification*, and *model checking*. In the following we briefly consider testing, simulation and formal verification, before concentrating on the main topic of these lecture notes, model checking. Although system validation based on formal methods has a great potential, and is well-accepted in fields like hardware verification and the verification of communication protocols, its use in software engineering is still limited. For instance, according to the German study mentioned before in 10-15% of all projects formal methods are used.

Reactive systems

In these lecture notes we will concentrate on the validation of so-called *reactive systems*. Reactive systems — this term has been coined by Pnueli (1985) — are characterized by a continuous interaction with their environment. They typically continuously receive inputs from their environment and, usually within quite a short delay, react on these inputs. Typical examples of reactive systems are operating systems, aircraft control systems, communication protocols, and process control software. For instance, a control program of a chemical process receives control signals regularly, like temperature and pressure, at several points in the process. Based on this information, the program can decide to turn on the heating elements, to switch off a pump, and so forth. As soon as a dangerous situation is anticipated, for example the pressure in the tank exceeds certain thresholds, the control software needs to take appropriate action. Usually reactive systems are rather complex: the nature of their interaction with the environment can be intricate and they typically have a distributed and concurrent nature.

As argued above, correctness and well-functioning of reactive systems is crucial. To obtain correctly functioning and dependable reactive systems a coherent and well-defined methodology is needed in which different phases can be distinguished.

- In the first phase, a thorough investigation of requirements is needed and as a result a requirement specification is obtained.
- Secondly, a conceptual design phase results in an abstract design specification. This specification needs to be validated for internal consistency and it needs to be checked against the requirement specification. This validation process can be supported by formal verification simulation and model checking techniques, where a model (like a finite-state automaton) of the abstract design specification can be extensively checked.
- Once a trustworthy specification has been obtained, the third phase consists of building a system that implements the abstract specification. Typically, testing is a useful technique to support the validation of the realization versus the original requirement specification.

1.2 Simulation

Simulation is based on a model that describes the possible behavior of the system design at hand. This model is executable in some sense, such that a software tool (called a simulator) can determine the system's behavior on the basis of some scenarios. In this way the user gets some insight on the reactions of the system on certain stimuli. The scenarios can be either provided by the user or by a tool, like a random generator that provides random scenarios. Simulation is typically useful for a quick, first assessment of the quality of a design. It is less suited to find subtle errors: it is infeasible (and mostly even impossible) to simulate all representative scenarios.

1.3 Testing

A widely applied and traditional way of validating the correctness of a design is by means of *testing*. In testing one takes the implementation of the system as realized (a piece of software, hardware, or a combination thereof), stimulates it with certain (probably well-chosen) inputs, called tests, and observes the reaction of the system. Finally, it is checked whether the reaction of the system conforms to the required output. The principle of testing is almost the same as that of simulation, the important distinction being that testing is performed on the real, executing, system implementation, whereas simulation is based on a model of the system.

Testing is a validation technique that is most used in practice — in software engineering, for instance, in all projects some form of testing is used — but almost exclusively on the basis of informal and heuristic methods. Since testing is based on observing only a small subset of all possible instances of system behavior, it can never be complete. That is to say, in Dijkstra's words:

> *Testing can only show the presence of errors, never their absence.*

Testing is, however, complementary to formal verification and model checking, that are based on a mathematical model of the system rather than on a real, executing system. Since testing can be applied to the real implementation, it is useful in most cases:

- where a valid and reliable model of the system is difficult to build due to complexity,
- where system parts cannot be formally modeled (as physical devices) or
- where the model is proprietary (e.g. in case of third-party testing).

Testing is usually the dominating technique for system validation and consists of applying a number of tests that have been obtained in an ad-hoc or heuristic manner to an implementation. Recently, however, the interest and possibilities of applying formal methods to testing are increasing. For instance, in the area of communication protocols this kind of research has resulted in a draft international standard on "formal methods in conformance testing" (ISO, 1996). There the process of testing is partitioned into several phases:

1. In the *test generation* phase, abstract descriptions of tests, so-called test cases, are systematically generated starting from a precise and unambiguous specification of the required properties in the specification. The test cases must be guaranteed to test these properties.
2. In the *test selection* phase, a representative set of abstract test cases is selected.
3. In the *test implementation* phase, the abstract test cases are transformed into executable test cases by compiling them or implementing them.
4. In the *test execution* phase, the executable test cases are applied to the implementation under test by executing them on a test execution system. The results of the test execution are observed and logged.
5. In the *test analysis* phase, the logged results are analyzed to decide whether they comply with the expected results.

The different phases may, and in practice often do, overlap, especially the last phases.

Testing is a technique that can be applied both to prototypes, in the form of systematic simulation, and to final products. Two basic approaches are *transparent box testing*, where the internal structure of an implementation can be observed and sometimes partially controlled (i.e. stimulated), and *black box testing*, where only the communication between the system under test and its environment can be tested and where the internal structure is completely hidden to the tester. In practical circumstances, testing is mostly somewhere in between these two extremes, and is sometimes referred to as *grey box testing*.

1.4 Formal verification

A complementary technique to simulation and testing is to *prove* that a system operates correctly, in a similar way as one can prove that, for example, the square of an even number is always an even number. The term for this mathematical demonstration of the correctness of a system is *(formal) verification*. The basic idea is to construct a formal (i.e. mathematical) *model* of the system under investigation which represents the possible behavior of the system. In addition, the correctness requirements are written in a formal *requirement specification* that represents the desirable behavior of the system. Based on these two specifications one checks by formal proof whether the possible behavior "agrees with" the desired behavior. Since the verification is treated in a mathematical fashion, the notion of "agree with" can be made precise, and verification amounts to proving or disproving the correctness with respect to this formal correctness notion.

In summary formal verification requires:

1. A model of the system, typically consisting of
   - a set of states, incorporating information about values of variables, program counters and the like, and
   - a transition relation, that describes how the system can change from

one state to another.

2. A specification method for expressing requirements in a formal way.
3. A set of proof rules to determine whether the model satisfies the stated requirements.

To obtain a more concrete feeling of what is meant we consider the way in which sequential programs can be formally verified.

Verifying sequential programs

This approach can be used to prove the correctness of sequential algorithms such as quick-sort, or the computation of the greatest common divisor of two integers. In a nutshell this works as follows. One starts by formalizing the desired behavior by using pre- and postconditions, formulas in predicate logic. The syntax of such formulas is, for instance, defined by

$$\phi ::= p \;\Big|\; \neg\,\phi \;\Big|\; \phi \,\wedge\, \phi$$

where p is a basic statement (like "x equals 2"), $\neg$ denotes negation and $\vee$ denotes disjunction. The other boolean connectives can be defined by: $\phi \vee \psi = \neg\,(\neg\,\phi \vee \neg\,\psi)$, true $= \phi \vee \neg\,\phi$, false $= \neg\,$true and $\phi \Rightarrow \psi = \neg\,\phi \vee \psi$. For simplicity we omit universal and existential quantification.

A *precondition* describes the set of interesting start states (i.e. the allowed input(s)), and the *postcondition* describes the set of desired final states (i.e. the required output(s)). Once the pre- and postcondition are formalized, the algorithm is coded in some abstract pseudo-code language (e.g. Dijkstra's guarded command language) and it is proven in a step-by-step fashion that the program satisfies its specification.

To construct the proofs, a proof system, that is, a set of proof rules, is used. These proof rules usually correspond to program constructs. They are written:

$$\{\,\phi\,\}\; S\; \{\,\psi\,\}$$

where ϕ is a precondition, S a program statement, and ψ a postcondition. The triple $\{\,\phi\,\}\,S\,\{\,\psi\,\}$ is known as the Hoare triple (Hoare 1969), christened to one of the pioneers in the field of formal verification of computer programs. There are two possible interpretations of Hoare triples, depending on whether one considers partial or total correctness.

- The formula $\{\,\phi\,\}S\{\,\psi\,\}$ is called *partially correct* if any terminating computation of S that starts in a state satisfying ϕ, terminates in a state satisfying ψ.
- $\{\,\phi\,\}\,S\,\{\,\psi\,\}$ is called *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, that is not terminates. In the following explanations we treat partial correctness unless stated otherwise.

The basic idea of the approach by Hoare (and others) is to prove the correctness of programs at a syntactical level, only using triples of the above form. To illustrate his approach we will briefly treat a proof system for a simple set of deterministic sequential programs. A program is called deterministic if it always provides the same result when provided a given input. These sequential programs are constructed according to the following grammar:

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

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.

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. A proof system for sequential deterministic programs is given in Table 1.1.

Axiom for **skip**	$\{\,\phi\,\}\ \mathbf{skip}\ \{\,\phi\,\}$
Axiom for assignment	$\{\,\phi[x := k]\,\}\ x := k\ \{\,\phi\,\}$
Sequential composition	$\dfrac{\{\,\phi\,\}\ S_1\ \{\,\chi\,\} \wedge \{\,\chi\,\}\ S_2\ \{\,\psi\,\}}{\{\,\phi\,\}\ S_1\ ;\ S_2\ \{\,\psi\,\}}$
Alternative	$\dfrac{\{\,\phi \wedge B\,\}\ S_1\ \{\,\psi\,\} \wedge \{\,\phi \wedge \neg B\,\}\ S_2\ \{\,\psi\,\}}{\{\,\phi\,\}\ \mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\ \{\,\psi\,\}}$
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 1.1: Proof system for partial correctness of sequential programs

The proof rule for the skip-statement, the statement that does nothing, is what one would expect: under any condition, if 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, e.g.

$$\{\,k^2 \text{ is even and } k = y\,\}\ x := k\ \{\,x^2 \text{ is even and } x = y\,\}.$$

The procedure of starting the proof from a postcondition is usually applied successively to parts of the program in a way such that finally the precondition of the entire program can be proved. The rule for sequential composition uses an intermediate predicate χ that characterizes the final state of S_1 and the starting state of S_2. The rule for alternative composition uses the boolean B whose value determines whether S_1 or S_2 is executed.

The proof rule for iteration needs some explanation. Stated in words, this rule states that predicate ϕ holds after the termination of **while** B **do** S **od** if the validity of ϕ can be maintained during each execution of the iteration-body

S. This explains why ϕ is called an invariant. One of the main difficulties in the proof of programs in this approach is to find appropriate invariants. In particular, this complicates the complete automation of these types of proofs.

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 facilitates the application of other proof rules. In particular the rule allows replacing pre- and postconditions by equivalent ones. It should be noted, though, that proving implications like $\phi \Rightarrow \phi'$ is in general undecidable.

We must stress that the above proof system allows the establishment of the relation between inputs and outputs of composed programs by only considering such relations for program parts. Proof rules and proof systems that exhibit this property are called *compositional*. For instance, the proof rule for sequential composition allows the correctness of the composed program $S_1 \, ; \, S_2$ to be established by considering the pre- and postconditions of its components S_1 and S_2.

Let us briefly consider total correctness. The proof system in Table 1.1 is insufficient to prove termination of sequential programs. The only syntactical construct that can possibly lead to divergent (i.e. non-terminating) computations is iteration. In order to prove termination, the idea is, therefore, to strengthen the proof rule for iteration by:

$$\frac{\{\,\phi \wedge B\,\}\; S \;\{\,\phi\,\}, \{\,\phi \wedge B \wedge n = N\,\}\; S\;\{\,n < N\,\}, \phi \Rightarrow n \geqslant 0}{\{\,\phi\,\}\;\textbf{while}\; B \;\textbf{do}\; S\;\textbf{od}\;\{\,\phi \wedge \neg B\,\}}$$

Here, the auxiliary variable N does not occur in ϕ, B, n, or S. The idea is that N is the initial value of n, and that at each iteration the value of n decreases (but remains positive). This construction precisely avoids infinite computations, since n cannot be decreased infinitely often without violating the condition $n \geqslant 0$. The variable n is known as the variant function.

Formal verification of parallel systems

For statements S_1 and S_2 let the construct $S_1 \parallel S_2$ denote the parallel composition of these statements. The major aim of applying formal verification to parallel programs is to obtain a proof rule such as:

$$\frac{\{\,\phi\,\}\,S_1\,\{\,\psi\,\},\ \{\,\phi'\,\}\,S_2\,\{\,\psi'\,\}}{\{\,\phi \wedge \phi'\,\}\,S_1 \parallel S_2\,\{\,\psi \wedge \psi'\,\}}$$

Such a proof rule would allow the verification of parallel systems in a compositional way in the same way as verifying sequential programs by considering the parts separately. Due to interaction between S_1 and S_2, albeit in the form of access to shared variables or by exchanging messages, this rule is unfortunately not valid in general. Starting with the work of Owicki and Gries (1976) much effort has been devoted to obtaining proof rules of the above form. There are several reasons why the achievement of this is far from straightforward.

The introduction of 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 these common variables are accessed. For instance, if S_1 is $x := x{+}2$, S_1' is $x := x{+}1\,;\, x := x{+}1$, and S_2 is $x := 0$, the value of x at termination of $S_1 \parallel S_2$ can be either 0 or 2, and the value of x at termination of $S_1' \parallel S_2$ can be 0, 1 or 2. The different outcomes for x depend on the order of execution of the statements in S_1 and S_2, or S_1' and S_2. Moreover, although the input-output behavior of S_1 and S_1' is obviously the same (increasing x by 2), there is no guarantee that this is true in an identical parallel context.

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. Instead it is important to be able to make, in addition, statements about what happens *during* computation. So properties should not only to refer to start and end-states, but the expression of properties about the executions themselves is needed.

The main problem of the classical approach for the verification of parallel and reactive systems as explained before is that it is completely focused 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.

Various efforts have been made to generalize the classical formal verification approach towards parallel programs, starting from the pioneering work of Owicki & Gries. Due to the interaction between components the proof rules are usually rather complex, and the development of a fully compositional proof system for parallel systems that interact via shared variables or via (synchronous or asynchronous) message passing has turned out to be difficult. The reader who is interested in such proof systems is referred to the seminal work (Apt & Olderog, 1997). For realistic systems, proofs in this style usually become very large — due to possible interactions between parallel programs $N \times M$ additional proof obligations have to be checked for parallel programs of length N and M in the approach of Owicki & Gries — and require quite some interaction and guidance (for instance, in the form of looking for appropriate invariants) from the user. As a result, such proofs are rather lengthy, tedious, and quite vulnerable to errors. Besides, the organization of proofs of such complexity in a comprehensible form is difficult.

The use of so-called *proof assistants*, software tools that assist the user in obtaining mathematical proofs, and *theorem provers*, tools that generate the proof of a certain theorem, may overcome these problems to some extent; their use in formal verification is a research topic that currently attracts a strong interest. We discuss such tools in some more detail later on in this chapter.

Temporal logic

As we have argued above, the correctness of a reactive system refers to the behavior of the system over time, it does not refer only to the input-output relationship of a computation (as pre- and postconditions do), since usually computations of reactive systems do not terminate. Consider, for instance, a communication protocol between two entities, a sender P and a receiver, which are connected via some bidirectional communication means. A property like

"if process P sends a message, then it will not send the next message
until it receives an acknowledgement"

cannot be formulated in a pre- and postcondition way. To facilitate the formal specification of these type of properties, propositional logic has been extended by some operators that refer to the behavior of a system over time. U (until) and G (globally) are examples of operators that refer to sequences of states (as executions). $\phi \, \mathsf{U} \, \psi$ means that property ϕ holds in all states until a state is reached in which ψ holds, and $\mathsf{G} \, \phi$ means that always, that is in all future states, ϕ holds. Using these operators we can formalize the above statement for the protocol by, for instance,

$$\mathsf{G} \, [\, \mathrm{snd}_P(m) \;\Rightarrow\; \neg \, (\mathrm{snd}_P(\mathrm{nxt}(m)) \, \mathsf{U} \, \mathrm{rcv}_P(\mathrm{ack})) \,].$$

Stated in words, if a message m is sent by process P, then there will be no transmission by P of some next message ($\mathrm{nxt}(m)$) until an acknowledgement has been received by P.

Logics extended by operators that allow the expression of properties about executions, in particular those that can express properties about the relative order between events, are called *temporal logics*. Such logics are rather well-established and have their origins in other fields many decades ago. The introduction of these logics into computer science is due to Pnueli (1977). Temporal logic is a well-accepted and commonly used specification technique for expressing properties of computations (of reactive systems) at a rather high level of abstraction.

In a similar way as in verifying sequential programs one can construct proof rules for temporal logic for reactive systems and prove the correctness of these systems with the same approach as we have seen for sequential programs using predicate logic. The disadvantages of the proof verification method, which usually requires much human involvement, are similar to those we mentioned above for checking parallel systems: they are tedious, labour-intensive, and require a high degree of user guidance. An interesting approach for reactive systems that we would like to mention, for which tool support is available, is TLA (Temporal Logic of Actions) of Lamport (1994). This approach allows one to specify requirements and system behavior in the same notation.

In these lecture notes we concentrate on another type of (formal) verification technique that is based on temporal logic, but that allows in general less involvement of the user in the verification process: **model checking**.

1.5 Model checking

The basic idea of what is known as *model checking* is to use algorithms, executed by computer tools, to verify the correctness of systems. The user inputs a description of a model of the system (the possible behavior) and a description of the requirements specification (the desirable behavior) and leaves the verification up to the machine. If an error is recognized the tool provides a counter-example showing under which circumstances the error can be generated. The counter-example consists of a scenario in which the model behaves in an undesired way. Thus the counter-example provides evidence that the model is faulty and needs to be revised.[1] This allows the user to locate the error and to repair the model specification before continuing. If no errors are found, the user can refine its model description (e.g. by taking more design decisions into account, so that the model becomes more concrete/realistic) and can restart the verification process.

The algorithms for model checking are typically based on an *exhaustive state space search* of the model of the system: for each state of the model it is checked whether it behaves "correctly", that is, whether the state satisfies the desired property. In its most simple form, this technique is known as *reachability analysis*. E.g., in order to determine whether a system can reach a state in which

[1] In some cases the formal requirement specification might be wrong, in the sense that the verification engine checks something which the user did not intend to check. Here, the user probably made a mistake in the formalization of the requirements.

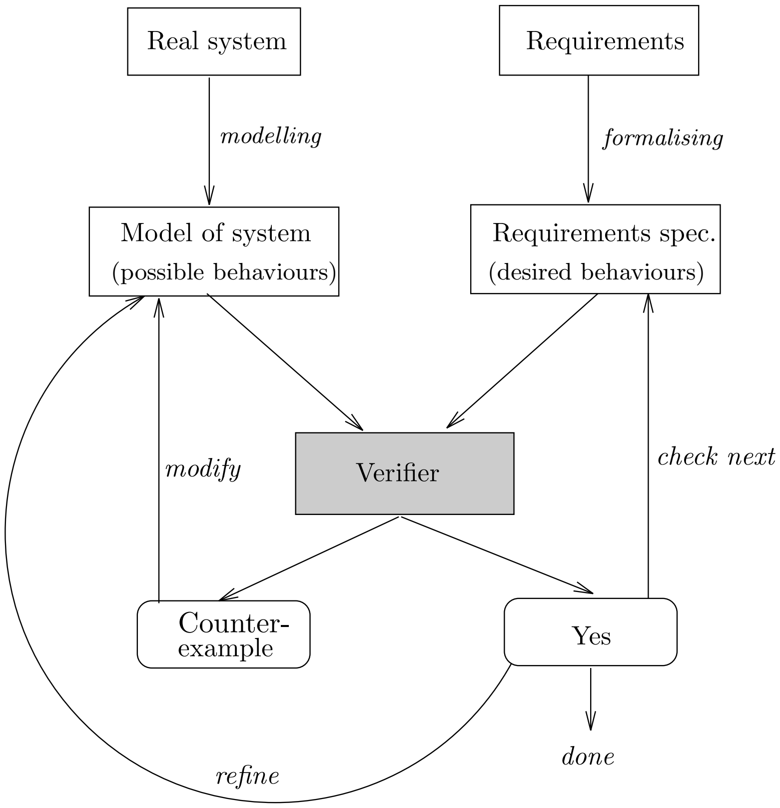


Figure 1.4: Verification methodology of model checking

no further progress is possible (a so-called deadlock), it suffices to determine all reachable states and to determine whether there exists a reachable state from which no further progress can be made. Reachability analysis is only applicable to proving freedom from deadlock and proving invariant properties, properties that hold during an entire computation. This is, for example, insufficient for communication protocols where a paramount concern is, for instance, that once a message is sent it is eventually being received. These types of "progress" properties are not covered by a usual reachability check.

There has been quite some work done on automated reachability analysis for communication protocols in the late seventies and early eighties (West, 1982). Here, protocols were modeled as a set of finite-state automata that communicate via bounded buffered, asynchronous message passing. Starting from the initial system state which is expressed in terms of the states of the interacting automata and message buffers, all system states are determined that can be reached by exchanging messages. Protocols like X.21, X.25, and IBM SNA protocols have been analyzed automatically using this techniques. Model checking can in fact be considered as a successor of these early state-space exploration techniques for protocols. It allows a wider class of properties to be examined, and handles state spaces much more efficiently than these early techniques.

Methods of model checking

There are basically two approaches in model checking that differ in the way the desired behavior, i.e. the requirement specification, is described:

1. *Logic-based or heterogeneous approach*: in this approach the desired system behavior is captured by stating a set of properties in some appropriate logic, usually some temporal or modal logic. A system — usually modeled as a finite-state automaton, where states represent the values of variables and control locations, and transitions indicate how a system can change from one state to another — is considered to be correct with respect to these requirements if it satisfies these properties for a given set of initial states.

2. *Behavior-based or homogeneous approach*: in this approach both the desired and the possible behavior are given in the same notation (e.g. an automaton), and equivalence relations (or pre-orders) are used as a correctness criterion. The equivalence relations usually capture a notion like "behaves the same as", whereas the pre-order relation represents a notion like "behaves at least as". Since there are different perspectives and intuitions about what it means for two processes to "behave the same" (or "behave at least as"), various equivalence (and pre-order) notions have been defined. One of the most well-known notions of equivalence is bisimulation. In a nutshell, two automata are bisimilar if one automaton can simulate every step of the other automaton, and vice versa. A frequently encountered notion of pre-order is (language) inclusion. An automaton A is included in automaton B, if all words accepted by A are accepted by B. A system is considered to be correct if the desired and the possible behavior are equivalent (or ordered) with respect to the equivalence (or pre-order) attribute under investigation.

Although these two techniques are conceptually different, connections between the two approaches can be established in the following way. In particular a logic induces an equivalence relation on systems as follows: two systems are equivalent if (and only if) they satisfy the same formulas. Using this concept relationships between different logics and equivalence relations have been established. For instance, it is known that two models that are bisimilar satisfy the same formulas of the logic CTL, a logic that is commonly used for model checking purposes (as we will see below). The connection between the two different approaches is now clear: if two models possess the same properties (checked using the logic approach), then they are behaviorally equivalent (as could be checked in the behavioral approach). The reverse direction is more interesting, since in general it is infeasible to check all properties of a certain logic, whereas checking for equivalences like bisimulation can be done rather efficiently (in time logarithmically in the number of states and linear in the number of transitions).

In these lecture notes we focus entirely on the logic-based approach. This approach originates from the independent work of two pairs: Quielle and Sifakis (1981) and Clarke and Emerson (1981). Since, in logical terms, one checks that the system description is a model of the temporal logic formulas, this logical approach has originally been referred to as model checking (this term has been coined by Clarke and Emerson).

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

Roughly speaking, model checking is performed as an exhaustive state space search that is guaranteed to terminate since the model is finite.

The benefits of model checking

- *General approach* with applications to hardware verification, software engineering, multi-agent systems, communication protocols, embedded systems, and so forth.

- Supports *partial* verification: a design can be verified against a partial specification, by considering only a subset of all requirements. This can result in improved efficiency, since one can restrict the validation to checking only the most relevant requirements while ignoring the checking of less important, though possibly computationally expensive requirements.

- Case studies have shown that the incorporation of model checking in the design process does not delay this process more than using simulation and testing. For several case studies the use of model checking has led to shorter development times. In addition, due to several advanced techniques, model checkers are able to deal with rather large state spaces (an example with 10^{130} states has been reported in the literature).

- Model checkers can potentially be *routinely used* by system designers with as much ease as they use, for instance, compilers; this is basically due to the fact that model checking does not require a high degree of interaction with the user.

- Rapidly increasing *interest of the industry*: jobs are offered where skills in applying model checkers are required; industry is building their own model checkers (e.g. Siemens and Lucent Technologies), have initiated their own research groups on model checking (e.g. Intel and Cadence) or is using existing tools (Fujitsu, Dutch Railways, to mention a few); finally, the first model checkers are becoming commercially available.

- *Sound and interesting mathematical foundations*: modeling, semantics, concurrency theory, logic and automata theory, data structures, graph algorithms, etc. constitute the basis of model checking.

The limitations of model checking

The main limitations of model checking are:

- Appropriate mainly to *control-intensive applications* with communications between components. It is less suited to data-intensive applications, since the treatment of data usually introduces infinite state spaces.

- The applicability of model checking is subject to *decidability issues*: for particular cases — like most classes of infinite-state systems — model checking is not effectively computable. Formal verification, though, is in principle applicable to such systems.

- Using model checking a *model* of the system is verified, rather than the real system itself. The fact that a model possesses certain properties does not guarantee that the final realization possesses the same properties. (For that purpose complementary techniques such as systematic testing are needed.) In short,

Any validation using model checking is only as good as the model of the system.

- *Only stated requirements are checked*: there is no guarantee of the completeness of desired properties.

- Finding appropriate abstraction (such as the system model and appropriate properties in temporal logic) requires some expertise.

- As any tool, model checking software might be unreliable. Since (as we will see in the sequel of these notes) standard and well-known algorithms constitute the basis for model checking, the reliability does not in principle cause severe problems. (For some cases, more advanced parts of model checking software have been proven correct using theorem provers.)

- It is impossible (in general) to check generalizations with model checking (Apt and Kozen, 1986). If, for instance, a protocol is verified to be correct for 1, 2 and 3 processes using model checking, it cannot provide an answer for the result of n processes, for arbitrary n. (Only for particular cases this is feasible.) Model checking can, however, suggest theorems for arbitrary parameters that subsequently can be verified using formal verification.

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

Model checking can provide a significant increase in the level of confidence of a system.

Since in model checking it is quite common to model the possible behavior of the system as (finite-state) automata, model checking is inherently vulnerable to the rather practical problem that the number of states may exceed the amount of computer memory available. This is in particular a problem for parallel and distributed systems that have many possible system states — the size of the state space of such systems is in the worst case proportional to the product of the size of the state spaces of their individual components. The problem that the number of states can become too large is known as the *state-space explosion problem*. Several effective methods have been developed to combat this problem; the major techniques for state-space reduction in the setting of model checking will be treated in Chapter 5 of these lecture notes.

1.6 Automated theorem proving

Automated *theorem proving* is an approach to automate the proof of mathematical theorems. This technique can effectively be used in fields where mathematical abstractions of problems are available. In case of system validation, the system specification and its realization are both considered as formulas, ϕ and ψ say, in some appropriate logic. Checking whether the implementation conforms to the specification now boils down to check whether $\psi \Rightarrow \phi$. This represents that each possible behavior of the implementation (i.e. satisfies ψ) is a possible behavior of the system specification (and thus satisfies ϕ). Note that the system specification can allow other behavior that is, however, not realized. For the proof of $\psi \Rightarrow \phi$, theorem provers can be used. Most theorem provers have algorithmic and search components. The general demand to prove theorems of a rather general type avoids to follow a solely algorithmic approach. Therefore, search components are incorporated. Different variants exist: highly automated, general-purpose theorem provers, and interactive programs with special-purpose capabilities.

Proof checking is a field that is closely related to theorem proving. A user can give a proof of a theorem to a proof checker. The checker answers whether the proof is valid. Usually the logics used in proof checking enable the proofs to be expressed more efficiently than those that are used in theorem provers. These differences in logic reflect the fact that proof checkers have an easier task than theorem provers, therefore checkers can deal with more complex proofs.

In order to reduce the search in theorem proving it is sensible to have some interaction with the user. The user may well be aware of what is the best strategy to conduct a proof. Usually such interactive systems help in giving a proof by keeping track of the things still to be done and by providing hints on how these remaining theorems can be proven. Moreover, each proof step is verified by the system. Typically many small steps have to be taken in order to arrive at a proof-checked proof and the degree of interaction with the user is 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 require much more scrunity than users are used to. Typically, human beings skip certain small parts of proofs ("trivial" or "analogous to") whereas the tool requires these steps explicitly. The verification process using theorem provers is therefore usually slow, error-prone and labour-intensive to apply. Besides this, the logic used by the tool requires a rather high degree of expertise of the user.

Logics that are used by theorem provers and proof checkers are usually variants of first-order predicate logic. In this logic we have an infinite set of variables and a set of function symbols 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 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 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 has a type (like a program variable `x` has type `int`), each function symbol has a set of argument types and a result type, and each predicate symbol has a set of argument types (but no result type). In these typed logics, quantifications are over types, since the variables are typed. Many theorem provers use *higher-order* logics: typed first-order predicate logic where variables can range over function-types or predicate-types. This enables to quantify over these types. In this way the logic becomes more expressive than first-order predicate logic.

Most theorem provers have *algorithmic* and *search* components. The algorithmic components are techniques to apply proof rules and to obtain conclusions from this. Important techniques that are used by theorem provers to support this are natural deduction (e.g. from the validity of ϕ_1 and the validity of ϕ_2 we may conclude the validity of $\phi_1 \ \wedge \ \phi_2$), 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). In contrast to traditional model checking, theorem proving can deal directly with infinite state spaces and can verify the validity of properties for arbitrary parameters values. It relies on proof principles such as structural induction to prove properties over infinite domains.

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 (often called a tactic) which tells 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 is not formalized. Strategies that are used by theorem provers are simple strategies, e.g. based on breadth-first and depth-first search strategies.

Completely automated theorem provers are not very useful in practice: the problem of theorem-proving in general is exponentially difficult, i.e. the length of a proof of a sentence of length n may be of size exponential in n. (To find such proof a time that is exponential in the length of the proof may be needed; hence in general theorem proving is double exponential in the size of the sentence to be proven.) For user-interactive theorem provers this complexity is reduced to a significant extent.

Some well-known and often applied theorem provers are Coq, Isabelle, PVS, NQTHM (Boyer-Moore), nuPRL, and HOL. A recent overview of checked proofs in the literature of sequential and distributed algorithms can be found in Groote, Monin & van de Pol (1998).

The following differences between theorem proving and model checking can be listed:

- Model checking is completely automatic and fast.
- Model checking can be applied to partial designs, and so can provide useful information about a system's correctness even if the system is not completely specified.
- Model checkers are rather user-friendly and easy to use; the use of theorem provers requires considerable expertise to guide and assist the verification process. In particular, it is difficult to familiarize oneself with the logical language (usually some powerful higher-order logic) of the theorem prover.
- Model checking is more applicable to control-intensive applications (like hardware, communication protocols, process control systems and, more general, reactive systems). Theorem proving can deal with infinite state spaces; it is therefore also suitable for data-intensive applications.
- When successful, theorem proving gives an (almost) maximal level of precision and reliability of the proof.
- Model checking can generate counter-examples, which can be used to aid in debugging.
- Using model checking a system design is checked for a fixed (and finite) set of parameters; using theorem provers a proof for arbitrary values of parameters can be given.

Model checking is not considered to be "better" than theorem proving; these techniques are to a large extent complementary and both have their benefits. The emerging effort to integrate these two techniques such that one can benefit from the advantages of both approaches is interesting. In this way, one could verify a system model for a small, finite set of parameters using model checking. From these verifications one constructs a general result and subsequently the system model can be proven to be correct, for instance using inductive arguments, for an arbitrary set of parameters with the help of a theorem prover.

1.7 Some industrial case studies of model checking

NewCoRe Project (AT & T)

- 1990–1992 Bell Laboratories (USA)
- 5ESS Switching Centre, part of Signalling Protocol no. 7
- ISDN User Part Protocol
- project size: 40-50 designers for writing the several thousands of lines of software
- 4-5 "verification engineers" started independently; they had in addition to adapt and develop tools to support the formal design process
- specification: 7,500 lines of SDL (the ITU-T Specification & Description Language)
- model checker: Spin
- 112 (!) serious design errors were found
- 145 formal requirements stated in temporal logic
- 10,000 verification runs (100/week)
- 55% (!) of the original design requirements were logically
- experiment to really go through two independent system design cycles has been given up: after 6 weeks (!) a series of logical inconsistencies have been uncovered in the requirement by the verification engineers, and the conventional design team could not be kept unaware
- unknown how many errors would have resulted, when the teams would have continued working independently inconsistent.

IEEE Futurebus+ Standard

- bus architecture for distributed systems
- bus segment: set of processors connected via a bus which share a common memory M.
- each processor has a local cache to store some data
- cache consistency protocol must ensure consistency of data, e.g.,
  - if 2 caches contain a copy of an item, the copies must have the same value
  - if M contains a data item which is 'non-modified' then each cache that has a copy of this item contains the same as M
- bus segments can be connected via bridges, yielding a hierarchical bus system
- model checker: SMV
- complexity: 2,300 lines of SMV-code
- configuration: 3 bus segments and 8 processors
- state space: 10^{30} states (after abstraction from several details)
- several non-trivial, fatal errors were found
- result: substantial revision of original standard proposal

1.8 Selected references

Use of (software) validation techniques in practice (in German):

- P. Liggesmeyer, M. Rothfelder, M. Rettelbach and T. Ackermann. Qualitätssicherung Software-basierter technischer Systeme – Problembereiche und Lösungsansätze. *Informatik Spektrum*, **21**: 249–258, 1998.

Testing based on formal methods:

- ISO/ITU-T. *Formal Methods in Conformance Testing.* Draft International Standard, 1996.

Formal verification:

- K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, 1997.
- C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, **12**: 576–580, 583, 1969.
- S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, **6**: 319–340, 1976.

Introduction of temporal logic in computer science:

- A. Pnueli. The temporal logic of programs. *Proceedings 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

Formal verification using temporal logic:

- L. Lamport. The temporal logic of actions. *ACM Transaction on Programming Languages and Systems*, **16**(3): 872–923, 1994.

Introduction and overview of model checking:

- E.M. Clarke and R.P. Kurshan. Computer-aided verification. *IEEE Spectrum*, **33**(6): 61–67, 1996.
- E.M. Clarke, J.M. Wing et. al. Formal methods: state of the art and future directions. *ACM Computing Surveys*, **28**(4): 626–643, 1996.

- P. Wolper. An introduction to model checking. *Position statement for panel discussion at the Software Quality Workshop*, unpublished note, 1995.

Origins of model checking:

- J. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. *Proceedings 5th International Symposium on Programming*, LNCS 137, pages 337–351, 1982.
- E.M. Clarke and E.A. Emerson. Synthesis of synchronisation skeletons for branching time logic. *Logic of Programs*, LNCS 131, pages 52–71, 1981.

Protocol reachability techniques:

- C.H. West. Applications and limitations of automated protocol validation. *Proceedings 2nd Symposium on Protocol Specification, Testing and Verification*, pages 361–371, 1982.

Limitations of model checking:

- K.R. Apt and D.C. Kozen. Limits for the automatic verification of finite-state concurrent systems. *Information Processing Letters*, **22**: 307–309, 1986.

Checking proofs of distributed systems:

- J.F. Groote, F. Monin and J. van de Pol. Checking verifications of protocols and distributed systems by computer. *Concur'98: Concurrency Theory*, LNCS 1432, pages 629–655, 1998.

NewCoRe project:

- G.J. HOLZMANN. The theory and practice of a formal method: NewCoRe. *Proceedings 13th IFIP World Congress*, pages 35–44, 1994.

IEEE Futurebus project:

- E.M. CLARKE, O. GRUMBERG, H. HIRAISHI, S. JHA, D.E. LONG, K.L. MCMILLAN AND L.A. NESS. Verification of the Futurebus+ cache coherence protocol. *Proceedings 11th Int. Symp. on Computer Hardware Description Languages and their Applications*, 1993.

LNCS is an abbreviation of Lecture Notes in Computer Science as published by Springer Verlag.

Chapter 2

Model Checking Linear 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 some actions, such as fetching a file or producing coffee. Thus a reactive system reacts to a stimulus from the environment by a reaction. After an interaction and accompanying reaction, a reactive system is — once more — able to interact. The continuous character of interaction in reactive systems differs from the traditional view on sequential systems. These sequential systems are considered as functions from inputs to outputs. After receiving input, the sequential system generates output(s) and terminates eventually after a finite number of computation steps. Due to the "transformation" of inputs into outputs these type of systems are sometimes also referred to as transformational systems. Properties of such transformational systems typically relate outputs generated to inputs provided in terms of pre- and postconditions as discussed previously.

Typical reactive systems are non-terminating. Due to their non-terminating behavior, properties of reactive systems we usually refer to the relative order of events in the system: i.e. the behavior of the system while over time. Lamport (1977) argued that the requirements that designers wish to impose on reactive systems fall basically into two categories:

- *Safety* properties state that "something bad never happens". A system satisfies such a property if it does not engage in the proscribed activity. For instance, a typical safety property for a coffee-machine is that the machine will provide never tea if the user requests coffee.

- *Liveness* properties state that "something good will eventually happen"; to satisfy such a property, the system must engage in some desired activity. An example liveness property for the coffee-machine is that it will provide coffee eventually after a sufficient payment by the user. Stated differently, the user will always be provided coffee at some time after inserting the payment.

Although informal, this classification has proven to be rather useful: the two classes of properties are (almost) disjoint, and most properties can be described as a combination of safety and liveness properties. (Later several authors refined and extended this classification with several other types of properties.)

Logics have been defined in order to describe precisely these type of properties. The most widely studied are *temporal logics*, which were introduced for specification and verification purposes in computer science by Pnueli (1977). (The origins of linear temporal logic go back to the field of philosophy where A. Prior invented it in the 1960s.) Temporal logics support the formulation of properties of system behavior over time. Different interpretations of temporal logics exist depending on how one considers the system to change with time.

1. *Linear* temporal logic allows the statement of properties of execution sequences of a system.

2. *Branching* temporal logic allow the user to write formulas which include some sensitivity to the choices available to a system during its execution. It allows the statement of properties of about possible execution sequences that start in a state.

In this chapter we deal with linear temporal logic; branching temporal logic will be treated in Chapter 3. Chapter 3 also contains a more extensive discussion about the differences between these two interpretations of temporal logic.

2.1 Syntax of PLTL

The syntax of propositional linear temporal logic is defined as follows. The starting point is a set of *atomic propositions*, i.e. propositions that cannot be further refined. Atomic propositions are the most basic statements that can be made. The set of atomic propositions is denoted by AP, and typical elements of AP are p, q and r. Examples of atomic propositions are x is greater than 0, or x equals one, given some variable x. Other examples are "it is raining" or "there are currently no customers in the shop". In principle, atomic propositions are defined over a set of variables $x, y, \ldots$, constants $0, 1, 2, \ldots$, functions $max, gcd, \ldots$ and predicates $x = 2, x\ mod\ 2 = 0, \ldots$, allowing e.g. statements such as $max(x, y) \leqslant 3$ or $x = y$. We will not dwell upon the precise definition of AP here and simple postulate its existence. AP is ranged over by p, q and r.

Notice that the choice of the set of atomic propositions AP is an important one; it fixes the most basic propositions that can be stated about the system under investigation. Fixing the set AP can therefore already be regarded as a *first step of abstraction*. If one, for instance, decides not to allow some system 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.

The following definition determines the set of basic formulas that can be stated in propositional linear temporal logic (PLTL, for short).

Definition 1. (**Syntax of propositional linear temporal logic**)
Let AP be a set of atomic propositions. Then:

1. For all $p \in AP$, 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.

The set of formulas constructed according to these rules is denoted by PLTL.

Note that the set of formulas obtained by the first three items gives us the set of formulas of propositional logic. Propositional logic is thus a proper subset of PLTL. The only temporal operators are X (pronounced "neXt") and U (pronounced "Until").

Alternatively, the syntax of PLTL can be given in Backus-Naur Form (BNF, for short) as follows: for $p \in AP$ the set of PLTL-formulas is defined by

$$\boxed{\phi ::= p \;\Big|\; \neg\,\phi \;\Big|\; \phi \,\vee\, \phi \;\Big|\; \mathsf{X}\,\phi \;\Big|\; \phi\,\mathsf{U}\,\phi.}$$

We will use this succinct notation for syntax definitions in the rest of these lecture notes.

The usual boolean operators $\wedge$ (conjunction), $\Rightarrow$ (implication) and $\Leftrightarrow$ (equivalence) are defined as

$$\begin{aligned} \phi \,\wedge\, \psi &\equiv \neg\,(\neg\,\phi \,\vee\, \neg\,\psi) \\ \phi \,\Rightarrow\, \psi &\equiv \neg\,\phi \vee \psi \\ \phi \Leftrightarrow \psi &\equiv (\phi \,\Rightarrow\, \psi) \,\wedge\, (\psi \,\Rightarrow\, \phi). \end{aligned}$$

The formula true equals $\phi \,\vee\, \neg\,\phi$ and false equals $\neg$ true. The temporal operators G (pronounced "always" or "Globally") and F (pronounced "eventually" or "Future") are defined by

$$\begin{aligned} \mathsf{F}\,\phi &\equiv \text{true}\,\mathsf{U}\,\phi \\ \mathsf{G}\,\phi &\equiv \neg\,\mathsf{F}\,\neg\,\phi. \end{aligned}$$

Since true is valid in all states, $\mathsf{F}\,\phi$ indeed denotes that ϕ holds at some point in the future. Suppose there is no point in the future for which $\neg\,\phi$ holds. Then, the counterpart of ϕ holds at any point. This explains the definition of $\mathsf{G}\,\phi$. In the literature F is sometimes denoted as $\Diamond$ and G as $\Box$. In these lecture notes we will use the traditional notations F and G.

To summarize, a formula without a temporal operator ($\mathsf{X}, \mathsf{F}, \mathsf{G}, \mathsf{U}$) at the "top level" refers to the current state, 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.

In order to diminish the use of brackets, a precedence order on the operators is imposed as follows. As usual the unary operators bind stronger than the binary ones. For instance, we write $\neg\,\phi\,\mathsf{U}\,\mathsf{F}\,\psi$ instead of $(\neg\,\phi)\,\mathsf{U}\,(\mathsf{F}\,\psi)$. $\neg$ and X bind equally strong and bind stronger than F and G which also bind equally strong. The temporal operator U takes precedence over $\wedge$, $\vee$, and $\Rightarrow$. $\Rightarrow$ binds weaker than $\wedge$ and $\vee$, and $\wedge$ and $\vee$ bind equally strong. For instance,

$$((\neg\,\phi) \Rightarrow \psi)\,\mathsf{U}\,((\mathsf{X}\,\phi) \;\wedge\; (\mathsf{F}\,\psi))$$

is denoted by

$$(\neg\,\phi \Rightarrow \psi)\,\mathsf{U}\,(\mathsf{X}\,\phi \;\wedge\; \mathsf{F}\,\psi).$$

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

2.2 Semantics of PLTL

The above definition provides us a recipe for the construction of PLTL-formulas, but it does not give an interpretation to these operators. Formally, PLTL is interpreted on sequences of states. Intuitively, $\mathsf{X}\,\phi$ means that ϕ is valid in the next state, $\mathsf{F}\,\phi$ means that ϕ is valid eventually, that is, it is valid now, or at some point in the future. But what do we mean by "state", "next state" and "some point in the future"? In order to define these notions unambiguously, a formal interpretation — usually called semantics — is given. The formal meaning of temporal logic formulas is defined in terms of a *model*.

Definition 2. (**Model for** PLTL)
A PLTL-model is a triple $\mathcal{M} = (S, R, \mathit{Label})$ where

- S is a non-empty denumerable set of states,
- $R : S \longrightarrow S$, assigning to $s \in S$ its unique successor state $R(s)$,
- $\mathit{Label} : S \longrightarrow 2^{AP}$, assigning to each state $s \in S$ the atomic propositions $\mathit{Label}(s)$ that are valid in s.

For the state $s \in S$, the state $R(s)$ is the unique next state of s. The important characteristic of the function R is that it acts as a generator for infinite sequences of states such as $s, R(s), R(R(s)), R(R(R(s))), \ldots$. For the semantics of PLTL these sequences of states are the cornerstones. One can equally well define a PLTL-model as a structure $(S, \sigma, \mathit{Label})$ where σ is an infinite sequence of states and S and Label are defined as above. In this alternative setting a finite sequence $s_0\, s_1\, \ldots\, s_k$ is identified with the infinite sequence $s_0\, s_1\, \ldots\, s_k\, s_k\, s_k\, \ldots$. This alternative approach is frequently encountered in the literature.

The function Label indicates which atomic propositions are valid for any state in $\mathcal{M}$. If for state s we have $\mathit{Label}(s) = \varnothing$ it means that no proposition is valid in s. The state s for which the proposition p is valid, i.e. $p \in \mathit{Label}(s)$, is sometimes referred to as a p-state.

Example 2. Let $AP = \{\, x = 0, x = 1, x \neq 0 \,\}$ *be a set of atomic propositions,* $S = \{\, s_0, \ldots, s_3 \,\}$ *a set of states,* $R(s_i) = s_{i+1}$ *for* $0 \leqslant i < 3$ *and* $R(s_3) = s_3$ *the successor function, and* $\mathit{Label}(s_0) = \{\, x \neq 0 \,\}$, $\mathit{Label}(s_1) = \mathit{Label}(s_2) = \{\, x = 0 \,\}$, $\mathit{Label}(s_3) = \{\, x = 1, x \neq 0 \,\}$ *the function which assigns atomic propositions to states. In the model* $\mathcal{M} = (S, R, \mathit{Label})$ *the atomic proposition "*$x = 0$*" is valid in states* s_1 *and* s_2*, "*$x \neq 0$*" is valid in* s_0 *and* s_3*, and "*$x = 1$*" is valid in state* s_3 *only.* *(End of example.)*

Note that an atomic proposition can either hold or not hold in any state of $\mathcal{M}$. No further interpretation is given. For instance, if "$x = 1$" is valid in state s, it does not mean that "$x \neq 0$" is also valid in that state. Technically this results from the fact that no constraints are put on the labelling *Label* of states with atomic propositions.

The meaning of formulas in logic is defined by means of a satisfaction relation (denoted by $\models$) between a model $\mathcal{M}$, one of its states s, and a formula ϕ. $(\mathcal{M}, s, \phi) \in \models$ is denoted by the following infix notation: $\mathcal{M}, s \models \phi$. The concept is that $\mathcal{M}, s \models \phi$ if and only if ϕ is valid in the state s of the model $\mathcal{M}$. When the model $\mathcal{M}$ is clear from the context we often omit the model and simply write $s \models \phi$ rather than $\mathcal{M}, s \models \phi$.

Definition 3. (**Semantics of** PLTL)
Let $p \in AP$ be an atomic proposition, $\mathcal{M} = (S, R, \mathit{Label})$ a PLTL-model, $s \in S$, and ϕ, ψ PLTL-formulas. The satisfaction relation $\models$ is defined by:

$$
\begin{array}{ll}
s \models p & \text{iff } p \in \mathit{Label}(s) \\
s \models \neg\,\phi & \text{iff } \neg\,(s \models \phi) \\
s \models \phi \,\vee\, \psi & \text{iff } (s \models \phi) \,\vee\, (s \models \psi) \\
s \models \mathsf{X}\,\phi & \text{iff } R(s) \models \phi \\
s \models \phi\,\mathsf{U}\,\psi & \text{iff } \exists\, j \geqslant 0.\, R^j(s) \models \psi \ \wedge\ (\forall\, 0 \leqslant k < j.\, R^k(s) \models \phi).
\end{array}
$$

Here $R^0(s) = s$, and $R^{n+1}(s) = R^n(R(s))$ for any $n \geqslant 0$.

(Although the logical operators on the right-hand side of a defining equation are denoted in the same way as the logical operators in the logic, they represent operators at a meta-level.) If $R(s) = s'$, the state s' is called a direct successor of s. If $R^n(s) = s'$ for $n \geqslant 1$, state s' is called a successor of s. If $\mathcal{M}, s \models \phi$ we say that model $\mathcal{M}$ satisfies ϕ in state s. Stated otherwise, the formula ϕ holds in state s of model $\mathcal{M}$.

The formal interpretation of the other connectives true, false, $\wedge$, $\Rightarrow$, G, and F can now be obtained from this definition by straightforward calculation. It is not difficult to see that true is valid in all states: take true $\equiv p \,\vee\, \neg\, p$, and the validity of true in state s reduces to $p \in \mathit{Label}(s) \vee p \notin \mathit{Label}(s)$ which is a valid statement for all states s. So $\mathcal{M}, s \models \text{true}$ for all $s \in S$. As an example we derive for the semantics of $\mathsf{F}\,\phi$:

$$
\begin{array}{ll}
 & s \models \mathsf{F}\,\phi \\
\Leftrightarrow & \{ \text{ definition of } \mathsf{F} \; \} \\
 & s \models \text{true}\,\mathsf{U}\,\phi \\
\Leftrightarrow & \{ \text{ semantics of } \; \mathsf{U} \; \} \\
 & \exists\, j \geqslant 0.\, R^j(s) \models \phi \;\wedge\; (\forall\, 0 \leqslant k < j.\, R^k(s) \models \text{true}) \\
\Leftrightarrow & \{ \text{ calculus } \} \\
 & \exists\, j \geqslant 0.\, R^j(s) \models \phi.
\end{array}
$$

Therefore, $\mathsf{F}\,\phi$ is valid in s if and only if there is some (not necessarily direct) successor state of s, or s itself where ϕ is valid.

Using this result for F, for the semantics of $\mathsf{G}\,\phi$ we now become:

$$
\begin{array}{ll}
 & s \models \mathsf{G}\,\phi \\
\Leftrightarrow & \{ \text{ definition of } \mathsf{G} \; \} \\
 & s \models \neg\,\mathsf{F}\,\neg\,\phi \\
\Leftrightarrow & \{ \text{ using result of previous derivation } \} \\
 & \neg\,[\exists\, j \geqslant 0.\, R^j(s) \models \neg\,\phi] \\
\Leftrightarrow & \{ \text{ semantics of } \; \neg\,\phi \; \} \\
 & \neg\,[\exists\, j \geqslant 0.\, \neg\,(R^j(s) \models \phi)] \\
\Leftrightarrow & \{ \text{ predicate calculus } \} \\
 & \forall\, j \geqslant 0.\, R^j(s) \models \phi.
\end{array}
$$

Therefore, the formula $\mathsf{G}\,\phi$ is valid in s if and only if for all successor states of s, including s itself, the formula ϕ is valid. This explains the expression "always" ϕ.

Example 3. *If the formula $\phi\,\mathsf{U}\,\psi$ is valid in s, then $\mathsf{F}\,\psi$ is also valid in s. Thus when stating $\phi\,\mathsf{U}\,\psi$ it is implicitly assumed that there exists some future state for which ψ holds. 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 sequence. The unless operator W *is defined by*

$$\phi \,\mathsf{W}\, \psi \;\equiv\; \mathsf{G}\,\phi \;\vee\; (\phi \,\mathsf{U}\, \psi).$$

(End of example.)

Example 4. *Let $\mathcal{M}$ be given as a sequence of states depicted as the first row in Figure 2.1. States are depicted as circles and the function R is denoted by arrows, i.e. there is an arrow from s to s' iff $R(s) = s'$. Since R is a (total) function, each state has precisely one outgoing arrow. The labelling* Label *is indicated below the states.*

In the lower rows the validity of the three formulas is shown, $\mathsf{F}\,p$, $\mathsf{G}\,p$, *and* $q \,\mathsf{U}\, p$, *for all states in the given model $\mathcal{M}$. A state is colored black if the formula is valid in that state, and colored white otherwise. The formula* $\mathsf{F}\,p$ *is valid in all but the last state. No p-state can be reached from the last state.* $\mathsf{G}\,p$ *is invalid in all states, since there is no state that only has p-states as successors. From all q-states the (unique) p-state can be reached in zero or more steps. Therefore, for all these states the formula* $q \,\mathsf{U}\, p$ *holds.* *(End of example.)*

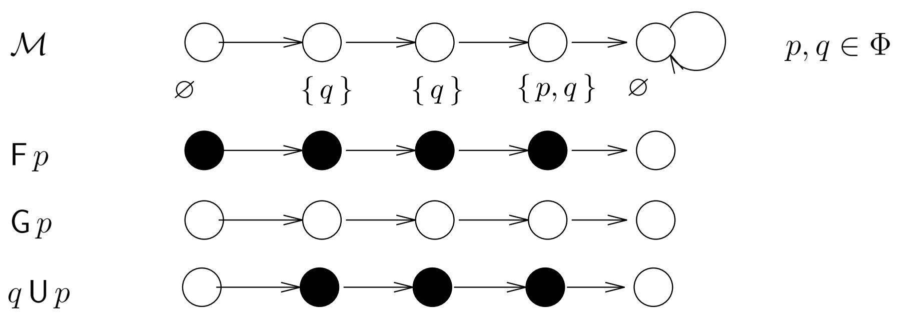


Figure 2.1: Example of interpretation of PLTL-formulas (I)

Example 5. *Let $\mathcal{M}$ be given as the sequence of states depicted in Figure 2.2 where p through t are atomic propositions. Consider the formula* $\mathsf{X}\,[r \;\Rightarrow\; (q \,\mathsf{U}\, s)]$ *which equals* $\mathsf{X}\,[\neg\, r \;\vee\; (q \,\mathsf{U}\, s)]$. *For all states whose direct successor does not satisfy r, the formula is valid due to the first disjunct. This applies to the second* *state. In the third state the formula is not valid since r is valid in its successor state but q and s are not. Finally, in the first and the fourth state the formula is valid, since in their direct successor state q is valid and in the state thereafter s is, so $q \,\mathsf{U}\, s$ is valid in their direct successor states. The validity of the formulas $\mathsf{G}\, p$, $\mathsf{F}\, t$, $\mathsf{G}\,\mathsf{F}\, s$ and the formula $\mathsf{X}\,[r \;\Rightarrow\; (q \,\mathsf{U}\, s)]$ is indicated in the rows depicted below $\mathcal{M}$.* *(End of example.)*

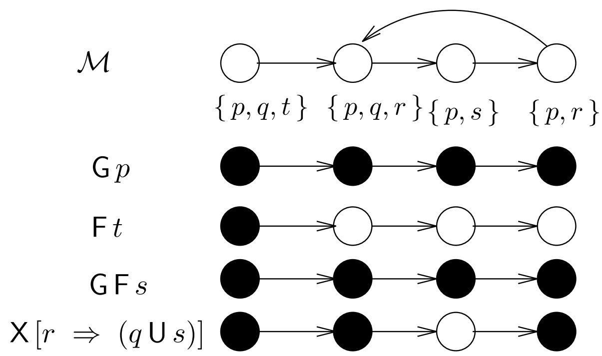


Figure 2.2: Example of interpretation of PLTL-formulas (II)

We give below some frequently used propositions in PLTL and provide their verbal interpretation such that $\mathcal{M}, s \models \phi$, i.e. ϕ holds in state s of $\mathcal{M}$.

1. $\phi \;\Rightarrow\; \mathsf{F}\, \psi$: if initially (that is, in s) ϕ, then eventually ψ.
2. $\mathsf{G}\,[\phi \;\Rightarrow\; \mathsf{F}\, \psi]$: if ϕ then eventually ψ (for all successor states of s and the initial state s).
3. $\mathsf{G}\,\mathsf{F}\, \phi$: ϕ holds infinitely often.
4. $\mathsf{F}\,\mathsf{G}\, \phi$: eventually permanently ϕ.
5. $\mathsf{G}\,[\phi \;\Rightarrow\; \mathsf{X}\, \phi]$: if ϕ is valid in some state, then it is also valid in its successor state.
6. $\mathsf{G}\,[\phi \;\Rightarrow\; \mathsf{G}\, \phi]$: once ϕ, always ϕ.

Model checking, satisfiability and validity

In the first chapter 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 characterisation. The model checking problem can formally be defined as:

> *The model checking problem is: given a (finite) model $\mathcal{M}$, a state s and a property ϕ, do we have $\mathcal{M}, s \models \phi$?*

The model checking problem should not be confused with the more traditional satisfiability problem in logic.

The *satisfiability* problem can be phrased as follows: given a property ϕ, does there exist a model $\mathcal{M}$ and state s, such that $\mathcal{M}, s \models \phi$? While for model checking the model $\mathcal{M}$ (and state s) is given, this is not the case for the satisfiability problem. For PLTL the satisfiability problem is decidable. Since satisfiability is decidable, the (simpler) model checking problem for PLTL is also decidable. Satisfiability is relevant to system validation using formal verification (cf. Chapter 1) in the following sense. As an example consider a system specification and its implementation, both specifications being formalised as PLTL-formulas ϕ and ψ. Checking whether the implementation conforms to the specification now reduces to checking whether $\psi \Rightarrow \phi$. By determining the satisfiability of this formula one is able to check whether there exists a model for which this desired relation does exist. If the formula is not satisfiable, that is, no such model does exist, it means that the relation between the implementation and specification cannot be realised in any model. Thus, the implementation is not a correct implementation of the specification.

A related problem that occurs frequently in the literature is the *validity* problem: given a property ϕ, do we have for all models $\mathcal{M}$ and all states s in these models, $\mathcal{M}, s \models \phi$? The difference to the satisfiability problem is that to solve the validity problem one has to check whether $\mathcal{M}, s \models \phi$ for *all* existing models $\mathcal{M}$ and states s, rather than to determine the existence of one (or more) such $\mathcal{M}$ and s. Logically speaking, the validity problem for ϕ equals the negation of the satisfiability problem for $\neg\, \phi$.

(Aside: 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* states of *all* models — since PLTL-models are allowed to have an infinite state space. However, due to the so-called finite-model property for PLTL, it suffices to consider only all models with a finite number of states to check validity. The finite-model property states that if a PLTL-formula is satisfiable, then it is satisfiable in a model with a finite set of states.)

2.3 Axiomatization

The validity of a PLTL-formula can of course be derived using the semantics as given before. This is usually a rather cumbersome task since we have to reason about the formal semantics which is defined in terms of the model $\mathcal{M}$. Let us, for instance, try to deduce that

$$\phi \,\mathsf{U}\, \psi \;\Leftrightarrow\; \psi \;\vee\; [\phi \;\wedge\; \mathsf{X}\,(\phi \,\mathsf{U}\, \psi)].$$

Intuitively this equation is valid: if in the current state 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 current state and in the next state $\phi \,\mathsf{U}\, \psi$ holds, then $\phi \,\mathsf{U}\, \psi$ holds. We obtain the following derivation:

$$
\begin{array}{ll}
 & s \models \psi \;\vee\; [\phi \;\wedge\; \mathsf{X}\,(\phi \,\mathsf{U}\, \psi)] \\
\Leftrightarrow & \{ \text{ semantics of } \wedge \text{ and } \vee \; \} \\
 & (s \models \psi) \;\vee\; (s \models \phi \;\wedge\; s \models \mathsf{X}\,(\phi \,\mathsf{U}\, \psi)) \\
\Leftrightarrow & \{ \text{ semantics of } \mathsf{X} \; \} \\
 & (s \models \psi) \;\vee\; (s \models \phi \;\wedge\; R(s) \models \phi \,\mathsf{U}\, \psi) \\
\Leftrightarrow & \{ \text{ semantics of } \mathsf{U} \; \} \\
 & \quad (s \models \psi) \\
 & \vee\; (s \models \phi \;\wedge\; [\exists\, j \geqslant 0.\, R^j(R(s)) \models \psi \;\wedge\; \forall\, 0 \leqslant k < j.\, R^k(R(s)) \models \phi]) \\
\Leftrightarrow & \{ \text{ calculus using } R^{n+1}(s) = R^n(R(s)) \; \} \\
 & \quad (s \models \psi)
\end{array}
$$

$$\vee\ [\exists\, j \geqslant 0.\, R^{j+1}(s) \models \psi\ \wedge\ \forall\, 0 \leqslant k < j.\, (R^{k+1}(s) \models \phi\ \wedge\ s \models \phi)]$$

$$\Leftrightarrow\ \{ \text{ calculus using } R^0(s) = s\ \}$$

$$(s \models \psi)\ \vee\ [\exists\, j \geqslant 0.\, R^{j+1}(s) \models \psi\ \wedge\ \forall\, 0 \leqslant k < j{+}1.\, R^k(s) \models \phi]$$

$$\Leftrightarrow\ \{ \text{ calculus using } R^0(s) = s\ \}$$

$$[\exists\, j = 0.\, R^0(s) \models \psi\ \wedge\ \forall\, 0 \leqslant k < j.\, R^k(s) \models \phi]$$

$$\vee\ [\exists\, j \geqslant 0.\, R^{j+1}(s) \models \psi\ \wedge\ \forall\, 0 \leqslant k < j{+}1.\, R^k(s) \models \phi]$$

$$\Leftrightarrow\ \{ \text{ predicate calculus } \}$$

$$[\exists\, j \geqslant 0.\, R^j(s) \models \psi\ \wedge\ \forall\, 0 \leqslant k < j.\, R^k(s) \models \phi]$$

$$\Leftrightarrow\ \{ \text{ semantics of } \mathsf{U}\ \}$$

$$s \models \phi \,\mathsf{U}\, \psi.$$

As we can see 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 proof rules that allow the rewriting of PLTL-formulas into semantically equivalent PLTL-formulas at a syntactical level. If, for instance, $\phi \equiv \psi$ is a rule (axiom), then $s \models \phi$ if and only if $s \models \psi$ for all $\mathcal{M}$ and state s. Semantically equivalent means that for all states in all models these rules are valid. The set of proof rules obtained in this way is called an *axiomatization*.

It is not the aim of this section to deal with all possible proof rules for PLTL but rather to give the reader some basic rules that are convenient. The rules presented below are grouped and each group has been given a name, for reference purposes. Using the idempotency and the absorption rules 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 proof rules can be proven using the semantic interpretation as we have seen for the first expansion law above. The difference is that we only have to perform these tedious proofs once; thereafter these rules can be universally applied. Note that the validity of the expansion rules for F and G follows directly from the validity of the expansion rule for U, using the definition of F and G.

An equational rule is said to be *sound* if it is valid. Formally, the axiom $\phi \equiv \psi$ is called sound if and only if, for any PLTL-model $\mathcal{M}$ and state s in $\mathcal{M}$:

$$\mathcal{M}, s \models \phi \text{ if and only if } \mathcal{M}, s \models \psi$$

Duality rules:	$\neg\,\mathsf{G}\,\phi$	$\equiv$	$\mathsf{F}\,\neg\,\phi$
	$\neg\,\mathsf{F}\,\phi$	$\equiv$	$\mathsf{G}\,\neg\,\phi$
	$\neg\,\mathsf{X}\,\phi$	$\equiv$	$\mathsf{X}\,\neg\,\phi$
Idempotency rules:	$\mathsf{G}\,\mathsf{G}\,\phi$	$\equiv$	$\mathsf{G}\,\phi$
	$\mathsf{F}\,\mathsf{F}\,\phi$	$\equiv$	$\mathsf{F}\,\phi$
	$\phi\,\mathsf{U}\,(\phi\,\mathsf{U}\,\psi)$	$\equiv$	$\phi\,\mathsf{U}\,\psi$
	$(\phi\,\mathsf{U}\,\psi)\,\mathsf{U}\,\psi$	$\equiv$	$\phi\,\mathsf{U}\,\psi$
Absorption rules:	$\mathsf{F}\,\mathsf{G}\,\mathsf{F}\,\phi$	$\equiv$	$\mathsf{G}\,\mathsf{F}\,\phi$
	$\mathsf{G}\,\mathsf{F}\,\mathsf{G}\,\phi$	$\equiv$	$\mathsf{F}\,\mathsf{G}\,\phi$
Commutation rule:	$\mathsf{X}\,(\phi\,\mathsf{U}\,\psi)$	$\equiv$	$(\mathsf{X}\,\phi)\,\mathsf{U}\,(\mathsf{X}\,\psi)$
Expansion rules:	$\phi\,\mathsf{U}\,\psi$	$\equiv$	$\psi\ \vee\ [\phi\ \wedge\ \mathsf{X}\,(\phi\,\mathsf{U}\,\psi)]$
	$\mathsf{F}\,\phi$	$\equiv$	$\phi\ \vee\ \mathsf{X}\,\mathsf{F}\,\phi$
	$\mathsf{G}\,\phi$	$\equiv$	$\phi\ \wedge\ \mathsf{X}\,\mathsf{G}\,\phi$

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 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, but falls outside the scope of these lecture notes.

2.4 Extensions of PLTL (optional)

Strict and non-strict interpretation. $\mathsf{G}\,\phi$ means that ϕ holds at all states including the current state. It is called a *non-strict* interpretation since it also refers to the current state. Opposed to this a *strict* interpretation does not refer to the current state. The strict interpretation of G, denoted $\tilde{\mathsf{G}}$, is defined by $\tilde{\mathsf{G}}\,\phi \equiv \mathsf{X}\,\mathsf{G}\,\phi$. That is, $\tilde{\mathsf{G}}\,\phi$ means that ϕ 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. In the opposite direction we have:

$$\begin{aligned}
\mathsf{G}\,\phi &\equiv \phi \wedge \tilde{\mathsf{G}}\,\phi \\
\mathsf{F}\,\phi &\equiv \phi \wedge \tilde{\mathsf{F}}\,\phi \\
\phi\,\mathsf{U}\,\psi &\equiv \psi \vee [\phi \wedge (\phi\,\tilde{\mathsf{U}}\,\psi)].
\end{aligned}$$

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 rule 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 rule for U. Although sometimes the strict interpretation is used, it is more common to use the non-strict interpretation as is done in these lecture notes.

Past operators. All operators refer to the future including the current state. Therefore, the operators are known as future operators. Sometimes PLTL is extended with *past* operators. This can be useful for a particular application, since sometimes properties are more easily expressed in terms of the past than in terms of the future. For instance, $\underline{\mathsf{G}}\,\phi$ ("always in the past") means — in the non-strict interpretation — that ϕ is valid now and in any state in the past. $\underline{\mathsf{F}}\,\phi$ ("sometime in the past") means that either ϕ is valid in the current state or in some state in the past and $\underline{\mathsf{X}}\,\phi$ means that ϕ 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 make it easier to write a specification; the expressive power of PLTL is not increased by the addition of past operators (Lichtenstein, Pnueli & Zuck, 1985)[1].

[1]If as an underlying semantic notion a discrete model is taken, as in this chapter.

2.5 Specifying properties in PLTL

In order to give the reader some idea of how to formulate informal properties in linear temporal logic, we treat two examples. In the first example we try to formalize properties of a simple communication system, and in the second example we treat a formal requirement specification of a leader election protocol in a distributed system where processes can start at arbitrary moments.

A communication channel

Consider a unidirectional channel between two communicating processes: a sender (S) and a receiver (R). S is equipped with an output buffer (S.out) and R with an input buffer (R.in). Both buffers have an infinite capacity. If 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 a unidirectional channel. R receives messages by deleting messages from its input buffer R.in. Assume that all messages are uniquely identified, and assume the set of atomic propositions $AP = \{ m \in S.out, m \in R.in \}$ where m denotes a message. We assume implicitly that all properties are stated for all messages m (i.e. universal quantification over m is assumed). This is for convenience and does not affect the decidability of the satisfiability problem if we assume that the number of messages is finite. 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 infinitely long.

We formalize the following informal requirements on the channel in PLTL:

- "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". If the channel does not lose messages it means that messages that are in S.out will eventually be delivered in R.in:

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

  (Notice that if we do not assume uniqueness of messages then this property does not suffice, since if e.g. two copies of m are transmitted and only the last copy is eventually received this would satisfy this property. It has been shown by several authors that uniqueness of messages is a prerequisite for the specification of requirements for message-passing systems in temporal logic.) Given the validity of the previous property, 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':

  $$\begin{array}{l} \mathsf{G}\,[m \in S.out \;\wedge\; \neg\, m' \in S.out \;\wedge\; \mathsf{F}\,(m' \in S.out) \\ \Rightarrow\; \mathsf{F}\,(m \in R.in \;\wedge\; \neg\, m' \in R.in \;\wedge\; \mathsf{F}\,(m' \in R.in))] \end{array}$$

  Notice 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 is replaced by X.)

- "The channel does not spontaneously generate messages". This means that for any m in R.in, it must have been previously sent by S. Using the past operator $\underline{\mathsf{F}}$ this can be formalized conveniently by:

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

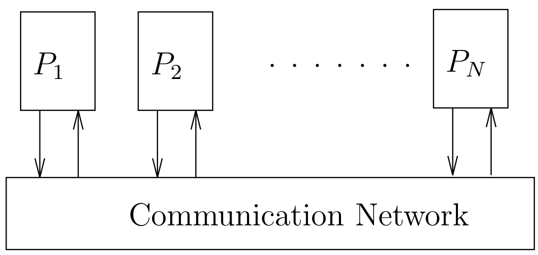
  In the absence of past operators, a it is possible to use the U-operator:

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

Dynamic leader election

In current distributed systems several functions (or services) are offered by some dedicated process(es) in the system. One might think of 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 doing this. However, for consistency reasons at any time only one process is allowed to actually offer the function. Therefore, one process — called the "leader" — must be elected to support that function. Sometimes it suffices to elect an arbitrary process, but for other functions it is important to elect the process with the best capabilities for performing that function. Here we abstract from specific capabilities and use ranking on basis of process identities. The idea is thus that a process with a higher identity has better capabilities.

Assume we have a *finite* number 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 (so, 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; 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.

Assume the set of atomic propositions

$$AP = \{ \, \mathit{leader}_i, \mathit{active}_i, i < j \mid 0 < i, j \leqslant N \, \}$$

where leader_i means that process i is a leader, active_i means that process i is active, and $i < j$ that the identity of i is smaller than the identity of j (in the total ordering on identities). We will use i, j as process identities. N is the number of processes. Assume that a process that is inactive is not a leader.

In the following formulations we use universal and existential quantifications over the set of identities. Strictly speaking, these are not part of PLTL. Since we deal with a finite number of processes the universal quantification $\forall \, i. \, P(i)$, where P is some proposition on processes, can be expanded into $P(1) \; \wedge \; \ldots \; \wedge \; P(N)$, and similarly we can expand $\exists i. \, P(i)$. The quantifications can thus be considered as simply abbreviations.

- "There is always one leader".

  $$\mathsf{G} \, [\exists \, i. \, \mathit{leader}_i \;\; \wedge \;\; (\forall \, j \neq i. \, \neg \, \mathit{leader}_j)]$$

  Since we deal with a dynamic system where all processes may be inactive initially (and thus no leader exists), this property will in general not be valid. Also, in a distributed system with asynchronous communication switching from one leader to another can hardly be made atomic, so it is convenient to allow the temporary absence of a leader. As a first attempt one could think of modifying the above formula into:

  $$\mathsf{G} \, \mathsf{F} \, [\exists \, i. \, \mathit{leader}_i \;\; \wedge \;\; (\forall \, j \neq i. \, \neg \, \mathit{leader}_j)]$$

  which allows there temporarily to be no leader, but which also allows there to be more than one leader at a time temporarily. For consistency reasons this is not desired. We therefore consider the following two properties.

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

  $$\mathsf{G} \, [\mathit{leader}_i \;\; \Rightarrow \;\; \forall \, j \neq i. \, \neg \, \mathit{leader}_j]$$

- "There will be enough leaders in due time". (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 intended.)

  $$\mathsf{G}\,\mathsf{F}\,[\exists\, i.\, \mathit{leader}_i]$$

  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.

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

  $$\mathsf{G}\,[\forall\, i, j.\; ((\mathit{leader}_i \;\wedge\; i < j \;\wedge\; \neg\, \mathit{leader}_j \;\wedge\; \mathit{active}_j) \;\Rightarrow\; \mathsf{F}\,\neg\, \mathit{leader}_i)]$$

  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.

  $$\mathsf{G}\,[\forall\, i, j.\, (\mathit{leader}_i \;\wedge\; \neg\,\mathsf{X}\, \mathit{leader}_i \;\wedge\; \mathsf{X}\,\mathsf{F}\, \mathit{leader}_j) \;\Rightarrow\; i < j]$$

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

2.6 Labelled Büchi automata

The previous section concludes the explanation and definition of PLTL. Model checking PLTL is based on finite automata that accept infinite words. We start in this section by briefly refreshing the notion of finite-state automata.

Automata on finite words

A major difference to the concept of finite-state automata as known from e.g. compiler theory is that states are labelled rather than transitions.

Definition 4. (**Labelled finite-state automaton**)
A labelled finite-state automaton (LFSA, for short) A is a tuple $(\Sigma, S, S^0, \rho, F, l)$ where:

- Σ is a non-empty set of symbols (ranged over by a, b, $\ldots$)
- S is a finite, non-empty set of states
- $S^0 \subseteq S$ is a non-empty set of initial states
- $\rho : S \longrightarrow 2^S$, a transition function
- $F \subseteq S$, a set of accepting states
- $l : S \longrightarrow \Sigma$, the labelling function of states.

By Σ^* we denote the set of finite sequences over Σ, and by Σ^ω the set of infinite sequences over Σ.

$\rho(s)$ is the set of states the automaton A can move into when it is in state s. Notation: we write $s \longrightarrow s'$ iff $s' \in \rho(s)$. An LFSA may be non-deterministic in the following sense: it may have several equally labelled initial states allowing A to start differently on the first input symbol, and the transition function may specify various equally labelled possible successor states for a given state. So, an LFSA is *deterministic* if and only if

$$| \{ s \in S^0 \mid l(s) = a \} | \leqslant 1$$

for all symbols $a \in \Sigma$, and

$$| \{ s' \in \rho(s) \mid l(s') = a \} | \leqslant 1$$

for all states $s \in S$ and all symbols $a \in \Sigma$. Thus, an LFSA is deterministic if the number of equally labelled initial states is at most once, and if for each symbol and each state the successor in uniquely determined.

Definition 5. (**Run of an LFSA**)
For LFSA A a *run* of A is a finite sequence of states $\sigma = s_0\, s_1\, \ldots\, s_n$ such that $s_0 \in S^0$ and $s_i \longrightarrow s_{i+1}$ for all $0 \leqslant i < n$. Run σ is called *accepting* iff $s_n \in F$.

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 $\longrightarrow$ from its predecessor in the sequence. A run is accepting if it ends in an accepting state. A finite word $w = a_0\, a_1\, \ldots\, a_n$ is *accepted* by A iff there exists an accepting run $\sigma = s_0\, s_1\, \ldots\, s_n$ such that $l(s_i) = a_i$ for $0 \leqslant i \leqslant n$. The language accepted by A, denoted $\mathcal{L}(A)$, is the set of finite words accepted by A, so

$$\mathcal{L}(A) = \{\, w \in \Sigma^* \mid w \text{ is accepted by } A \,\}.$$

If $F = \varnothing$ there are no accepting runs, and thus $\mathcal{L}(A) = \varnothing$ in this case.

Example 6. *An example LFSA is depicted in Figure 2.3. Here we have* $\Sigma = \{\, a, b, c \,\}$, $S = \{\, s_1, s_2, s_3 \,\}$, $S^0 = \{\, s_1, s_3 \,\}$, $l(s_1) = a$, $l(s_2) = b$ *and* $l(s_3) = c$. *In addition* $F = \{\, s_2 \,\}$ *and* $\rho(s_1) = \{\, s_2 \,\}$, $\rho(s_2) = \{\, s_1, s_2, s_3 \,\}$ *and* $\rho(s_3) = \{\, s_1, s_3 \,\}$. *The LFSA is deterministic: for each input symbol the starting state is uniquely determined, and for any state the next state on the input of any symbol is uniquely determined.*

Example runs of this automaton are s_1, s_3, $s_3\, s_3\, s_3$, $s_1\, s_2\, s_3$, *and* $s_1\, s_2\, s_2\, s_3\, s_3\, s_1\, s_2$. *Accepting runs are runs that finish in the accepting state* s_2, *for instance,* $s_1\, s_2$, $s_3\, s_1\, s_2$, *and* $s_1\, s_2\, s_2\, s_3\, s_3\, s_1\, s_2$. *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 is labelled with cca.*

Each word accepted by our example automaton consists of zero or more times a symbol c, then an a (in order to be able to reach the accepting state s_2*), and one or more b's, where this sequence of c's, a, and b's can be repeated a finite number* *of times. Formally, the language accepted by this automaton is characterized by the regular expression* $(c^*ab^+)^+$. *(Here* b^+ *means one or more but finitely many times* b*, whereas* b^* *means zero or more times* b*; so,* b^+ *equals* bb^**.) (End of example.)*

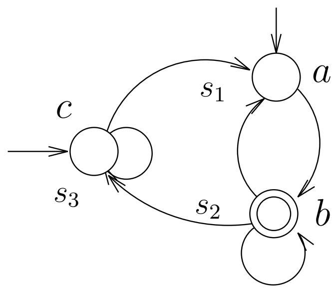


Figure 2.3: An example LFSA

Automata on infinite words

Automata play a special role in model checking PLTL-formulas. 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 an accepting state. The fact that for model checking we consider infinite behavior should not surprise the reader as reactive systems typically do not terminate (cf. Chapter 1). Automata with this alternative characterization of accepting run are called *Büchi automata* or ω-automata. A labelled Büchi automaton (LBA, for short) is an LFSA that accepts infinite words rather than finite words. Note that the automaton itself is still finite: an LBA contains only a finite number of states and finitely many transitions. An LBA thus has the same components as an LFSA, 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?

Definition 6. (**Run of an LBA**)
A run σ of the LBA A is an infinite sequence of states $s_0\, s_1\, \ldots$ such that $s_0 \in S^0$ and $s_i \longrightarrow s_{i+1}$ for $i \geqslant 0$. Let *lim*(σ) be the set of states that occur in σ infinitely often. The run σ is *accepting* iff *lim*$(\sigma) \cap F \neq \varnothing$.

A word $w = a_0\,a_1\,\ldots \in \Sigma^\omega$ is accepting iff there exists an accepting run $s_0\,s_1\,\ldots$ such that $l(s_i) = a_i$ for all $i \geqslant 0$. Since a run of an LBA is infinite we cannot define acceptance by the fact that the final state of a run is accepting or not. Such final state does not exist. Rather according to Büchi's acceptance criterion, a run is accepting if some accepting state is visited infinitely often. Notice that since σ is an infinite sequence of states and S is a finite, non-empty set of states, $lim(\sigma) \neq \varnothing$: there always exists some state that is visited by σ infinitely often. The set of infinite words accepted by the LBA A is the language of A, denoted $\mathcal{L}_\omega(A)$, so

$$\mathcal{L}_\omega(A) = \{\, w \in \Sigma^\omega \mid w \text{ is accepted by } A \,\}.$$

If $F = \varnothing$ there are no accepting states, no accepting runs, and thus $\mathcal{L}_\omega(A) = \varnothing$ in this case.

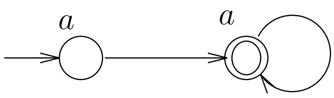
Definition 7. (**Equivalence of Büchi automaton**)
LBAs A and A' are *equivalent* if and only if $\mathcal{L}_\omega(A) = \mathcal{L}_\omega(A')$.

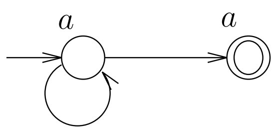
Example 7. Consider the LBA of Figure 2.3. An example run of this Büchi automaton is $s_1\,s_2\,s_2\,s_2\,\ldots$, or shortly, $s_1\,s_2^\omega$. Some other runs are s_3^ω and $(s_1\,s_2\,s_3)^\omega$. The runs that go infinitely often through the accepting state s_2 are accepting. For instance, $s_1\,s_2^\omega$ and $(s_1\,s_2\,s_3)^\omega$ are accepting runs, whereas s_3^ω is not. The accepting words that correspond to these accepting runs are ab^ω and $(abc)^\omega$, respectively. The language accepted by this LBA is $c^*a\,(b\,(c^*a \mid \varepsilon))^\omega$. Here, ε stands for the empty word, and $\mid$ stands for "or", i.e. $(a \mid b)$ means either a or b. So, the LBA 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 does not need to be. (End of example.)

It is interesting to consider more carefully the relation 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 carried out in the following example.

Example 8. In this example we consider some differences between finite-state *automata and Büchi automata. Let A_1 and A_2 be two automata. Let $\mathcal{L}(A_i)$ denote the finite language accepted by A_i ($i = 1, 2$), and $\mathcal{L}_\omega(A_i)$ the infinite language accepted by A_i.*

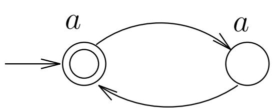
1. *If A_1 and A_2 accept the same finite words, then this does not mean that they also accept the same infinite words. The following two example automata show this:*

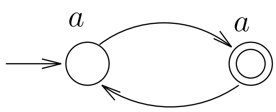
   A_1 

   A_2 

   We have $\mathcal{L}(A_1) = \mathcal{L}(A_2) = \{\, a^{n+2} \mid n \geqslant 0 \,\}$, but $\mathcal{L}_\omega(A_1) = a^\omega$ and $\mathcal{L}_\omega(A_2) = \varnothing$.

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 

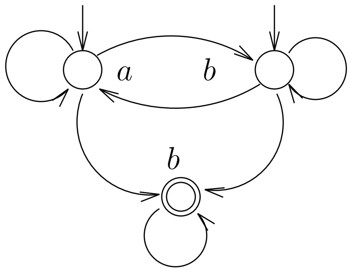
   A_2 

   We have $\mathcal{L}_\omega(A_1) = \mathcal{L}_\omega(A_2) = a^\omega$, but $\mathcal{L}(A_1) = \{\, a^{2n+1} \mid n \geqslant 0 \,\}$ and $\mathcal{L}(A_2) = \{\, a^{2n} \mid n \geqslant 1 \,\}$.

3. *If A_1 and A_2 are deterministic, then $\mathcal{L}(A_1) = \mathcal{L}(A_2) \;\Rightarrow\; \mathcal{L}_\omega(A_1) = \mathcal{L}_\omega(A_2)$. The reverse is, however, not true, as illustrated by the previous example.*

(End of example.)

Another important difference between finite-state automata and Büchi automata is the expressive power of deterministic and non-deterministic automata. Non-deterministic LFSAs are *equally* expressive as deterministic LFSAs, whereas non-deterministic LBAs are strictly *more* expressive than deterministic LBAs. For any non-deterministic LFSA it is known that there exists a deterministic LFSA that accepts the same language (Vardi, 1995). The algorithm for transforming a non-deterministic LFSA into a deterministic one is known as the (Rabin-Scott) subset construction. The number of states of the resulting deterministic automaton is exponential in the number of states of the non-deterministic LFSA. For LBAs such an algorithm does not exist, since non-deterministic LBAs are strictly more expressive than deterministic ones. That is, there exists a non-deterministic LBA for which there does not exist a deterministic version that accepts the same infinite words. For example, the language $(a \mid b)^* b^\omega$ can be accepted by the non-deterministic LBA



whereas there does not exist a deterministic LBA that accepts this language.

Due to the various ways in which the acceptance of infinite words can be defined, different variants of automata over infinite words do exist in the literature. These automata are christened according to the scientist that proposed the acceptance criterion: Streett, Rabin and Muller. For completeness, we have listed the most prominent types of automata over infinite words in Table 2.1 indicating the kind of acceptance sets and acceptance criterion.

2.7 Basic model checking scheme for PLTL

What do Büchi automata have to do with PLTL formulas? Assume that we label the states of an LBA with sets of symbols, i.e. $l : S \longrightarrow 2^\Sigma$. In this case, a word $w = a_0\, a_1 \ldots$ is accepting iff there is an accepting run $s_0\, s_1 \ldots$ such that $a_i \in l(s_i)$, for $i \geqslant 0$. (Up to now we only considered singleton sets of labels, rather than sets of arbitrary cardinality.) Now let $\Sigma = 2^{AP}$, where AP is the set of atomic propositions. Thus, states are labelled with (sets of) sets of atomic

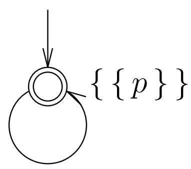
Name	Acceptance sets	Acceptance condition
Büchi	$F \subseteq S$	$lim(\sigma) \cap F \neq \varnothing$
Generalised Büchi	$\mathcal{F} = \{ F_1, \ldots, F_k \}$	$\forall\, i.\, lim(\sigma) \cap F_i \neq \varnothing$
	where $F_i \subseteq S$	
Muller	idem	$\exists\, i.\, lim(\sigma) \cap F_i = F_i$
Rabin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

Table 2.1: Major types of automata on infinite words

propositions, whereas words consist of symbols where each symbol is a set of atomic propositions. The reason that sets of sets of atomic propositions are used, and not just sets, is explained in Example 9. The key feature is:

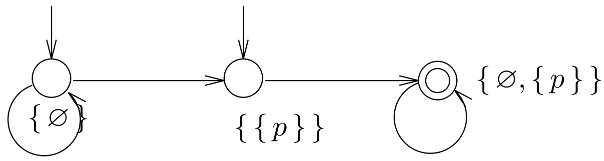
the association of the PLTL*-formula* ϕ *with an LBA*
which accepts all infinite words,
i.e. sequences of (sets of) atomic propositions, which make ϕ *valid.*

Example 9. *Let* $AP = \{ p \}$ *and suppose we have the following LBA that is labelled in the sense explained just above:*



As any accepting run of this Büchi automaton goes infinitely often through a state labelled with $\{ \{ p \} \}$*, the language of this automaton is* $(\{ \{ p \} \})^\omega$*, or* p^ω*, for short. Thus, each accepting run visits a state that satisfies p infinitely often. That is, the accepting runs of the above automaton correspond to the sequences of atomic propositions for which the* PLTL*-formula* G p *holds. (End of example.)*

Example 10. *As a second example, consider* $AP = \{ p \}$ *and the following LBA:*



This automaton can either start in a state satisfying p *(i.e. the state labelled by* $\{\{p\}\}$*, or in a state not satisfying* p *(i.e. the state labelled by* $\{\varnothing\}$*. Once a run has passed the* p*-state, it accepts any run that goes infinitely often through the rightmost state. As this state is labelled with* $\neg p$ *(i.e.* $\varnothing$*) and with* p *(i.e.* $\{p\}$*), it means that any run whose elements are labelled by* $p \vee \neg p$ *(i.e. true) is accepted. Thus, this Büchi automaton accepts any run that once satisfies* p*, and anything else afterwards. This corresponds to the formula* $\mathsf{F}\, p$*. (End of example.)*

From these examples we infer that $\{\varnothing\}$ 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

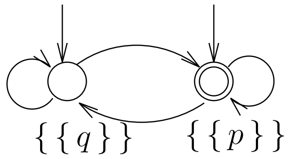
$$(\forall\, p \in AP_i : p) \;\wedge\; (\forall\, p \in AP - AP_i : \neg p)$$

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

$$[\![AP_{k_1}]\!] \;\vee\; \ldots \;\vee\; [\![AP_{k_m}]\!]$$

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

Example 11. Let $AP = \{p, q\}$ *and consider the Büchi automaton*



Given the above encoding of sets into propositional formulas, we can consider the leftmost state being labelled with $q \wedge \neg p$ *and the rightmost state with* $p \wedge \neg q$*. Any accepting run of this automaton thus visits a state that satisfies* $p \wedge \neg q$ *infinitely often, while between any two successive states a state satisfying* $q \wedge \neg p$ *may be visited for a finite number of times. Thus, the accepting runs of this automaton correspond to the accepting sequences of the formula*

$$G[(q \wedge \neg p) \,\mathsf{U}\, (p \wedge \neg q)]$$

(End of example.)

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

Theorem 8.
For any PLTL*-formula* ϕ *a Büchi automaton* A *can be constructed on the alphabet* $\Sigma = 2^{AP}$ *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 LBA 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 2.2. 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 2.2: 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. 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.[2]

Observe that

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

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 LBA 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 LBAs an algorithm exists that is polynomial in the size of A, but since deterministic LBAs are strictly less expressive than non-deterministic LBAs, 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 2.3. The idea is to construct an LBA 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

[2]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.

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 2.3: Basic recipe for model checking PLTL

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 LBA 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.

2.8 From PLTL-formulas to labelled Büchi automata

The algorithm that we treat here for associating an LBA with a PLTL-formula is adopted from Gerth, Peled, Vardi & Wolper (1995) and consists of the steps indicated in Figure 2.4. We deal with each step separately. The main step is

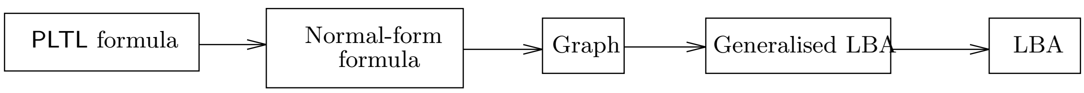


Figure 2.4: Schematic view of PLTL to LBA algorithm

the construction of a graph starting from a normal-form formula. The reason for adopting this more recent algorithm rather than the original algorithm by Wolper, Vardi and Sistla (1983) is that in general the resulting LBAs for the recent algorithm are substantially smaller. The worst-case complexity of both algorithms is, however, identical.

Normal-form formulas

In order to construct an LBA for a given PLTL-formula ϕ, ϕ is first transformed into *normal form*. This means that ϕ does not contain F and G (one can easily eliminate them by using the definitions $\mathsf{F}\,\psi \equiv \text{true}\,\mathsf{U}\,\psi$ and $\mathsf{G}\,\psi \equiv \neg\,\mathsf{F}\,\neg\,\psi$), and that all negations in ϕ are adjacent to atomic propositions. 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, an auxiliary temporal operator $\overline{\mathsf{U}}$ is introduced which is defined by

$$(\neg\,\phi)\,\overline{\mathsf{U}}\,(\neg\,\psi) \equiv \neg\,(\phi\,\mathsf{U}\,\psi).$$

In the original algorithm this operator was denoted V, but in order to avoid confusion with $\vee$ (disjunction) we use $\overline{\mathsf{U}}$.

Definition 9. (**Normal-form PLTL-formulas**)
For $p \in AP$, an atomic proposition, the set of PLTL-formulas in normal form is defined by:

$$\phi ::= p \;\Big|\; \neg\,p \;\Big|\; \phi \;\vee\; \phi \;\Big|\; \phi \;\wedge\; \phi \;\Big|\; \mathsf{X}\,\phi \;\Big|\; \phi\,\mathsf{U}\,\phi \;\Big|\; \phi\,\overline{\mathsf{U}}\,\phi.$$

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

$$\begin{array}{rcl}
\neg\,(\phi \;\vee\; \psi) & \equiv & (\neg\,\phi) \;\wedge\; (\neg\,\psi) \\
\neg\,(\phi \;\wedge\; \psi) & \equiv & (\neg\,\phi) \;\vee\; (\neg\,\psi) \\
\neg\,\mathsf{X}\,\phi & \equiv & \mathsf{X}\,(\neg\,\phi) \\
\neg\,(\phi\,\mathsf{U}\,\psi) & \equiv & (\neg\,\phi)\,\overline{\mathsf{U}}\,(\neg\,\psi) \\
\neg\,(\phi\,\overline{\mathsf{U}}\,\psi) & \equiv & (\neg\,\phi)\,\mathsf{U}\,(\neg\,\psi)
\end{array}$$

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

Example 12. *As an example we derive the normal-form of* $\neg\,\mathsf{X}\,[r \;\Rightarrow\; p\,\mathsf{U}\,q]$.

$$
\begin{array}{ll}
 & \neg\,\mathsf{X}\,[r \;\Rightarrow\; p\,\mathsf{U}\,q] \\
\Leftrightarrow & \{\ \textit{definition of}\ \Rightarrow\ \} \\
 & \neg\,\mathsf{X}\,[\neg\, r \;\vee\; (p\,\mathsf{U}\,q)] \\
\Leftrightarrow & \{\ \neg\,\mathsf{X}\,\phi \;\Leftrightarrow\; \mathsf{X}\,(\neg\,\phi)\ \} \\
 & \mathsf{X}\,[\neg\,(\neg\, r \;\vee\; (p\,\mathsf{U}\,q))] \\
\Leftrightarrow & \{\ \textit{predicate calculus}\ \} \\
 & \mathsf{X}\,[r \;\wedge\; \neg\,(p\,\mathsf{U}\,q)] \\
\Leftrightarrow & \{\ \textit{definition of}\ \overline{\mathsf{U}}\ \} \\
 & \mathsf{X}\,[r \;\wedge\; (\neg\, p)\,\overline{\mathsf{U}}\,(\neg\, q)].
\end{array}
$$

As a next example, the interested reader might check that the normal form of $\mathsf{G}\,p$ *is* $\mathit{false}\,\overline{\mathsf{U}}\,p$. *(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 size of ϕ, denoted by $\mid \phi \mid$. The size 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.

Program notation (intermezzo)

In order to present the algorithms in these lecture notes in a uniform style, and — perhaps even more importantly — in order not to focus on a programming language which might be popular today, but old-fashioned as soon as a new language is launched, we choose an *abstract* program notation. This notation is strongly based on Dijkstra's guarded command language (Dijkstra, 1976). In a nutshell the following constructs are used in this language. Statements can be sequenced by using the symbol ; as a separator.

- The empty statement (denoted **skip**). It simply means "do nothing".

- The let statement. It allows the non-deterministic choice of some element of a given set. For instance, **let** x **in** V, where x is a variable and V a non-empty set, establishes that some (arbitrary) element of V is chosen and assigned to x.

- Concurrent assignment statement. Here a number of different variables can be assigned simultaneously. Thus one can write $x, y := E_1, E_2$. The interchange of variables can now be described simply by $x, y := y, x$.

- Guarded commands. A guarded command consists of a guard, that is, a boolean expression, and a list of statements. The list of statements is only executed if the guard is true initially. For instance, the guarded command $(V \neq \varnothing) \longrightarrow x, y := 0, 7$ means that the assignment $x, y := 0, 7$ will be executed provided the set V is non-empty. Guarded commands can be sequenced by using the symbol [] as a separator. (Note that guarded commands are not statements, but are part of statements.)

- If-statement. An if-statement has the form

  if guarded command [] ... [] guarded command **fi**

  where all guards are assumed to be defined (i.e., the evaluation of a guard always terminates). One of the guarded commands whose guard is true is selected and the associated statement list is executed. If more than one guard is true, the selection is made non-deterministically from the guarded commands whose guards are true. If no guard is true, the statement is aborted.

- Do-statement. A do-statement has the form

  do guarded command [] ... [] guarded command **od**

  If there are no true guards, the statement terminates; if there are true guards, one of them is selected non-deterministically and its statement list is executed. After execution of a selected statement list, all guards are re-evaluated. Thus the execution of the do-statement continues until there are no true guards left.

- Comments. A comment is enclosed between the brackets (* and *).

This concludes the description of the program notation. We will be quite liberal in the types of variables that we use. In fact we allow sets, sequences, bags (multi-sets), and so on, without aiming at a particular realization of these types. For sets we use the common notation whereas for sequences we use $\langle\,\rangle$ to denote an empty sequence, $\langle v \rangle$ to denote the sequence consisting of a single element v, and $\langle v \rangle ^\frown S$ and $S ^\frown \langle v \rangle$ for adding an element v at the front or at the end of sequence S.

Constructing a graph

Suppose the transformation of the PLTL-formula at hand into normal form yields a formula ϕ. In the next step of the algorithm (cf. Table 2.4) a graph $\mathcal{G}_\phi = (V, E)$ is constructed, with a set of vertices V, and a set of edges $E \subseteq V \times V$. Vertices have the following structure:

Definition 10. (**Vertex**)
A *vertex* v is a tuple (P, N, O, Sc) with

- $P \in 2^{V \,\cup\, \{\, init \,\}}$, a set of predecessors
- N, O, Sc, sets of normal-form formulas.

For $v = (P, N, O, Sc)$ we write $P(v) = P$, $N(v) = N$, $O(v) = O$ and $Sc(v) = Sc$.

P is the set of predecessors of a vertex. A special name *init* is used for indicating initial vertices (that is, vertices without incoming edges), $init \notin V$. So, vertex v is initial iff $init \in P(v)$. *init* is just a placeholder and not a physical vertex. $N(v) \cup O(v)$ are the formulas that are to be checked for v. N (New) is the set of formulas that have not yet been processed; O (Old) is the set of formulas that have already been processed. Sc (Succ) contains those formulas that must hold at all vertices which are immediate successors of vertices satisfying the formulas in O.

The graph $\mathcal{G}_\phi$ is constructed in a *depth-first traversal order* starting with a single vertex labelled with ϕ, the input formula. Selecting in each step as the next

function *CreateGraph* (ϕ : *Formula*): **set of** *Vertex*;
(* pre-condition: ϕ is a PLTL-formula in normal form *)
begin var S : **sequence of** *Vertex*,
  Z : **set of** *Vertex*, (* already explored vertices *)
  v, w_1, w_2 : *Vertex*;
  $S, Z := \langle (\{ \mathit{init} \}, \{ \phi \}, \varnothing, \varnothing) \rangle, \varnothing;$
  do $S \neq \langle \, \rangle \longrightarrow$ (* let $S = \langle v \rangle ^\frown S'$ *)
    if $N(v) = \varnothing \longrightarrow$ (* all proof obligations of v have been checked *)
      if $(\exists \, w \in Z. \, \mathit{Sc}(v) = \mathit{Sc}(w) \ \wedge \ O(v) = O(w)) \longrightarrow$
        $P(w), S := P(w) \cup P(v), S'$ (* w is a copy of v *)
      [] $\neg \, (\exists \, w \in Z. \, \ldots) \longrightarrow$
        $S, Z := \langle (\{ v \}, \mathit{Sc}(v), \varnothing, \varnothing) \rangle ^\frown S', Z \cup \{ v \};$
      fi
    [] $N(v) \neq \varnothing \longrightarrow$ (* some proof obligations of v are left *)
      let ψ **in** $N(v)$;
      $N(v) := N(v) \setminus \{ \psi \};$
      if $\psi \in AP \ \vee \ (\neg \, \psi) \in AP \longrightarrow$
        if $(\neg \, \psi) \in O(v) \longrightarrow S := S'$ (* discard v *)
        [] $\neg \, \psi \notin O(v) \longrightarrow$ **skip**
        fi
      [] $\psi = (\psi_1 \ \wedge \ \psi_2) \longrightarrow N(v) := N(v) \cup (\{ \psi_1, \psi_2 \} \setminus O(v))$
      [] $\psi = \mathsf{X} \, \varphi \longrightarrow \mathit{Sc}(v) := \mathit{Sc}(v) \cup \{ \varphi \}$
      [] $\psi \in \{ \psi_1 \, \mathsf{U} \, \psi_2, \psi_1 \, \overline{\mathsf{U}} \, \psi_2, \psi_1 \vee \psi_2 \} \longrightarrow$ (* split v *)
        $w_1, w_2 := v, v;$
        $N(w_1) := N(w_1) \cup (F_1(\psi) \setminus O(w_1));$
        $N(w_2) := N(w_2) \cup (F_2(\psi) \setminus O(w_2));$
        $O(w_1), O(w_2) := O(w_1) \cup \{ \psi \}, O(w_2) \cup \{ \psi \};$
        $S := \langle w_1 \rangle ^\frown (\langle w_2 \rangle ^\frown S')$
      fi
      $O(v) := O(v) \cup \{ \psi \}$
    fi
  od;
  return Z;
(* post-condition: Z is the set of vertices of the graph $\mathcal{G}_\phi$ *)
(* where the initial vertices are vertices in Z with $\mathit{init} \in P$ *)
(* and the edges are given by the P-components of vertices in Z *)
end

Table 2.4: Algorithm for constructing a graph for PLTL-formula ϕ

state a successor state of the state that has been most recently reached which has non-visited successor states is called a depth-first search strategy. The sequence S (usually implemented as a stack) contains the vertices in depth-first order that have been visited but that not have yet been explored completely. Initially S consists of one vertex *init* labeled with ϕ. The set Z contains the set of vertices created, initially $\varnothing$. When all vertices have been explored (i.e. $S = \langle\,\rangle$) we are finished, and the algorithm terminates. Vertex v is fully explored when all the formulas that must hold in v have been checked, that is, when $N(v) = \varnothing$.

Suppose $N(v) = \varnothing$. If there exists already a vertex (in Z) with the same proof obligations in O and Sc, then no new vertex has to be created. Instead, it suffices to take the existing vertex in Z and augment its incoming edges with those of v, indicating that the existing vertex can now be reached in the same way as v. In addition, v is removed from S since we have finished with v. If such a copy of v does not exist, a vertex is created (that is, Z is extended with v), and the successors of v have to be explored (a new element is added to S labelled with the proof obligations for all successor vertices of v, $Sc(v)$).

Suppose that there are still formulas to be checked, i.e. $N(v) \neq \varnothing$. At each state a sub-formula ψ of ϕ that remains to be satisfied is selected from $N(v)$ and is decomposed according to its structure. The following cases are distinguished:

- ψ is a negation of some atomic proposition that has been processed before. Then we have found a contradiction and the vertex v is not further explored (i.e. $S := S'$).
- ψ is some atomic proposition whose negation has not been processed before. Then do nothing.
- ψ is a conjunction. Clearly, in order for $\psi_1 \wedge \psi_2$ to be satisfied we have to check whether ψ_1 and ψ_2 are both satisfied. Thus, ψ_1 and ψ_2 are added to $N(v)$. In order to avoid that ψ_1 and ψ_2 are checked more than once we only add them to $N(v)$ if they do not have been processed before (i.e. $\psi_i \notin O(v)$).
- ψ is a next-formula, say $\mathsf{X}\,\varphi$. In order for v to satisfy ψ, it suffices that φ holds at all immediate successors of v. Thus, φ is added to $Sc(v)$.

- ψ is a disjunction, a U- or a $\overline{\mathsf{U}}$-formula. For these cases the vertex v is *split* into two vertices (w_1 and w_2).

  These vertices correspond to the two ways in which ψ can be made valid. The way in which v is split depends on the structure of ψ. The basic principle is to rewrite ψ into a disjunction such that each part of the disjunction can be checked in each of the split versions of v, the vertices w_1 and w_2.

  - $\psi = \psi_1 \vee \psi_2$. Then the validity of either ψ_1 or ψ_2 suffices to make ψ valid. Given that $F_i(\psi) = \{ \psi_i \}$ for i=1, 2, this reduces to checking either ψ_1 (in w_1) or ψ_2 (in w_2).
  - $\psi = \psi_1 \,\mathsf{U}\, \psi_2$. Here the equation, one of the expansion laws we have seen before,

    $$\psi_1 \,\mathsf{U}\, \psi_2 \;\Leftrightarrow\; \psi_2 \;\vee\; [\psi_1 \;\wedge\; \mathsf{X}\,(\psi_1 \,\mathsf{U}\, \psi_2)]$$

    is used to split v. One vertex (w_1) is used for checking $\psi_1 \wedge \mathsf{X}\,(\psi_1 \,\mathsf{U}\, \psi_2)$: $F_1(\psi) = \{ \psi_1 \;\wedge\; \mathsf{X}\,(\psi_1 \,\mathsf{U}\, \psi_2) \}$. The other vertex ($w_2$) is used for checking ψ_2: $F_2(\psi) = \{ \psi_2 \}$.
  - $\psi = \psi_1 \,\overline{\mathsf{U}}\, \psi_2$. Here the equation

    $$\psi_1 \,\overline{\mathsf{U}}\, \psi_2 \;\Leftrightarrow\; \psi_2 \;\wedge\; [\psi_1 \;\vee\; \mathsf{X}\,(\psi_1 \,\overline{\mathsf{U}}\, \psi_2)]$$

    is used to split v. Rewriting this in disjunctive form yields

    $$\psi_2 \;\wedge\; [\psi_1 \;\vee\; \mathsf{X}\,(\psi_1 \,\overline{\mathsf{U}}\, \psi_2)] \;\Leftrightarrow\; (\psi_2 \;\wedge\; \psi_1) \;\vee\; [\psi_2 \;\wedge\; \mathsf{X}\,(\psi_1 \,\overline{\mathsf{U}}\, \psi_2)].$$

    In this case $F_1(\psi) = \{ \psi_1 \;\wedge\; \psi_2 \}$ and $F_2(\psi) = \{ \psi_2 \;\wedge\; \mathsf{X}\,(\psi_1 \,\overline{\mathsf{U}}\, \psi_2) \}$.

  The search proceeds in depth-first order by considering w_1 and w_2.

Example 13. As an example of the CreateGraph *algorithm consider $\phi = p \,\mathsf{U}\, q$ where p and q are atomic propositions. It is left to the interested reader to check that the result of applying* CreateGraph *to this formula results in the graph depicted in Figure 2.5. (End of example.)*

We conclude by a short complexity analysis of this step. Since in principle each set of sub-formulas of ϕ can give rise to a vertex in the graph $\mathcal{G}_\phi$, the

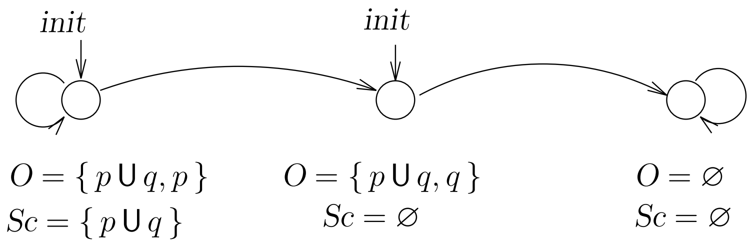


Figure 2.5: Result of applying the *CreateGraph*-algorithm to $p \,\mathsf{U}\, q$

number of vertices is (in the worst case) proportional to the number of subsets of sub-formulas of ϕ. Since the number of sub-formulas of a formula is equal to the length of the formula, the number of vertices is in the worst case equal to $2^{|\phi|}$. The worst-case time complexity of this graph-creating algorithm is therefore $\mathcal{O}(2^{|\phi|})$. We will address later on how this influences the worst-case time complexity of model checking PLTL.

From a graph to a generalized LBA

A *generalized* LBA can be obtained in the following way from the graph constructed according to the algorithm of the previous section. First we introduce the notion of generalized LBA.

Definition 11. (**Generalised LBA**)
A *generalized labelled Büchi automaton* (GLBA) A is a tuple $(\Sigma, S, S^0, \rho, \mathcal{F}, l)$ where all components are the same as for an LBA, except that $\mathcal{F}$ is a *set* of accepting sets $\{\, F_1, \ldots, F_k \,\}$ for $k \geqslant 0$ with $F_i \subseteq S$, i.e. $\mathcal{F} \subseteq 2^S$ rather than a single set of accepting states.

Run $\sigma = s_0\, s_1 \ldots$ is accepting for GLBA A if and only if for each acceptance set $F_i \in \mathcal{F}$ there exists a state in F_i that appears in σ infinitely often. Recall that $lim(\sigma)$ is the set of states that appear infinitely often in σ. The acceptance condition formally amounts to

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

Note that if $\mathcal{F} = \varnothing$ all runs go infinitely often through all acceptance sets in $\mathcal{F}$, so any run is accepting in this case.

Example 14. *Consider the GLBA depicted in Figure 2.6. 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 $a\,(b\,(a \mid \varepsilon))^{\omega}$.*

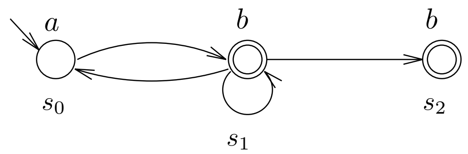


Figure 2.6: An example generalized Büchi automaton

If we now let $\mathcal{F}$ to consist of $\{ s_1 \}$ and $\{ s_2 \}$, i.e. two accepting sets consisting of s_1 and s_2, respectively, the GLBA has no accepting run, and thus the 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.)

Definition 12. (**GLBA from graph**)
For the PLTL-formula ϕ in normal form, the associated GLBA $A = (\Sigma, S, S^0, \rho, \mathcal{F}, l)$ is defined by:

- $\Sigma = 2^{AP}$
- $S = \mathit{CreateGraph}(\phi)$
- $S^0 = \{ s \in S \mid \mathit{init} \in P(s) \}$
- $s \longrightarrow s'$ iff $s \in P(s')$ and $s \neq \mathit{init}$
- $\mathcal{F} = \{ \{ s \in S \mid \phi_1 \,\mathsf{U}\, \phi_2 \notin O(s) \;\vee\; \phi_2 \in O(s) \} \mid \phi_1 \,\mathsf{U}\, \phi_2 \in \mathit{Sub}(\phi) \}$

- $l(s) = \{ \mathcal{P} \subseteq AP \mid Pos(s) \subseteq \mathcal{P} \ \wedge \ \mathcal{P} \cap Neg(s) = \varnothing \}$.

$Sub(\phi)$ denotes the set of sub-formulas of ϕ. This set can easily be defined by induction on the structure of the formula and is omitted here. $Pos(s) = O(s) \cap AP$, the atomic propositions that are valid in s, and $Neg(s) = \{ p \in AP \mid \neg p \in O(s) \}$, the set of negative atomic propositions that are valid in s.

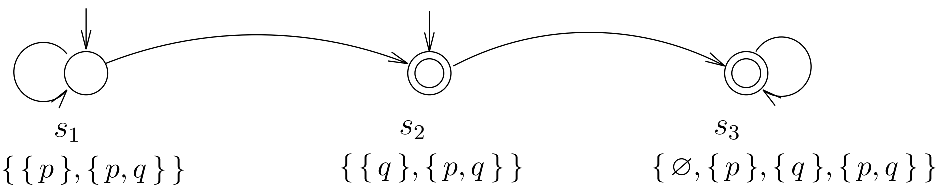
The set of states is simply the set of vertices generated by the function *CreateGraph* applied to ϕ, the normal-form formula under consideration. Notice that only those vertices v for which $N(v) = \varnothing$ are returned by this function. For these vertices all formulas to be checked have been considered. The set of initial states are those states which have the special marking *init* in the set of predecessors. We have a transition from s to s' if s is a predecessor of s' and $s \neq init$. For each sub-formula $\phi_1 \mathsf{U} \phi_2$ in ϕ an acceptance set is defined that consists of those states s such that either $\phi_2 \in O(s)$, or $\phi_1 \mathsf{U} \phi_2 \notin O(s)$. The number of sets of acceptance thus equals the number of U-sub-formulas of ϕ. For formulas without U-sub-formulas $\mathcal{F}$ equals $\varnothing$, denoting that all runs are accepting. States are labelled with sets of sets of atomic propositions. A state s is labelled with any set of atomic propositions that contain the valid propositions in s, $Pos(s)$ and which does not contain any proposition in $Neg(s)$, the negative propositions that are valid in s.

The above construction of acceptance sets avoids to accept a run $\sigma = s_0\, s_1 \ldots$ where $\phi_1 \mathsf{U} \phi_2$ in $O(s_i)$, but for which there is no s_j, $j > i$ for which ϕ_2 is valid. (Note that the acceptance of these runs is not avoided by the construction of the graph by *CreateGraph*.) From the *CreateGraph*-algorithm it follows that if $\phi_1 \mathsf{U} \phi_2 \in O(s_i)$ and $\phi_2 \notin O(s_{i+1})$, then $\phi_1 \mathsf{U} \phi_2 \in O(s_{i+1})$. This can be seen by the splitting of a vertex labelled with $\phi_1 \mathsf{U} \phi_2$ in the algorithm: the vertex that is not labelled with ϕ_2 is labelled with $\mathsf{X}(\phi_1 \mathsf{U} \phi_2)$. So, if $\phi_1 \mathsf{U} \phi_2$ is valid in s_i then it is also valid in state s_{i+1} (given that ϕ_2 is not valid in that state). Now, let F_i be the acceptance set associated with $\phi_1 \mathsf{U} \phi_2$. σ as described above is not an accepting run, since there is no state in σ that is in F_i, so there is no accepting state in F_i that is visited infinitely often by σ.

The correctness of the construction follows from the following result (by Gerth, Peled, Vardi & Wolper, 1995):

Theorem 13.
The GLBA obtained by transforming the graph generated by CreateGraph(ϕ) *as described above accepts exactly those sequences over* $(2^{AP})^{\omega}$ *that satisfy the normal-form* PLTL*-formula* ϕ.

Example 15. *The GLBA that corresponds to the graph in Figure 2.5 is:*



The two accepting states belong to one acceptance set, that is $\mathcal{F} = \{\{s_2, s_3\}\}$. *The labels of the states deserve some explanation. For* s_1 *we have* $\text{Pos}(s_1) = O(s_1) \cap AP = \{p\}$, *and* $\text{Neg}(s_1) = \varnothing$, *since* s_1 *does not contain any negative atomic proposition in* $O(s_1)$. *Consequently,* $l(s_1) = \{\{p\},\{p,q\}\}$. *A similar reasoning justifies the labelling of the other states. The only accepting state that can be visited infinitely often is* s_3. *In* s_3 *it is irrelevant what is valid. It is only important that* s_3 *is reached, since then* $p \,\mathsf{U}\, q$ *is guaranteed. In order to reach state* s_3, *state* s_2 *must be visited first. Therefore each accepting run of this GLBA has a finite prefix for which* p *is valid until* q *is valid (in state* s_2*). Either* q *is valid initially, by starting in* s_2, *or one reaches* s_2 *from having started in* s_1 *where* p *is true (and sometimes even already* q*).* *(End of example.)*

To give the reader some idea of the size of the GLBA that is obtained for a given PLTL-formula we list for some formulas the following: number of states $|S|$, transitions $|\rho|$, and acceptance sets $|\mathcal{F}|$ (Gerth et. al, 1995).

Formula	$\lvert S\rvert$	$\lvert\rho\rvert$	$\lvert\mathcal{F}\rvert$	Formula	$\lvert S\rvert$	$\lvert\rho\rvert$	$\lvert\mathcal{F}\rvert$
$p \,\mathsf{U}\, q$	3	4	1	$(\mathsf{F}\, p) \,\mathsf{U}\, (\mathsf{G}\, q)$	8	15	2
$p \,\mathsf{U}\, (q \,\mathsf{U}\, r)$	4	6	2	$(\mathsf{G}\, p) \,\mathsf{U}\, q$	5	6	1
$\neg\, [p \,\mathsf{U}\, (q \,\mathsf{U}\, r)]$	7	15	0	$\neg\, (\mathsf{F}\,\mathsf{F}\, p \Leftrightarrow \mathsf{F}\, p)$	22	41	2
$\mathsf{G}\,\mathsf{F}\, p \Rightarrow \mathsf{G}\,\mathsf{F}\, q$	9	15	2				

Notice that for $\neg\,[p \,\mathsf{U}\, (q \,\mathsf{U}\, r)]$ no accepting states are obtained, since the normal form of this formula does not contain an until-formula. Therefore any run of the automaton that is obtained for this formula is accepting.

From a generalized LBA to an LBA

The method of transforming a generalized LBA A with k acceptance sets into an LBA A' is to make k copies of A, one copy per acceptance set. Then each state s 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 accepting state in F_i is reached in the i-th copy, then the automaton switches to the $(i{+}1)$-th copy. This transformation is formally defined in:

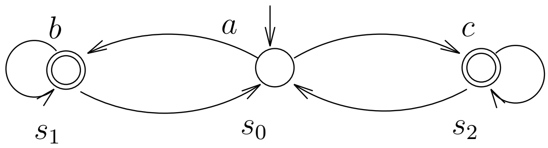
Definition 14. (**From a GLBA to an LBA**)
Let $A = (\Sigma, S, S^0, \rho, \mathcal{F}, l)$ be a GLBA with $\mathcal{F} = \{ F_1, \ldots, F_k \}$. The ordinary LBA $A' = (\Sigma, S', S^{0'}, \rho', F', l')$ such that $\mathcal{L}_\omega(A) = \mathcal{L}_\omega(A')$ is obtained as follows

- $S' = S \times \{ i \mid 0 < i \leqslant k \}$
- $S^{0'} = S^0 \times \{ i \}$ for some $0 < i \leqslant k$
- $(s, i) \longrightarrow' (s', i)$ iff $s \longrightarrow s'$ and $s \notin F_i$
  $(s, i) \longrightarrow' (s', (i \textbf{ mod } k) + 1)$ iff $s \longrightarrow s'$ and $s \in F_i$
- $F' = F_i \times \{ i \}$ for some $0 < i \leqslant k$
- $l'(s, i) = l(s)$.

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

For a run of A' to be accepting it has to visit some state (s, i) infinitely often, where s is an accepting state in F_i in the GLBA A. As soon as a run reaches this state (s, i) the automaton 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 accepting 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 accepting state in each copy infinitely often. Given this concept it is not hard to see that A and A' accept the same language, and hence are equivalent.

Example 16. Consider the following generalized LBA:



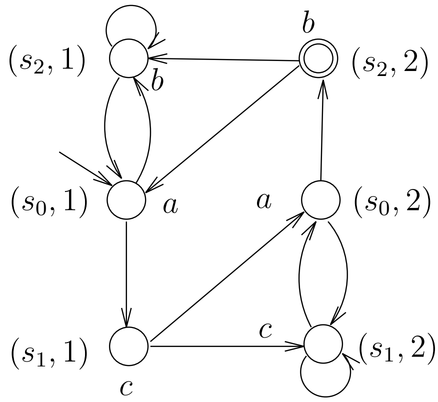
It has two acceptance sets $F_1 = \{ s_1 \}$ *and* $F_2 = \{ s_2 \}$*. The states of the corresponding simple LBA are*

$$\{ s_0, s_1, s_2 \} \times \{ 1, 2 \}$$

Some example transitions are:

$$\begin{array}{llll}
(s_0, 1) \longrightarrow (s_1, 1) & \textit{since} & s_0 \longrightarrow s_1 & \textit{and } s_0 \notin F_1 \\
(s_0, 1) \longrightarrow (s_2, 1) & \textit{since} & s_0 \longrightarrow s_2 & \textit{and } s_0 \notin F_1 \\
(s_1, 1) \longrightarrow (s_0, 2) & \textit{since} & s_1 \longrightarrow s_0 & \textit{and } s_1 \in F_1 \\
(s_1, 2) \longrightarrow (s_1, 2) & \textit{since} & s_1 \longrightarrow s_1 & \textit{and } s_1 \notin F_2 \\
(s_2, 2) \longrightarrow (s_2, 1) & \textit{since} & s_2 \longrightarrow s_2 & \textit{and } s_2 \in F_2
\end{array}$$

The justification of the other transitions is left to the reader. A possible resulting simple LBA is



where the set of initial states is chosen to be $\{ (s_0, 1) \}$ *and the set of accepting states is chosen to be* $\{ (s_2, 2) \}$. *Any accepting run of the simple LBA 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 LBA visits some state of* F_1 *and some state of* F_2 *infinitely often.* *(End of example.)*

This concludes the description of how to obtain a (simple) LBA A_ϕ for a given PLTL-formula ϕ. Notice that the number of states in the simple LBA A is $\mathcal{O}(k \times \mid S \mid)$ where S is the set of states of the GLBA under consideration and k the number of acceptance sets.

2.9 Checking for emptiness

We have seen how to convert a PLTL-formula ϕ into a Büchi automaton A_ϕ. We assume that a model of the system to be verified is also given as a Büchi automaton A_{sys}. Then the third step of the model checking method (cf. Table 2.3) is to check whether the runs accepted by these automata are disjoint, i.e. whether

$$\mathcal{L}_\omega(A_{sys}) \cap \mathcal{L}_\omega(A_{\neg \phi}) = \varnothing$$

$\mathcal{L}_\omega(A_{\neg \phi})$ represents all computations that satisfy $\neg \phi$, that is, it characterizes those computations that violate the desired property ϕ. Stated differently, $A_{\neg \phi}$

characterizes the undesired behavior. $\mathcal{L}_\omega(A_{sys})$ represents all possible behavior of the model of the system. The model *sys* satisfies property ϕ if there is no computation that fulfill $\neg\,\phi$, thus if there is no possible undesired behavior.

Checking whether A_{sys} and $A_{\neg\,\phi}$ have some accepting run in common consists of the steps described in Table 2.5. First a new automaton is constructed that accepts the intersection of the languages of A_{sys} and $A_{\neg\,\phi}$. This is called the (synchronous) product automaton. Since Büchi automata are closed under products, such an automaton always exists. The problem of checking whether a given automaton generates the empty language is known as the *emptiness problem*.

1.	*construct the product automaton* $A_{sys} \otimes A_{\neg\,\phi}$ *which accepts the language* $\mathcal{L}_\omega(A_{sys}) \cap \mathcal{L}_\omega(A_{\neg\,\phi})$, *and*
2.	*check whether* $\mathcal{L}_\omega(A_{sys} \otimes A_{\neg\,\phi}) = \varnothing$.

Table 2.5: Method for solving the emptiness problem

Product of automata on finite words

In order to better understand the product automaton construction for Büchi automata we first deal with ordinary finite-state automata. For labelled finite-state automata A_1 and A_2 it is not difficult to construct an automaton that accepts the language $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$, viz. the automaton $A_1 \times A_2$. The product operator $\times$ is defined by

Definition 15. (**Synchronous product of LFSA**)
Let $A_i = (\Sigma, S_i, S_i^0, \rho_i, F_i, l_i)$ for i=1, 2 be two LFSA (that is, automata over finite words). The *product automaton* $A_1 \times A_2 = (\Sigma, S, S^0, \rho, F, l)$ is defined as follows

- $S = \{\, (s, s') \in S_1 \times S_2 \mid l_1(s) = l_2(s')\,\}$
- $S^0 = (S_1^0 \times S_2^0) \cap S$
- $(s_1, s_2) \longrightarrow (s_1', s_2')$ iff $s_1 \longrightarrow_1 s_1'$ and $s_2 \longrightarrow_2 s_2'$

- $F = (F_1 \times F_2) \cap S$
- $l(s, s') = l_1(s)$ (which equals $l_2(s')$).

States are pairs (s, s') such that $s \in S_1$ and $s' \in S_2$ and s and s' are equally labelled. It is not difficult to see that $\mathcal{L}(A_1 \times A_2) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$. Every accepting run $(s_0, s'_0)\,(s_1, s'_1)\,\ldots\,(s_n, s'_n)$ of $A_1 \times A_2$ must end in a state $(s_n, s'_n) \in F$. 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 if $s_n \in F_1$. Similarly, $s'_0\,s'_1\,\ldots\,s'_n$ is an accepting run of A_2 if $s'_n \in F_2$. Thus any accepting run of $A_1 \times A_2$ must be an accepting run of A_i when projected on a run of A_i for i=1, 2.

Product of automata on infinite words

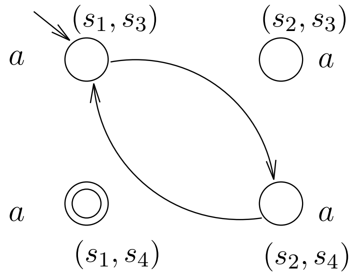
For Büchi automata this simple procedure is, however, not appropriate. In the above construction, the set of accepting states F is the product of acceptance sets F_1 and F_2. When we view A_1 and A_2 as automata on infinite words this construction means that $A_1 \times A_2$ accepts an infinite word w if there are accepting runs σ_1 and σ_2 of A_1 and A_2 on w, where both runs go infinitely often and *simultaneously* through accepting 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 17. Consider the two Büchi automata

The language accepted by these LBA is: $\mathcal{L}_\omega(A_1) = \mathcal{L}_\omega(A_2) = a^\omega$. *But the automaton* $A_1 \times A_2$ *is:*



$A_1 \times A_2$

which, clearly has no accepting run. So, $\mathcal{L}_\omega(A_1 \times A_2) = \varnothing$ *which differs from* $\mathcal{L}_\omega(A_1) \cap \mathcal{L}_\omega(A_2) = a^\omega$. *The point is that the product automaton assumes that* A_1 *and* A_2 *go simultaneously through an accepting 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 accepting state,* A_2 *is not, and vice versa.*

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

Fortunately there is a modification of the pure product automata construction that works for Büchi automata, due to Choueka (1974).

Definition 16. (**Synchronous product of Büchi automata**)
Let $A_i = (\Sigma, S_i, S_i^0, \rho_i, F_i, l_i)$ for i=1, 2 be two LBAs (that is, automata over infinite words). The *product automaton* $A_1 \otimes A_2 = (\Sigma, S, S^0, \rho, F, l)$ is defined as follows

- $S = \{\, (s, s') \in S_1 \times S_2 \mid l_1(s) = l_2(s') \,\} \times \{\, 1, 2 \,\}$
- $S^0 = ((S_1^0 \times S_2^0) \times \{\, 1 \,\}) \cap S$
- if $s_1 \longrightarrow_1 s_1'$ and $s_2 \longrightarrow_2 s_2'$ then

  (i) if $s_1 \in F_1$ then $(s_1, s_2, 1) \longrightarrow (s_1', s_2', 2)$

  (ii) if $s_2 \in F_2$ then $(s_1, s_2, 2) \longrightarrow (s_1', s_2', 1)$

(iii) in all other cases $(s_1, s_2, i) \longrightarrow (s'_1, s'_2, i)$ for $i{=}1, 2$

- $F = ((F_1 \times S_2) \times \{ 1 \}) \cap S$
- $l(s, s', i) = l_1(s)$ (which equals $l_2(s')$).

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 accepting state, the pointer moves to another track (rules (i) and (ii)). More precisely, if it goes through an accepting 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 accepting states infinitely often, since a run is accepted iff it goes through $F_1 \times S_2 \times \{ 1 \}$ infinitely often. This means that the first track visits infinitely often an accepting state with the pointer pointing to the first track. Whenever the first track visits an accepting 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 accepting state (of A_2). Thus in order to visit an accepting state of $A_1 \otimes A_2$, both A_1 and A_2 have to visit an accepting state infinitely often and we have

$$\mathcal{L}_\omega(A_1 \otimes A_2) = \mathcal{L}_\omega(A_1) \cap \mathcal{L}_\omega(A_2).$$

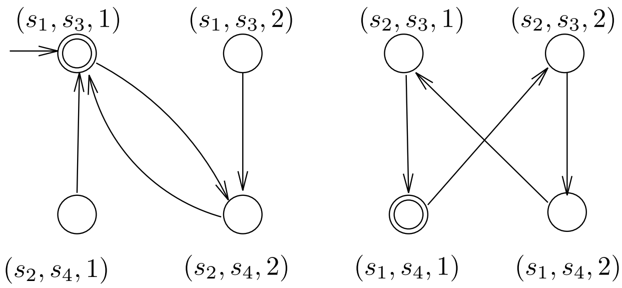
Example 18. Consider the Büchi automata A_1 and A_2 from the previous example. The LBA $A_1 \otimes A_2$ is constructed as follows. The set of states equals

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

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

$$\begin{array}{llll}
(s_1, s_3, 1) \longrightarrow (s_2, s_4, 2) & \textit{since} & s_1 \longrightarrow_1 s_2 \textit{ and } s_3 \longrightarrow_2 s_4 \textit{ and } s_1 \in F_1 \\
(s_1, s_4, 2) \longrightarrow (s_2, s_3, 1) & \textit{since} & s_1 \longrightarrow_1 s_2 \textit{ and } s_4 \longrightarrow_2 s_3 \textit{ and } s_4 \in F_2 \\
(s_1, s_3, 2) \longrightarrow (s_2, s_4, 1) & \textit{since} & s_1 \longrightarrow_1 s_2 \textit{ and } s_3 \longrightarrow_2 s_4 \textit{ and } s_3 \notin F_2
\end{array}$$

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 states are labeled with a. Clearly, the 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.)*

Notice that the number of states of the automaton $A_1 \otimes A_2$ is proportional to the product of the number of states in A_1 and A_2. This is a worst-case indication. Usually this upper-bound is not reached in model checking since $A_{sys} \otimes A_{\neg\phi}$ is a small automaton. This is due to the fact that only sequences are accepted by $A_{sys} \otimes A_{\neg\phi}$ that violate the desired property ϕ. One expects only few of these, none if the system is correct.

The emptiness problem

The second subproblem (cf. Table 2.5) is: given an LBA A how does one determine whether A is *empty*, i.e. whether $\mathcal{L}_\omega(A) = \varnothing$. An LBA A is non-empty iff it has an accepting state reachable from some initial state which is — in addition — 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 accepting state. 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 $s_i \longrightarrow s_{i+1}$ for $0 \leqslant i < k$.

Theorem 17.
Let $A = (\Sigma, S, S^0, \rho, F, l)$ be an LBA. $\mathcal{L}_\omega(A) \neq \varnothing$ iff there exists $s_0 \in S^0$ 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 accepting state. In the other direction the argument is only slightly more involved. Suppose there are states $s_0 \in S^0$ 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 $s_0\, s_1 \ldots s_k$, $k \geqslant 0$, and a sequence of symbols $a_0 \ldots a_k$ such that $s_k = s'$ (the sequence ends in s'), $s_i \longrightarrow s_{i+1}$ $(0 \leqslant i < k)$ and $a_i \in l(s_i)$ 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 $b_0 \ldots b_n$ such that $s'_0 = s'_n = s'$, $n > 0$, and $s'_i \longrightarrow s'_{i+1}$ $(0 \leqslant i < n)$ and $b_i \in l(s'_i)$ with $a_k = b_0$. 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, in order

To determine whether a given A is non-empty, we check whether A has a reachable cycle that contains an accepting state.

(Such a reachable cycle is also called a — non-trivial, since it must contain at least one edge — maximal strong component of A.) The algorithm that checks whether A has a reachable cycle that contains an accepting state consists of two steps. For convenience, assume that A has a single initial state s_0, i.e. $S^0 = \{ s_0 \}$.

1. The first part computes all accepting states that are reachable from the initial state. For convenience the accepting states are ordered. This is performed by the function *ReachAccept* which is listed in Table 2.6.

function $ReachAccept\,(s_0 : Vertex)$: **sequence of** $Vertex$;
(* precondition: true *)
begin var S : **sequence of** $Vertex$, (* path from s_0 to current vertex *)
    R : **sequence of** $Vertex$, (* reachable accepting states *)
    Z : **set of** $Vertex$; (* visited vertices *)
  $S, R, Z := \langle s_0 \rangle, \langle\, \rangle, \varnothing$;
  do $S \neq \langle\, \rangle \longrightarrow$ (* let $S = \langle s \rangle ^\frown S'$ *)
    if $\rho(s) \subseteq Z \longrightarrow S := S'$;
        if $s \in F \longrightarrow R := R^\frown \langle s \rangle$
        [] $s \notin F \longrightarrow$ **skip**
        fi
    [] $\rho(s) \not\subseteq Z \longrightarrow$ **let** s' **in** $\rho(s) \setminus Z$;
        $S, Z := \langle s' \rangle ^\frown S, Z \cup \{ s' \}$
    fi;
  od;
  return R;
(* postcondition: R contains accepting states reachable from s_0 *)
(* ordered such that if $R = \langle s_1, \ldots, s_k \rangle$ then *)
(* $i < j$ implies that s_i is explored before s_j *)
end

Table 2.6: Algorithm for determining reachable accepting states

2. The second part checks whether an accepting state computed by function *ReachAccept* is a member of a cycle. This is performed by the function *DetectCycle* which is listed in Table 2.7.

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 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. $\rho(s) \subseteq Z$), s is removed from S, and if s is accepting it is appended to R. 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 S are at the front of S; S is usually implemented as a stack.

Example 19. Consider the LBA depicted in Figure 2.7. Assume that the order of entering the sequence S equals $ADEFBCG$, that is, first state A is visited, then state D, then E, and so on. This is the order of visiting. The order of exploration is $FEDGCBA$. This is the order in which the search is finished in a state. Thus the successor states of F are all visited first, then those of E, and so on. Given that C, E and F are accepting states, the sequence R will contain FEC, the order of exploration projected onto the accepting states. (End of

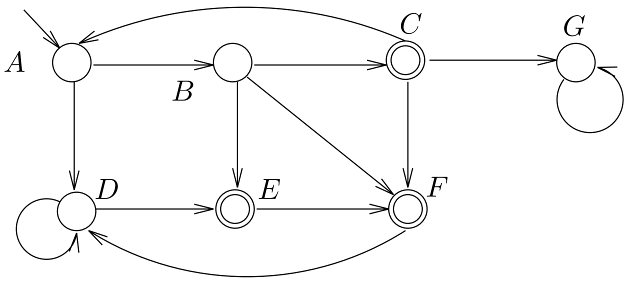


Figure 2.7: Example LBA

example.)

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

Rather than computing *ReachAccept* and *DetectCycle* as two separate phases, it is possible to check whether an accepting state is a member of a cycle on-the-fly, that is while determining all accepting states reachable from s_0. We thus obtain a *nested depth-first search*: in the outermost search an accepting state is identified which is reachable from s_0, whereas in the innermost search it is determined whether such a state is a member of a cycle. The resulting program is shown in Table 2.8.

function *DetectCycle*(R : **sequence of** *Vertex*) : *Bool*;
(* precondition: true *)
begin var S : **sequence of** *Vertex*,
$\quad Z$: **set of** *Vertex*,
$\quad b$: *Bool*;
$\quad S, Z, b := \langle\,\rangle, \varnothing, \text{false};$
$\quad$ **do** $R \neq \langle\,\rangle \;\wedge\; \neg b \longrightarrow$ (* let $R = \langle s\rangle^\frown R'$ *)
$\qquad R, S := R', \langle s\rangle^\frown S;$
$\qquad$ **do** $S \neq \langle\,\rangle \;\wedge\; \neg b \longrightarrow$ (* let $S = \langle s'\rangle^\frown S'$ *)
$\qquad\quad b := (s \in \rho(s'));$
$\qquad\quad$ **if** $\rho(s') \subseteq Z \longrightarrow S := S'$
$\qquad\quad$ $[]\ \rho(s') \not\subseteq Z \longrightarrow$ **let** s'' **in** $\rho(s') \setminus Z;$
$\qquad\qquad\qquad\qquad S, Z := \langle s''\rangle^\frown S, Z \cup \{\, s''\,\}$
$\qquad\quad$ **fi**;
$\qquad$ **od**;
$\quad$ **od**;
$\quad$ **return** b;
(* postcondition: b is true if some state in R is member of a cycle *)
(* b is false if there is no such state in R *)
end

Table 2.7: Algorithm for determining the existence of an acceptance cycle

```
function ReachAcceptandDetectCycle (s₀ : Vertex) : Bool;
begin var S₁, S₁ : sequence of Vertex,
          Z₁, Z₂ : set of Vertex,
          b : Bool;
          S₁, S₂, Z₁, Z₂, b := ⟨s₀⟩, ⟨ ⟩, ∅, ∅, false;
      do S₁ ≠ ⟨ ⟩ ∧ ¬b ⟶ (* let S₁ = ⟨s⟩⌢S₁′ *)
          if ρ(s) ⊆ Z₁ ⟶
             S₁ := S₁′;
             if s ∉ F ⟶ skip
             [] s ∈ F ⟶ S₂ := ⟨s⟩⌢S₂;
                         do S₂ ≠ ⟨ ⟩ ∧ ¬b ⟶ (* let S₂ = ⟨s′⟩⌢S₂′ *)
                             b := (s ∈ ρ(s′));
                             if ρ(s′) ⊆ Z₂ ⟶ S₂ := S₂′
                             [] ρ(s′) ⊈ Z₂ ⟶ let s″ in ρ(s′) \ Z₂;
                                                  S₂, Z₂ := ⟨s″⟩⌢S₂′, Z₂ ∪ { s″ }
                             fi;
                         od;
             fi;
          [] ρ(s) ⊈ Z₁ ⟶ let s″ in ρ(s) \ Z₁;
                            S₁, Z₁ := ⟨s″⟩⌢S₁′, Z₁ ∪ { s″ }
          fi;
      od;
      return b;
end
```

Table 2.8: Integrated program for *ReachAccept* and *DetectCycle*

Now assume we apply this algorithm to the product automaton $A_{sys} \otimes A_{\neg\,\phi}$. An interesting aspect of this program is that if an error occurs, that is, if a cycle is determined in $A_{sys} \otimes A_{\neg\,\phi}$ that contains an accepting state s, then an example of an incorrect path can be computed easily: sequence S_1 contains the path from the starting state s_0 to s (in reversed order), while sequence S_2 contains the cycle from s to s (in reversed order). In this way, a counterexample that shows how ϕ is violated can be produced easily, namely $(S_1' {}^\frown S_2)^{-1}$ where $S_1 = \langle s \rangle {}^\frown S_1'$.

The time complexity of the nested depth-first search algorithm of Table 2.8 is proportional to the number of states plus the number of transitions in the LBA under consideration, i.e. $\mathcal{O}(\mid S \mid + \mid \rho \mid)$. Recall that we assumed that the LBA only contains a single initial state. If there are more initial states, the whole procedure should be carried out for each initial state. This yields a worst-case time complexity of

$$\mathcal{O}(\mid S^0 \mid \times (\mid S \mid + \mid \rho \mid)).$$

2.10 Summary of steps in PLTL-model checking

An overview of the different steps of model checking PLTL is shown in Figure 2.8. 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.

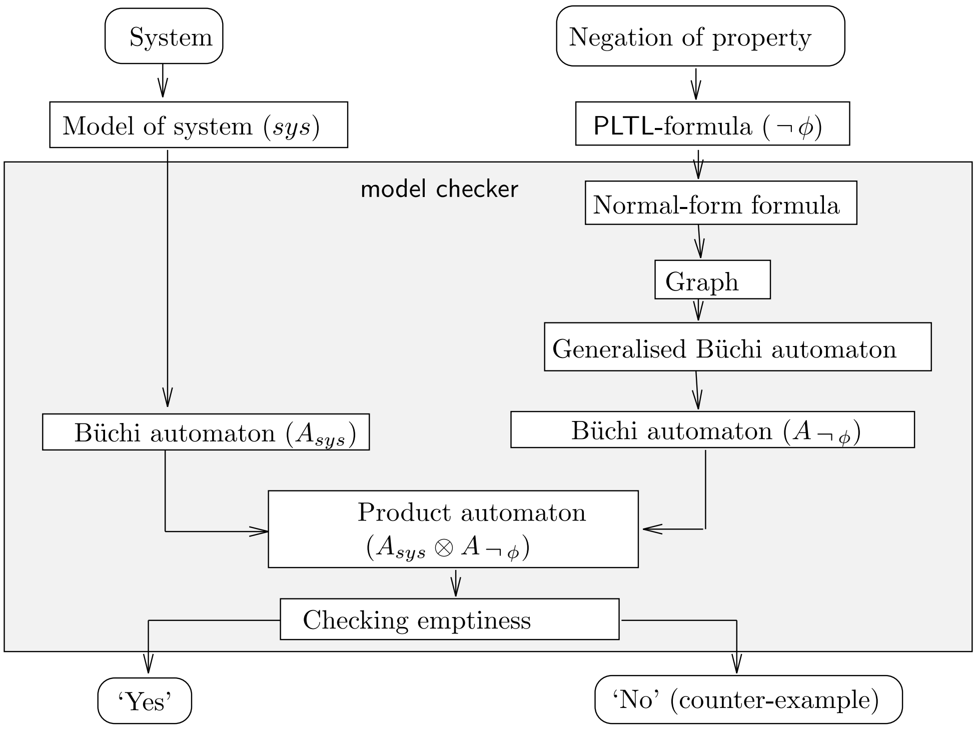


Figure 2.8: Overview of model-checking PLTL

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 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 LBA do not affect this worst-case complexity, the resulting LBA 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 $\mathcal{O}(|\ S_{sys}\ | \times 2^{|\phi|})$ where S_{sys} denotes the set of states in the LBA A_{sys}. Since the time complexity of checking the emptiness of an LBA 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* $\mathcal{O}(|\ S_{sys}\ |^2 \times 2^{|\phi|})$

since in worst case we have $|\ S_{sys}\ |^2$ 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).

2.11 The model checker Spin

Spin (see `http://netlib.bell-labs.com/netlib/spin/whatispin.html`) allows the simulation of a specification written in the language Promela (PROtocol MEta-LAnguage) and the verification of several types of properties, such as model checking of PLTL-formulas (without the next operator X) and verification of state properties, unreachable code, and so on. An overview of the Spin-tool is given in Figure 2.9. The algorithms that are used by SPIN in order to perform model checking of PLTL-formulas are basically the algorithms that we have treated in this chapter. For instance, the conversion of a PLTL-formula into a Büchi automaton is performed using the 5 steps that we have described before[3], and the detection of emptiness is done by a nested-depth first search procedure. It is beyond the scope of these lecture notes to give an extensive description of

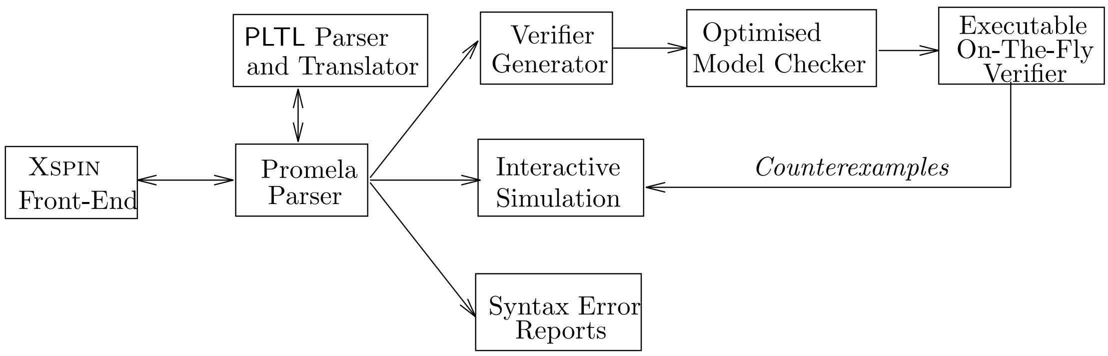


Figure 2.9: The structure of SPIN simulation and verification

SPIN; there are several overview articles and an introductory book of Holzmann — the developer of SPIN — that cover this (Holzmann, 1991). Instead, we would like to give the reader some insight into how systems can be modeled in PROMELA and how an example verification and simulation is treated using SPIN. For that purpose we use a standard leader election protocol known from the literature. First we give a brief overview of the core of the language PROMELA.

A PROMELA specification consists roughly of sequential processes, (local and global) variables, and communication channels used to connect sequential processes. A process, say `P`, is defined as follows

```
proctype P (formal parameters) { local definitions; statements }
```

This construct only defines the process `P`, some variables and constants that are local to `P`, and its behavior, but it does not instantiate the process. Processes are

[3]Although there are some subtle differences (as pointed out to me by Rob Gerth), e.g. the generated automaton by SPIN — called never claim — does accept also finite runs.

started through the `run` construct that, for instance, appears in the initialization section indicated by `init`. Here, an instance of the processes `P` and `Q` are created and started.

```
init{ run P(actual parameters); run Q(actual parameters); ...... }
```

Statements in PROMELA can be either enabled or blocked. If a statement is blocked, execution of the statement halts at this point until the statement becomes enabled. So, for instance, the statement `(a == b)` is equivalent to `while (a != b) do skip`. PROMELA is strongly influenced by Dijkstra's guarded command language that we have used as an abstract program notation. Statements are separated by semicolons. To summarize the following constructs exist in PROMELA:

- The empty statement (written `skip`). It means 'do nothing'.
- Assignment statement, for instance, `x = 7` means that the variable `x` of type `byte` is assigned the value 7.
- If-statement. An if-statement has the form

  ```
  if :: guard1 -> statements1
     ::   ......
     :: guardn -> statementsn
  fi
  ```

  One of the alternatives whose guard is enabled is selected and the corresponding (non-empty) list of statements is executed. Non-determinism is allowed: if more than one guard is enabled, the selection is made non-deterministically. A special guard `else` is enabled when all other guards are blocked. If there is no else-guard and all other guards are blocked, the if-statement itself is blocked, and the execution halts until one of the guards becomes enabled.
- Do-statement: has the same form as an if-statement, with the only difference that `if` is replaced by `do` and `fi` by `od`. Notice that a do-statement is

in principal always an infinite loop, since if all guards are blocked, it waits until one of the guards becomes enabled. A do-statement can therefore only be terminated by the use of `goto` or `break`.

- Send and receive statements. Processes can be interconnected via unidirectional channels of arbitrary (but finite) capacity. Channels of capacity 0 are also allowed. Channels are specified in the same way as ordinary variables; e.g. `chan c = [5] of byte` defines a channel named `c` of capacity 5 (bytes). A channel can be viewed as a first-in first-out buffer. Sending a message is indicated by an exclamation mark, reception by a question mark. Suppose `c` has a positive capacity. Then `c!2` is enabled if `c` is not fully occupied. Its execution puts the element 2 at the end of the buffer. `c?x` is enabled if `c` is non-empty. Its effect is to remove the head of the buffer `c` and assign its value to the variable `x`. If `c` is unbuffered, i.e. it has size 0, then `c!` (and `c?`) is enabled if there is a corresponding `c?` (and `c!`, respectively) which can be executed simultaneously (and the types of the parameters match). This facility is useful for modeling synchronous communication between processes.

Other PROMELA constructs that we need for our small case study will be introduced on-the-fly in the sequel.

Dolev, Klawe and Rodeh's leader election protocol

It is assumed that there are N processes ($N \geqslant 2$) in a ring topology, connected by unbounded channels. A process can only send messages in a clockwise manner. Initially, each process has a unique identifier *ident*, below assumed to be a natural number. The purpose of a leader election protocol is to make sure that exactly one process will become the leader. In addition, it is the idea that the process with the highest identifier should be elected as a leader. Here the concept is that the identifier of a process represents its capabilities and it is desired that the process with the largest capabilities wins the election. (In real life this is also often desired, but usually the outcome of an election does not conform to the latter principle.) In the protocol of Dolev, Klawe and Rodeh (1982) each process in the ring performs the following task:

<u>*active:*</u>
$d := ident$;
do forever
begin
    $send(d)$;
    $receive(e)$;
    if $e = d$ **then stop**; (* process d is the leader *)
    $send(e)$;
    $receive(f)$;
    if $e \geqslant max(d, f)$ **then** $d := e$ **else goto** *relay*;
end

<u>*relay:*</u>
do forever
begin
  $receive(d)$;
  $send(d)$
end

The intuition behind the protocol is as follows. Initially each process is <u>*active*</u>. As long as a process is active it is responsible for a certain process number (kept in variable d). This value may change during the evolvement of the protocol. When a process determines that it does not keep the identity of a leader-in-spe, it becomes relaying. If a process is <u>relaying</u> it passes messages from left to right in a transparent way, that is, without inspecting nor modifying their contents.

Each active process sends its variable d to its clockwise neighbor, and then waits until it receives the value (e) of its nearest active anti-clockwise neighbor. If the process receives its own d, it concludes that it is the only active process left and that d is indeed the identity of the new leader, and terminates. (Notice that the process itself does not have to become the new leader!) In case a different value is received ($e \neq d$), then the process waits for the second message (f) that contains the value d kept by the second nearest active anti-clockwise neighbor. If the value of the nearest active anti-clockwise neighbor is the largest among e, f, and d, then the process updates its local value (i.e. $d := e$), otherwise it becomes relaying. Thus from every set of active neighbours one will become relaying in every round.

To illustrate how the protocol works, an example run of the protocol for 4 processes with identities 3, 4, 27 and 121, respectively, is shown in Figure 2.10. Here, processes are indicates by circles, and communication channels by arrows. The content of a channel is indicated by the label of the arrow, where the semi-colon symbol is used to separate distinct messages. Each circle is labelled by a triple indicating the current value of the local variables d, e and f. A black circle indicates that the process is relaying, otherwise it is active.

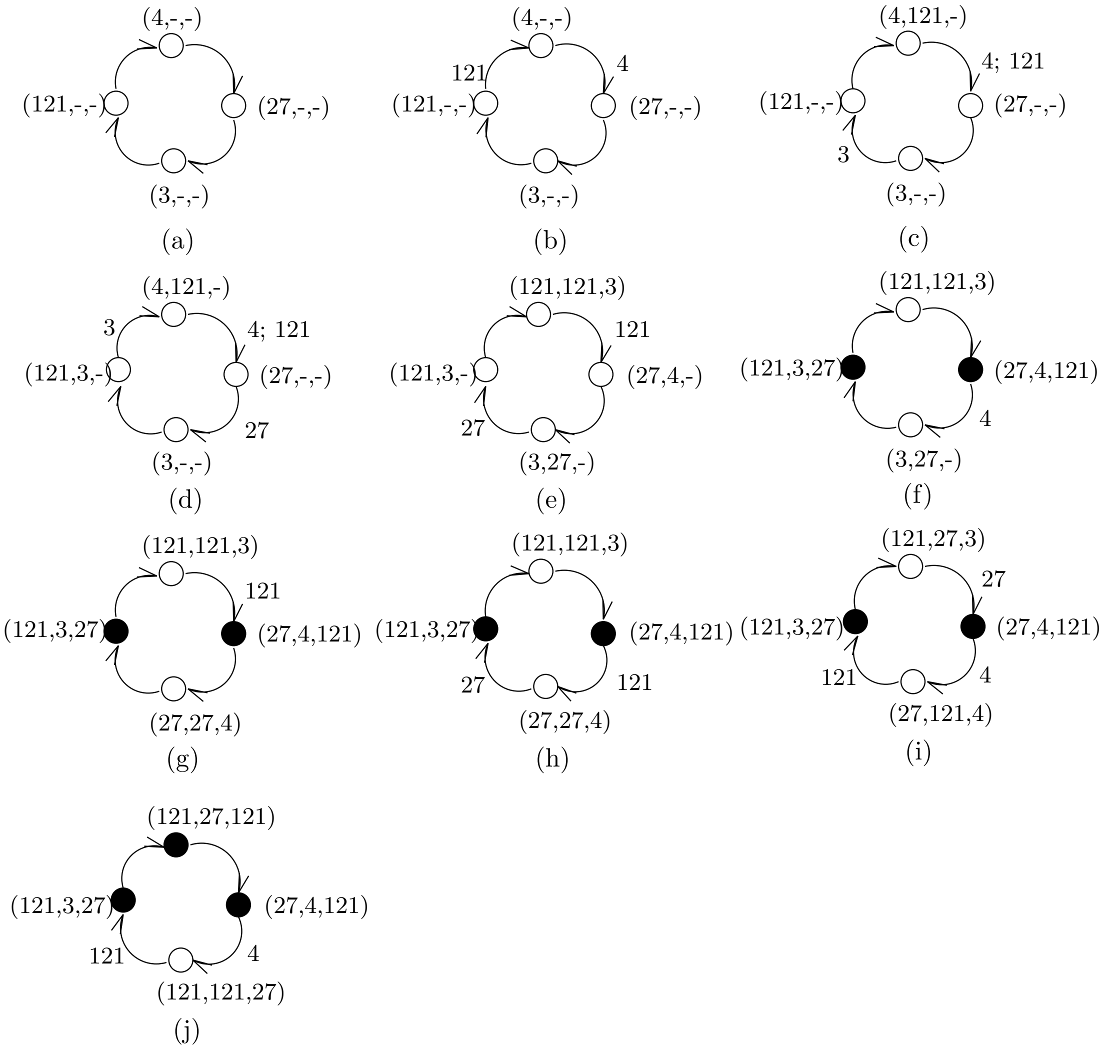


Figure 2.10: Example run of the leader election protocol

Modeling the leader election protocol in Promela

The Promela specification consists of three parts: definitions of global variables, description of the behaviour of a process in the ring, and an initialisation section where all processes are started. The behavioural part of a `process` is actually obtained from a straightforward translation of the protocol description given just above. We have added two print comments to facilitate simulation. A process is connected to its anti-clockwise neighbour via channel `in`, and via channel `out` to its clockwise neighbour.

```
proctype process (chan in, out; byte ident)
{ byte d, e, f;

  printf("MSC: %d\n", ident);

activ:
  d = ident;
  do :: true -> out!d;
                in?e;
                if :: (e == d) ->
                        printf("MSC: %d is LEADER\n", d);
                        goto stop  /* process d is leader */
                   :: else -> skip
                fi;
                out!e;
                in?f;
                if :: (e >= d) && (e >= f) -> d = e
                   :: else -> goto relay
                fi
  od;

relay:
end:
  do :: in?d -> out!d
  od;
```

```
stop:
  skip
}
```

A process can be in three states: `activ` (we do not use "active", since this is a reserved keyword in PROMELA), `relay`, and `stop`. A process can only reach the stop state if it has recognised a new leader. A process that does not recognise this leader will end in the relay state. In order to avoid the generation of an invalid endstate (a state that is neither the end of a program body nor labeled with `end:`) we label the relay state as an end-state. If one were to verify the above PROMELA specification without this endlabel, SPIN would find an invalid end-state and report this as an error.

Notice that, for instance, the statement `in?e` is enabled only if the channel `in` is non-empty. If the channel is empty, the statement is blocked and the execution of the process is suspended until the statement becomes enabled. Therefore `in?e` should be read as: wait until a message arrives in channel `in` and then store its value in the variable `e`.

The PROMELA specification starts with the definition of the global variables:

```
#define N       5               /* number of processes */
#define I       3               /* process with smallest identifier */
#define L       10              /* size of buffer (>= 2*N) */

chan q[N] = [L] of { byte };  /* N channels of capacity L each */
```

The constant `I` is used for initialisation purposes only; it indicates the process that will obtain the smallest `ident`. The capacity `L` of connecting channels is taken to be at least $2N$ since each process sends at most two messages in each round of the election. Since messages in the leader election protocol carry only a single parameter (a process identifier), the channels `q[0]` through `q[N-1]` can contain items of type `byte`.

In order to create the ring topology and to instantiate the processes in the ring, the `init` construct is used. This part of the PROMELA specification is, for instance, as follows:

```
init {
  byte i;
  atomic {
        i = 1;
        do :: i <= N -> run process (q[i-1], q[i%N], (N+I-i)%N+1);
                        i = i+1
           :: i > N -> break
        od
 }
}
```

where `%N` denotes the modulo N-operator. A process is started with some actual parameters by the `run process ()` statement. The first parameter of a process is the incoming channel, the second parameter is the outgoing channel, and the third parameter is its identity. There are, of course, different mechanisms to assign identifiers to processes, and we simply selected one. It is, however, assumed that each process gets a unique identifier. This is essential for the correctness of the algorithm of Dolev et. al. In order to allow all processes to start at the same time we use the `atomic` construct. This prevents some process starts to already execute its process body, as long as other processes have not yet been created.

Notice that we have fixed one particular assignment of process identities to processes. The results we will obtain by simulation and verification are valid only for this assignment. In order to increase confidence in the correctness of the protocol, other – in fact, all possible — assignments need to be checked. Another possibility is to write a PROMELA fragment that assigns process identifiers randomly.

The PROMELA specification ontained in this way can now be verified, and no unreachable code or deadlocks are obtained. In order to obtain insight into the operation of the protocol a (random, guided, or interactive) simulation can be carried out. This produces, for instance, the result shown in Figure 2.11.

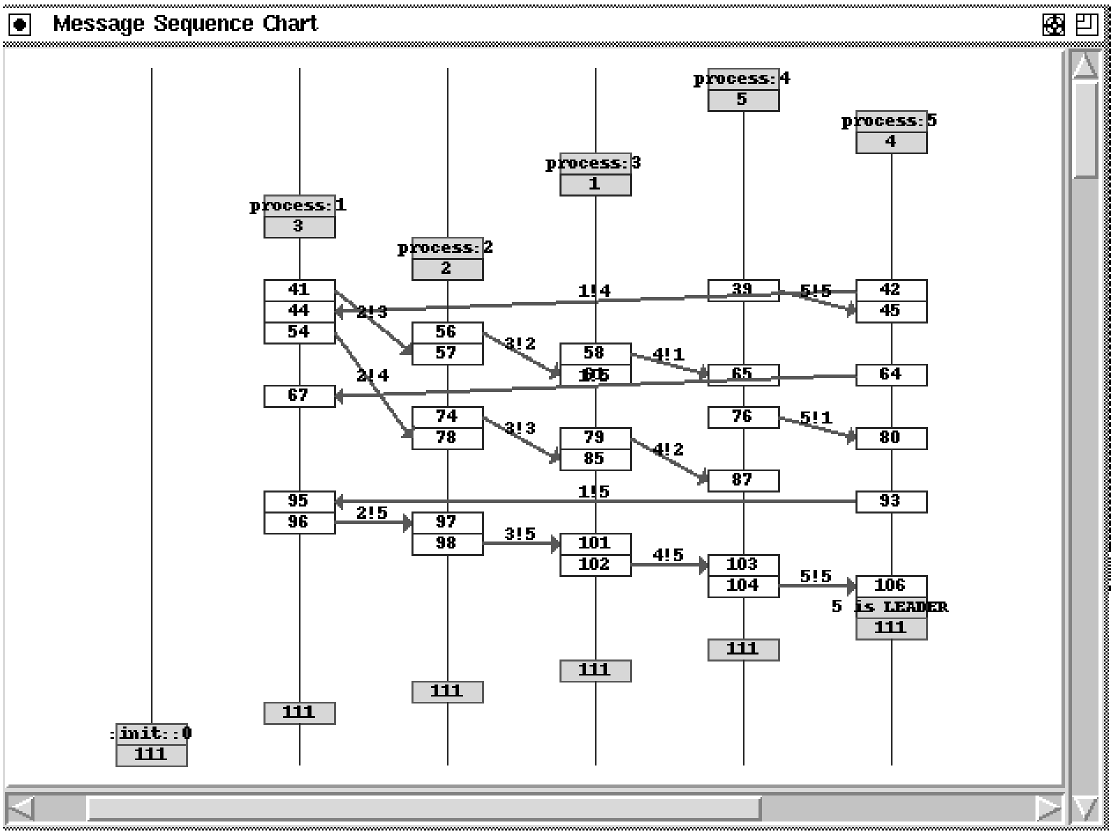


Figure 2.11: Visualisation of a simulation run in SPIN

Here, the behaviour of each process, including the root process (leftmost line), is represented along a vertical line.

Model checking the PROMELA specification

The most important property that a leader election protocol must possess is that it does not allow more than one process to become a leader. This is expressed in PLTL as:

$$\mathsf{G}\,\neg\,[\#leaders \geqslant 2],$$

or, equivalently,

$$\mathsf{G}\,[\#leaders \leqslant 1].$$

This property is checked using SPIN in the following way. One defines the formula as `[] p` where `[]` corresponds to G and `p` is defined as

```
#define p  (nr_leaders <= 1)
```

where `nr_leaders` is an auxiliary global variable in the PROMELA specification which keeps track of the number of leaders in the system. Therefore the PROMELA specification is extended in the following way.

```
byte nr_leaders = 0;

proctype process (chan in, out; byte ident)
{ .....as before.....

activ:
  d = ident;
  do :: true -> out!d;
                in?e;
                if :: (e == d) ->
                           printf("MSC: %d is LEADER\n", d);
                           nr_leaders = nr_leaders + 1;
                           goto stop  /* process d is leader */
                   :: else -> skip
                fi;

   .....as before.....
}
```

The property `[] p` is automatically converted by SPIN into a labelled Büchi automaton. In essence, this has been achieved by implementing the basic part of this chapter (Holzmann, 1997). One of the major differences is that SPIN labels transitions with atomic propositions, whereas we have labeled the states with (sets of) atomic propositions. The automaton generated by SPIN is as follows.

```
/*
* Formula As Typed: []  p
* The Never Claim Below Corresponds
* To The Negated Formula !([]  p)
* (formalizing violations of the original)
*/

never {    /* !([]  p) */
T0_init:
        if
        :: (1) -> goto T0_init
        :: (! ((p))) -> goto accept_all
        fi;
accept_all:
        skip
}
```

Notice that this is the Büchi automaton that corresponds to the negation of the formula to be checked; in our notation this would be $A_{\neg\phi}$, where ϕ corresponds to $\mathsf{G}\,[\#\mathit{leaders} \leqslant 1]$. If this property is saved in the file `property1`, then SPIN can automatically check it if we add at the end (or beginning) of our PROMELA specification the statement:

```
#include  "property1"
```

The result of the verification of this property is positive:

```
Full state space search for:
never-claim          +
assertion violations + (if within scope of claim)
```

```
cycle checks            - (disabled by -DSAFETY)
invalid endstates       - (disabled by never-claim)

State-vector 140 byte, depth reached 155, errors: 0
   16585 states, stored
   44589 states, matched
   61174 transitions (= stored+matched)
      18 atomic steps
hash conflicts: 1766 (resolved)
(max size 2^19 states)

Stats on memory usage (in Megabytes):
2.388  equivalent memory usage for states
                          (stored*(State-vector + overhead))
2.046  actual memory usage for states (compression: 85.65%)
State-vector as stored = 119 byte + 4 byte overhead
2.097  memory used for hash-table (-w19)
0.200  memory used for DFS stack (-m10000)
4.448  total actual memory usage
```

as indicated by the fact that `errors: 0`. Even for this small protocol with 5 processes (plus their interconnecting channels) the state space is already rather large. SPIN therefore provides various mechanisms in order to make the state space smaller, either through storing it in a compact way (using partial-order reduction) or by avoiding searching the entire state space (using hashing), or even a combination of both. These techniques are further discussed in Chapter 5 of these lecture notes. The effect of applying partial-order reduction to the above example is that only 3189 states and 5014 transitions are stored rather than 16585 states and 61172 transitions, the values if partial-order reduction is not applied. In the remainder of this example we will use partial-order reduction as a standard option.

In a similar way as above the following properties could be verified: always a leader will be elected eventually, the elected leader will be the process with the highest number, if a process is elected as a leader it remains a leader forever. The interested reader is encouraged to formulate these properties in PLTL and perform these verifications.

In order to see what happens when we check a property that is violated we check $\mathsf{G}\,[\#leaders = 1]$. Obviously, this property is violated in the state where all process instantiations are created, since then there is no leader. If we run SPIN we indeed obtain an error, and a counter-example is generated and stored in a file. In order to analyze the error, a guided simulation run can be carried out, based on the generated counter-example. The shortest trace that leads to the error found can be generated on request. This option can be indicated when the verification is started. Basically, SPIN then performs a breadth-first search rather than a depth-first search.

Verification using assertions

An alternative and often more efficient way of verification is the use of so-called *assertions*. An assertion has a boolean expression as argument. Its execution has no side-effects. If a PROMELA specification contains an assertion, SPIN checks whether there exists a run in which the assertion is violated. In such a case, an error is indicated. Since, in general, the state space is not increased when checking PLTL-properties (where the product automaton of the property and the specification is constructed) checking for assertions is usually more efficient. As an example of the use of assertions, suppose we want to check whether it is always the case that the number of leaders is at most one. This can be checked by adding the in-line assertion

```
assert(nr_leaders <= 1)
```

to our PROMELA specification at the point where a process determines that it has won the election and increments the variable `nr_leaders`. So we obtain as the body of a process:

```
byte nr_leaders = 0;

proctype process (chan in, out; byte ident)
```

```
{ .....as before.....

activ:
  d = ident;
  do :: true -> out!d;
                in?e;
                if :: (e == d) ->
                          printf("MSC: %d is LEADER\n", d);
                          nr_leaders = nr_leaders + 1;
                          assert(nr_leaders <= 1);
                          goto stop  /* process d is leader */
                   :: else -> skip
                fi;

   .....as before.....
}
```

Verification of this PROMELA specification yields no errors, and uses the same number of states and transitions as the verification of the corresponding PLTL-formula $\mathsf{G}\,[\#leaders \leqslant 1]$.

Exercises

EXERCISE 1. Let $\mathcal{M} = (S, R, Label)$ with $S = \{ s_0, s_1, s_2, \ldots, s_{10} \}$, $R(s_i) = s_{i+1}$ for $0 \leqslant i \leqslant 9$ and $R(s_{10}) = s_{10}$, and $Label(s_i) = \{ x = i \}$. Determine the set of states in S for which $(3 \leqslant x \leqslant 7)\,\mathsf{U}\,(x = 8)$ is valid. Do the same for $(0 \leqslant x \leqslant 2)\,\mathsf{U}\,((3 \leqslant x \leqslant 8)\,\mathsf{U}\,(x = 9))$.

EXERCISE 2.

1. Prove or disprove using the semantics of PLTL that $\mathsf{G}\,[\phi \;\Rightarrow\; \mathsf{X}\,\phi]$ equals $\mathsf{G}\,[\phi \;\Rightarrow\; \mathsf{G}\,\phi]$ where ϕ is a PLTL-formula. State in each step of your proof which result or definition you are using.

2. Let $\mathcal{M} = (S, R, \mathit{Label})$ be an PLTL-model, s a state in $\mathcal{M}$ and ϕ, ψ PLTL-formulas. Consider as the semantics of the operator Z

$$\begin{aligned}&\mathcal{M}, s \models \phi \,\mathsf{Z}\, \psi \text{ iff}\\ &\exists j \geqslant 0.\, \mathcal{M}, R^j(s) \models \psi \ \wedge \ [\forall\, 0 \leqslant k < j.\, \mathcal{M}, R^k(s) \models (\phi \ \wedge \ \neg\, \psi)]\end{aligned}$$

Prove or disprove the equivalence between $\phi \,\mathsf{Z}\, \psi$ and $\phi \,\mathsf{U}\, \psi$.

EXERCISE 3. Give a formal interpretation of $\underline{\mathsf{G}}$ and $\underline{\mathsf{F}}$ in terms of a satisfaction relation. Do the same with the past versions of next (previous) and until (since). (*Hint*: give the formal interpretation in terms of a model $(S, \sigma, \mathit{Label})$ where σ is an infinite sequence of states.)

EXERCISE 4. The binary operator B ("before") intuitively means: $\phi \,\mathsf{B}\, \psi$ is valid in state s if ϕ becomes true before ψ becomes true. Formalize the semantics of this operator, or define this operator in terms of other temporal operators like W.

EXERCISE 5. Prove or disprove the validity of the commutation rule

$$(\mathsf{F}\,\phi)\,\mathsf{U}\,(\mathsf{F}\,\psi) \ \Leftrightarrow \ \mathsf{F}\,(\phi\,\mathsf{U}\,\psi).$$

EXERCISE 6. Consider the specification of the dynamic leader election problem of Section 2.5. What can you say about the validity of the last but one property if we change $\mathsf{F} \neg\, \mathit{leader}_i$ into $\mathsf{X}\,\mathsf{F} \neg\, \mathit{leader}_i$?. Answer the same question for the case where we omit $\neg\, \mathit{leader}_j$ in the premise of that property knowing that we also require that there must always be at most one leader. What can you say about the validity of the last property if we change $\mathsf{X}\,\mathsf{F}\, \mathit{leader}_j$ into $\mathsf{F}\, \mathit{leader}_j$? Justify your answers. Finally, consider the following variant of the last property

$$\mathsf{G}\,[\forall\, i, j.\, (\mathit{leader}_i \ \wedge \ \mathsf{X}\,(\neg\, \mathit{leader}_i \,\mathsf{U}\, \mathit{leader}_j) \ \Rightarrow \ i < j]$$

Is this alternative formulation of the last informal property equivalent to the last property of Section 2.5?

EXERCISE 7.

1. Let $\Sigma = \{ a, b \}$. Construct a labelled Büchi automaton (LBA) A that accepts infinite words w over Σ such that a occurs infinitely many times in w and between any two successive a's an odd number of b's occur. Formally, $\mathcal{L}_\omega(A) = \{ (ab^{2n+1})^\omega \mid n \geqslant 0 \}$ where $\mathcal{L}_\omega(A)$ denotes the set of infinite words accepted by A.

2. Let $\Sigma = \{ a, b, c \}$. Construct an automaton as in the previous exercise, except that now there are an odd number of b's or an odd number of c's between any two successive a's. What is the language accepted by this automaton?

3. Let $\Sigma = \{ a, b, c \}$. Construct an automaton as the previous exercise except that now an odd number of b's or c's appear between any two successive a's. That is, *abcba* is now allowed as a finite sub-word. What is the language accepted by this automaton?

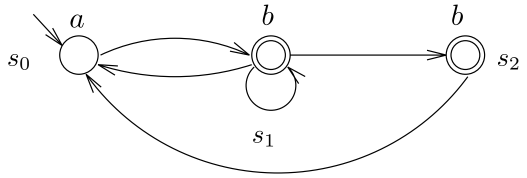
4. Construct an automaton as in exercise 2., with the extra condition that only a finite number of c's may occur in any accepting word. What is the language accepted by this automaton?

5. Let $\Sigma = \{ a, b, c \}$. Construct an LBA A that accepts infinite words w over Σ such that a occurs infinitely many times in w and an odd number of b's occur and an odd number of c's occur between any two successive a's. (I owe this variant to L. Kerber.)

Exercise 8. Construct the graph of the PLTL-formula $(q \vee \neg q) \, \mathsf{U} \, p$ using the *CreateGraph* algorithm. Subsequently, generate the corresponding GLBA and explain the result (i.e. make intuitively clear why the resulting GLBA is correct).

Exercise 9. Consider the GLBA: $S = \{ s_1, s_2 \}$ with $S^0 = \{ s_1 \}$, $l(s_1) = a$ and $l(s_2) = b$ such that $\rho(s_1) = \rho(s_2) = S$, and $F_1 = \{ s_1 \}$ and $F_2 = \{ s_2 \}$. Construct the corresponding LBA.

Exercise 10. Consider the following generalized LBA with acceptance sets $\mathcal{F} = \{ \{ s_2 \}, \{ s_3 \} \}$:



Transform this GLBA into an LBA and argue that both automata accept the same language.

EXERCISE 11. Define the following binary operator $\oplus$ on Büchi automata: for LBAs A_1 and A_2 let $A_1 \oplus A_2$ be a generalized LBA that after transformation into an LBA is equivalent to the product of A_1 and A_2. (I owe this exercise to L. Kerber.) (*Hint:* consider the transformation of a GLBA into an LBA.)

EXERCISE 12. Translate using SPIN the following linear-time temporal logic formulas into a Büchi automaton and explain the result: $\mathsf{G}\,[p \ \Rightarrow\ \mathsf{F}\,q]$ and $\mathsf{G}\,[p\,\mathsf{U}\,q]$, where p and q are atomic propositions. (*Remark:* note that SPIN labels transitions whereas in the lectures we labeled the states of an automaton instead.)

EXERCISE 13. With linear-time temporal logic formulas one can check properties of a PROMELA specification using SPIN. Alternatively, assertions can be used. Consider the following PROMELA specification:

```
bit X, Y;

proctype C()
 {
   do
   :: true -> X = 0; Y = X
   :: true -> X = 1; Y = X
   od
 }

proctype monitor()
 {
   assert(X==Y)
```

```
 }

init{ atomic{ run C(); run monitor() } }
```

Verify this PROMELA specification using SPIN. Explain the result. Modify the PROMELA specification such that the assertion `assert(X==Y)` is not violated. (I owe this exercise to D. Latella.)

EXERCISE 14. (Holzmann 1993) If two or more concurrent processes execute the same code and access the same data, there is a potential problem that they may overwrite each others results and corrupt the data. The mutual exclusion problem is the problem of restricting access to a *critical section* in the code to a single process at a time, assuming only the indivisibility of read and write instructions. The problem and a first solution were first published by Dijkstra (1965). The following "improved" solution appeared one year later by another author. It is reproduced here as it was published (in pseudo Algol).

boolean array $b(0;1)$ **integer** k, i, j;
comment process i, with i either 0 or 1 and $j = 1-i$
$C0$: $b(i)$:= false;
$C1$: **if** k != i **then begin**
$C2$: **if** $not(b(j))$ **then go to** $C2$;
  else $k := i$; **go to** $C1$ **end**;
  else critical section;
  $b(i)$:= true;
  remainder of program;
  go to $C0$;
  end

Questions:

(a) Model this mutual exclusion program in PROMELA.

(b) Disprove the correctness of the program using properties specified in linear-time temporal logic.

(c) Disprove the correctness of the program using assertions.

(d) Compare the state spaces of the verification runs of (b) and (c) and explain the difference in the size of the state space.

EXERCISE 15. We assume N processes in a ring topology, connected by unbounded queues. A process can only send messages in a clockwise manner. Initially, each process has a unique identifier *ident* (which is assumed to be a natural number). A process can be either active or relaying. Initially a process is active. In Peterson's leader election algorithm (1982) each process in the ring carries out the following task:

<u>*active:*</u>
$d := ident$;
do forever
begin
  /* start phase */
  $send(d)$;
  $receive(e)$;
  if $e = ident$ **then** announce elected;
  if $d > e$ **then** $send(d)$ **else** $send(e)$;
  $receive(f)$;
  if $f = ident$ **then** announce elected;
  if $e \geqslant max(d, f)$ **then** $d := e$ **else goto** *relay*;
end

<u>*relay:*</u>
do forever
begin
  $receive(d)$;
  if $d = ident$ **then** announce elected;
  $send(d)$
end

Model this leader election protocol in PROMELA (avoid invalid end-states!), and verify the following properties:

1. There is always at most one leader.
2. Always a leader will be elected eventually.

3. The elected leader will be the process with the highest number.

4. The maximum total amount of messages sent in order to elect a leader is at most $2N\lfloor \log_2 N \rfloor + N$. (*Hint:* choose an appropriate N such that $\log_2 N$ is easily computable.)

Check the above properties for two different assignments of process identities to processes.

2.12 Selected references

Linear temporal logic:

- A. PNUELI. The temporal logic of programs. *Proceedings 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.
- R. GOTZHEIN. Temporal logic and applications–a tutorial. *Computer Networks & ISDN Systems*, **24**: 203–218, 1992.
- Z. MANNA AND A. PNUELI. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

Past operators in temporal logic:

- O. LICHTENSTEIN, A. PNUELI, AND L. ZUCK. The glory of the past. *Logics of Programs*, LNCS 193, pages 196–218, 1985.

Büchi automata and the connection between PLTL and LBAs:

- M. VARDI. An automata-theoretic approach to linear temporal logic. *Logics for Concurrency – Structure versus Automata*, LNCS 1043, pages 238–265, 1995.

- P. Wolper, M. Vardi, A.P. Sistla. Reasoning about infinite computation paths. *Proceedings 24th IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1983.
- E.A. Emerson and C.-L. Lei. Modalities for model checking: branching time logic strikes back. *Proceedings 12th ACM Symposium on Principles of Programming Languages*, pages 84–96, 1985.
- R. Gerth, D. Peled, M. Vardi, P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. *Proceedings 13th Symposium on Protocol Specification, Testing and Verification*, pages 3–18, 1995.

Guarded command language:

- E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

Checking emptiness of Büchi automata:

- R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, **1**(2): 146–160, 1972.
- C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Journal of Formal Methods in System Design* **1**: 275–288, 1992.

Product automaton on Büchi automata:

- Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer and System Sciences* **8**: 117–141, 1974.

Complexity of PLTL:

- A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM* **32**(3): 733–749, 1985.

SPIN:

- G.J. HOLZMANN. The model checker SPIN. *IEEE Transactions on Software Engineering* **23**(5): 279–295, 1997.
- G.J. HOLZMANN. Design and validation of protocols: a tutorial. *Computer Networks & ISDN Systems* **25**: 981–1017, 1993.
- G.J. HOLZMANN. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
- R. GERTH. *Concise Promela reference*, 1997. (available on the web-site for SPIN.)

Leader election protocol:

- D. DOLEV, M. KLAWE, AND M. RODEH. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms* **3**: 245–260, 1982.

Chapter 3

Model Checking Branching Temporal Logic

Pnueli (1977) has introduced linear temporal logic to the computer science community for the specification and verification of reactive systems. In the previous chapter we have treated one important exponent of linear temporal logic, called PLTL. This temporal logic is called linear, since the (qualitative notion of) time is linear: at each moment of time there is only one possible successor state and thus only one possible future. Technically speaking, this follows from the fact that the interpretation of linear temporal logic-formulas using the satisfaction relation $\models$ is defined in terms of a model in which a state s has precisely one successor state $R(s)$. Thus for each state s the model thus generates a unique infinite sequence of states $s, R(s), R(R(s)), \ldots$. A sequence of states represents a computation. Since the semantics of linear temporal logic is based on such "sequence-generating" models, the temporal operators X, U, F and G in fact describe the ordering of events along a *single* time path, i.e. a single computation of a system.

In the early eighties another type of temporal logic for specification and verification purposes which is not based on a linear notion of time, but on a branching notion of time was introduced (Clarke and Emerson, 1980). This logic is formally based on models where at each moment there may be several different possible futures.[1] Due to this branching notion of time, this class of temporal logic is

[1]This does not imply that linear temporal logic cannot be interpreted over a branching

called *branching temporal logic*. Technically speaking, branching boils down to the fact that a state can have different possible successor states. Thus $R(s)$ is a (non-empty) set of states, rather than a single state as for PLTL. The underlying notion of the semantics of a branching temporal logic is thus 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. More precisely, the tree rooted at state s represents all possible infinite computations that start in s.

The temporal operators in branching temporal logic allow the expression of properties of (some or all) computations of a system. 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 is eventually reached that fulfills ϕ. This does, however, not exclude the fact that there can also be computations for which this property does not hold, for instance, computations for which ϕ is never fulfilled. The property $\mathsf{AF}\,\phi$, for instance, differs from this existential property over computations in that it requires that all computations satisfy the property $\mathsf{F}\,\phi$.

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, in our opinion, two main issues that justify the treatment of model checking linear and branching temporal logic in these lecture notes:

- 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 logic, and vice versa.

- The traditional *techniques* used for efficient model checking of linear tem-

model. It is possible to consider this logic over sequences of state, i.e. possible traversals through the branching model. Notice that in Chapter2 we took a different approach by directly defining linear temporal logic over sequences.

poral logic are quite different from those used for efficient model checking of branching temporal logic. (Although some promising unifying developments are currently taking place.) This results, for instance, in significantly different complexity results.

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. To mention a few important ones in increasing expressive power:

- Hennessy-Milner logic (1985)
- Unified System of Branching-Time Logic (Ben-Ari, Manna and Pnueli, 1983)
- Computation Tree Logic (Clarke and Emerson, 1980)
- Extended Computation Tree Logic (Clarke and Emerson, 1981) and
- Modal μ-Calculus (Kozen, 1983).

Modal μ-calculus is thus the most expressive among these languages, and Hennessy-Milner logic is the least expressive. The fact that the modal μ-calculus is the most expressive means that for any formula ϕ expressed in one of the other types of logic mentioned, an equivalent formula ψ in the modal μ-calculus can be given.

In this chapter we consider model checking of Computation Tree Logic (CTL). This is not only the temporal logic that was used originally by Clarke and Emerson (1981) and (in a slightly different form) by Quielle and Sifakis (1982) for model checking, but — more importantly — it can be considered as a branching counterpart of PLTL, the linear temporal logic that we considered in the second chapter, for which efficient model checking is possible.

3.1 Syntax of CTL

The syntax of computation tree logic is defined as follows. The most elementary expressions in CTL are atomic propositions, as in the definition of PLTL. The set of atomic propositions is denoted by AP with typical elements p, q, and r. We define the syntax of CTL in Backus-Naur Form:

Definition 18. (**Syntax of computation tree logic**)
For $p \in AP$ the set of CTL-formulas is defined by:

$$\boxed{\phi \quad ::= \quad p \;\Big|\; \neg\,\phi \;\Big|\; \phi \,\vee\, \phi \;\Big|\; \mathsf{EX}\,\phi \;\Big|\; \mathsf{E}\,[\phi\,\mathsf{U}\,\phi] \;\Big|\; \mathsf{A}\,[\phi\,\mathsf{U}\,\phi].}$$

Four temporal operators are used:

- EX (pronounced "for some path next")
- E (pronounced "for some path")
- A (pronounced "for all paths") and
- U (pronounced "until").

X and U are the linear temporal operators that express a property over a single path, whereas E expresses a property over soma path, and A expresses a property over all paths. The existential and universal path operators E and A can be used in combination with either X or U. Note that the operator AX is not elementary and is defined below.

The boolean operators true, false, $\wedge$, $\Rightarrow$ and $\Leftrightarrow$ are defined in the usual way (see the previous chapter). Given that $\mathsf{F}\,\phi = \text{true}\,\mathsf{U}\,\phi$ we define the following abbreviations:

$$\begin{array}{lcl} \mathsf{EF}\,\phi & \equiv & \mathsf{E}\,[\text{true}\,\mathsf{U}\,\phi] \\ \mathsf{AF}\,\phi & \equiv & \mathsf{A}\,[\text{true}\,\mathsf{U}\,\phi]. \end{array}$$

$\mathsf{EF}\,\phi$ is pronounced "ϕ holds potentially" and $\mathsf{AF}\,\phi$ is pronounced "ϕ is inevitable". Since $\mathsf{G}\,\phi = \neg\,\mathsf{F}\,\neg\,\phi$ and $\mathsf{A}\,\phi = \neg\,\mathsf{E}\,\neg\,\phi$ we have in addition: [2]

$$\begin{array}{lcl}
\mathsf{EG}\,\phi & \equiv & \neg\,\mathsf{AF}\,\neg\,\phi \\
\mathsf{AG}\,\phi & \equiv & \neg\,\mathsf{EF}\,\neg\,\phi \\
\mathsf{AX}\,\phi & \equiv & \neg\,\mathsf{EX}\,\neg\,\phi.
\end{array}$$

For instance, we have

$$\begin{array}{ll}
 & \neg\,\mathsf{A}\,(\mathsf{F}\,\neg\,\phi) \\
\Leftrightarrow & \{\ \mathsf{A}\,\psi \equiv \neg\,\mathsf{E}\,\neg\,\psi\ \} \\
 & \neg\,\neg\,\mathsf{E}\,\neg\,(\mathsf{F}\,\neg\,\phi) \\
\Leftrightarrow & \{\ \mathsf{G}\,\psi \equiv \neg\,\mathsf{F}\,\neg\,\phi;\ \text{calculus}\ \} \\
 & \mathsf{EG}\,\phi.
\end{array}$$

$\mathsf{EG}\,\phi$ 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 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 $\mathsf{AG}\,(p \Rightarrow \mathsf{EG}\,q)$.

Example 20. *Let $AP = \{\, x = 1, x < 2, x \geqslant 3 \,\}$ be the set of atomic propositions.*

- *Examples of* CTL*-formulas are:* $\mathsf{EX}\,(x = 1)$*,* $\mathsf{AX}\,(x = 1)$*,* $x < 2 \,\vee\, x = 1$*,* $\mathsf{E}\,[(x < 2)\,\mathsf{U}\,(x \geqslant 3)]$ *and* $\mathsf{AF}\,(x < 2)$*.*
- *The expression* $\mathsf{E}\,[x = 1 \,\wedge\, \mathsf{AX}\,(x \geqslant 3)]$ *is, for example, not a* CTL*-formula, since* $x = 1 \,\wedge\, \mathsf{AX}\,(x \geqslant 3)$ *is neither a next- nor an until-formula. The expression* $\mathsf{EF}\,[\mathsf{G}\,(x = 1)]$ *is also not a* CTL*-formula.* $\mathsf{EG}\,[x = 1 \,\wedge\, \mathsf{AX}\,(x \geqslant 3)]$ *is, however, a well-formed* CTL*-formula. Likewise* $\mathsf{EF}\,[\mathsf{EG}\,(x = 1)]$ *and* $\mathsf{EF}\,[\mathsf{AG}\,(x = 1)]$ *are well-formed* CTL*-formulas.*

[2] The equation $\mathsf{A}\,\phi = \neg\,\mathsf{E}\,\neg\,\phi$ only holds for ϕ where ϕ is a CTL-formula for which the outermost existential or universal path quantifier is "stripped off". It allows, for instance, rewriting $\mathsf{E}\,[\neg\,\mathsf{F}\,\neg\,\phi]$ into $\neg\,\mathsf{AF}\,\neg\,\phi$.

(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, then the more expressive branching temporal logic CTL* (Clarke and Emerson, 1981) is obtained. The logic CTL* permits an arbitrary PLTL-formula to be preceded by either E or A. It contains, for instance, $\mathsf{E}\,[p \;\wedge\; \mathsf{X}\,q]$ and $\mathsf{A}\,[\mathsf{F}\,p \;\wedge\; \mathsf{G}\,q]$, formulas which are not syntactical terms of 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 3.3. We do not consider model checking CTL* in these lecture notes, since model checking of CTL* is of intractable complexity: the model checking problem for this logic is PSPACE-complete in the size of the system specification (Clarke, Emerson and Sistla, 1986). We therefore consider CTL for which more efficient model checking algorithms do exist. Although CTL does not possess the full expressive power of CTL*, various (industrial) case studies have shown that it is frequently sufficiently powerful to express most required properties.

3.2 Semantics of CTL

As we have seen in the previous chapter, the interpretation of the linear temporal logic PLTL is defined in terms of a model $\mathcal{M} = (S, R, \mathit{Label})$ where S is a set of states, *Label* an assignment of atomic propositions to states, and R a total function that assigns a unique successor state to any given state. Since the successor of state s, $R(s)$, is unique, the model $\mathcal{M}$ generates for each state s a sequence of states $s, R(s), R(R(s)), \ldots$. These sequences represent computation paths starting at s, and since PLTL-formulas refer to a single path, the interpretation of PLTL is defined in terms of such sequences.

Branching temporal logic does, however, not refer to a single computation path, but to some (or all) possible computation paths. A single sequence is therefore insufficient to model this. In order to adequately represent the moments at which branching is possible, the notion of sequence is replaced by the notion of a *tree*. Accordingly a CTL-model is a "tree-generating" model. Formally, this is expressed as follows:

Definition 19. (**CTL-Model**)
A CTL-model is a triple $\mathcal{M} = (S, R, Label)$ where

- S is a non-empty set of states,
- $R \subseteq S \times S$ is a total relation on S, which relates to $s \in S$ its possible successor states,
- $Label : S \longrightarrow 2^{AP}$, assigns to each state $s \in S$ the atomic propositions $Label(s)$ that are valid in s.

$\mathcal{M}$ is also known as a *Kripke structure* [3], since Kripke used similar structures to provide a semantics for modal logic, a kind of logic that is closely related to temporal logic (Kripke, 1963).

Notice that the only difference to a PLTL-model is that R is now a total relation rather than a total function. A relation $R \subseteq S \times S$ is total if and only if it relates to each state $s \in S$ at least one successor state: $\forall\, s \in S.\, (\exists\, s' \in S.\, (s, s') \in R)$.

Example 21. Let $AP = \{\, x = 0, x = 1, x \neq 0 \,\}$ be a set of atomic propositions, $S = \{\, s_0, \ldots, s_3 \,\}$ be a set of states with labelling $Label(s_0) = \{\, x \neq 0 \,\}$, $Label(s_1) = Label(s_2) = \{\, x = 0 \,\}$, and $Label(s_3) = \{\, x = 1, x \neq 0 \,\}$, and transition relation R be given by

$$R = \{\, (s_0, s_1), (s_1, s_2), (s_1, s_3), (s_3, s_3), (s_2, s_3), (s_3, s_2) \,\}.$$

$\mathcal{M} = (S, R, Label)$ is a CTL*-model and is depicted in Figure 3.1(a). Here states are depicted by circles, 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 $Label(s)$ is indicated beside the state s. (End of example.)*

[3]Although usually a Kripke structure is required to have an identified set of initial state $S_0 \subseteq S$ with $S_0 \neq \varnothing$.

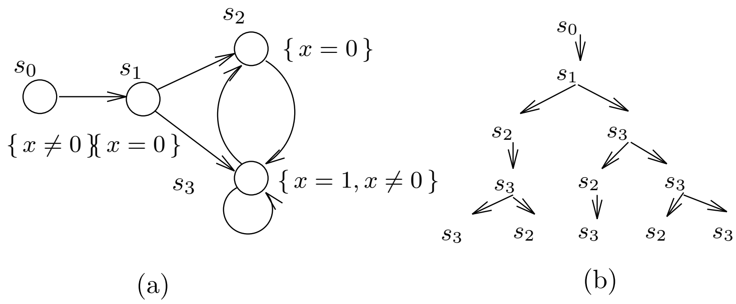


Figure 3.1: An example of a CTL-model and a (prefix of) of one of its infinite computation trees

Before presenting the semantics we introduce some auxiliary concepts. Let $\mathcal{M} = (S, R, \mathit{Label})$ be a CTL-model.

Definition 20. (**Path**)
A *path* is an infinite sequence of states $s_0\, s_1\, s_2 \ldots$ such that $(s_i, s_{i+1}) \in R$ for all $i \geqslant 0$.

Let $\sigma \in S^\omega$ denote a path (of states). For $i \geqslant 0$ we use $\sigma[i]$ to denote the $(i{+}1)$-th element of σ, i.e. if $\sigma = t_0\, t_1\, t_2 \ldots$ then $\sigma[i] = t_i$, where t_i is a state.

Definition 21. (**Set of paths starting in a state**)
The set of paths starting in state s of the model $\mathcal{M}$ is defined by

$$P_{\mathcal{M}}(s) = \{ \sigma \in S^\omega \mid \sigma[0] = s \}.$$

For any CTL-model $\mathcal{M} = (S, R, \mathit{Label})$ and state $s \in S$ there is an infinite computation tree with root labelled s such that (s', s'') is an arc in the tree if and only if $(s', s'') \in R$. A state s for which $p \in \mathit{Label}(s)$ is sometimes called a p-state. σ is called a p-path if it consists solely of p-states.

Example 22. Consider the CTL-*model of Figure 3.1(a). A finite prefix of the infinite computation tree rooted at state* s_0 *is depicted in Figure 3.1(b). Examples of paths are* $s_0\, s_1\, s_2\, s_3^\omega$, $s_0\, s_1\, (s_2\, s_3)^\omega$ *and* $s_0\, s_1\, (s_3\, s_2)^*\, s_3^\omega$. $P_{\mathcal{M}}(s_3)$, *for example, equals the set* $\{ (s_3\, s_2)^* s_3^\omega, (s_3^+\, s_2)^\omega \}$. *(End of example.)*

The semantics of CTL is defined by a satisfaction relation (denoted by $\models$) between a model $\mathcal{M}$, one of its states s, and a formula ϕ. As before, we write $\mathcal{M}, s \models \phi$ rather than $((\mathcal{M}, s), \phi) \in \models$. We have $(\mathcal{M}, s) \models \phi$ if and only if ϕ is valid in state s of model $\mathcal{M}$. As in the previous chapter, we omit $\mathcal{M}$ if the model is clear from the context.

Definition 22. (**Semantics of** CTL)
Let $p \in AP$ be an atomic proposition, $\mathcal{M} = (S, R, Label)$ be a CTL-model, $s \in S$, and ϕ, ψ be CTL-formulas. The satisfaction relation $\models$ is defined by:

$$
\begin{array}{ll}
s \models p & \text{iff } p \in Label(s) \\
s \models \neg\,\phi & \text{iff } \neg\,(s \models \phi) \\
s \models \phi \;\vee\; \psi & \text{iff } (s \models \phi) \;\vee\; (s \models \psi) \\
s \models \mathsf{EX}\,\phi & \text{iff } \exists\,\sigma \in P_{\mathcal{M}}(s).\,\sigma[1] \models \phi \\
s \models \mathsf{E}\,[\phi\,\mathsf{U}\,\psi] & \text{iff } \exists\,\sigma \in P_{\mathcal{M}}(s).(\exists\,j \geqslant 0.\,\sigma[j] \models \psi \;\wedge\; (\forall\,0 \leqslant k < j.\,\sigma[k] \models \phi)) \\
s \models \mathsf{A}\,[\phi\,\mathsf{U}\,\psi] & \text{iff } \forall\,\sigma \in P_{\mathcal{M}}(s).(\exists\,j \geqslant 0.\,\sigma[j] \models \psi \;\wedge\; (\forall\,0 \leqslant k < j.\,\sigma[k] \models \phi)).
\end{array}
$$

The interpretations for atomic propositions, negation and conjunction are as usual. $\mathsf{EX}\,\phi$ 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 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 at 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 the property $\phi\,\mathsf{U}\,\psi$.

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 above definition. To illustrate such a derivation we derive the formal semantics of $\mathsf{EG}\,\phi$.

$$
\begin{array}{ll}
 & s \models \mathsf{EG}\,\phi \\
\Leftrightarrow & \{\text{ definition of } \mathsf{EG} \} \\
 & s \models \neg\,\mathsf{AF}\,\neg\,\phi \\
\Leftrightarrow & \{\text{ definition of } \mathsf{AF} \} \\
 & s \models \neg\,\mathsf{A}\,[\text{true}\,\mathsf{U}\,\neg\,\phi] \\
\Leftrightarrow & \{\text{ semantics of } \neg \;\}
\end{array}
$$

$$
\begin{array}{ll}
& \neg\,(s \models \mathsf{A}\,[\mathrm{true}\,\mathsf{U}\,\neg\,\phi]) \\
\Leftrightarrow & \{\text{ semantics of } \mathsf{A}\,[\phi\,\mathsf{U}\,\psi] \text{ }\} \\
& \neg\,[\forall\,\sigma \in P_{\mathcal{M}}(s).\,(\exists\,j \geqslant 0.\,\sigma[j] \models \neg\,\phi \;\wedge\; (\forall\,0 \leqslant k < j.\,\sigma[k] \models \mathrm{true}))] \\
\Leftrightarrow & \{\; s \models \mathrm{true} \text{ for all states } s \;\} \\
& \neg\,[\forall\,\sigma \in P_{\mathcal{M}}(s).\,(\exists\,j \geqslant 0.\,\sigma[j] \models \neg\,\phi)] \\
\Leftrightarrow & \{\text{ semantics of } \neg\,;\text{ predicate calculus }\} \\
& \exists\,\sigma \in P_{\mathcal{M}}(s).\,\neg\,(\exists\,j \geqslant 0.\,\neg\,(\sigma[j] \models \phi)) \\
\Leftrightarrow & \{\text{ predicate calculus }\} \\
& \exists\,\sigma \in P_{\mathcal{M}}(s).\,(\forall\,j \geqslant 0.\,\sigma[j] \models \phi)\,.
\end{array}
$$

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 property 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 property ϕ holds. $\mathsf{EF}\,\phi$ is valid in state s if and only if ϕ holds eventually along some path that starts in s. $\mathsf{AF}\,\phi$ is valid iff 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.

Example 23. *Let the* CTL*-model* $\mathcal{M}$ *be given as depicted in Figure 3.2(a). In the figures below the validity of several formulas is indicated for all states of* $\mathcal{M}$*. A state is colored black if the formula is valid in that state, and otherwise colored white.*

- *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 (such as* $s_0\,s_1\,s_3^{\omega}$*) 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.*

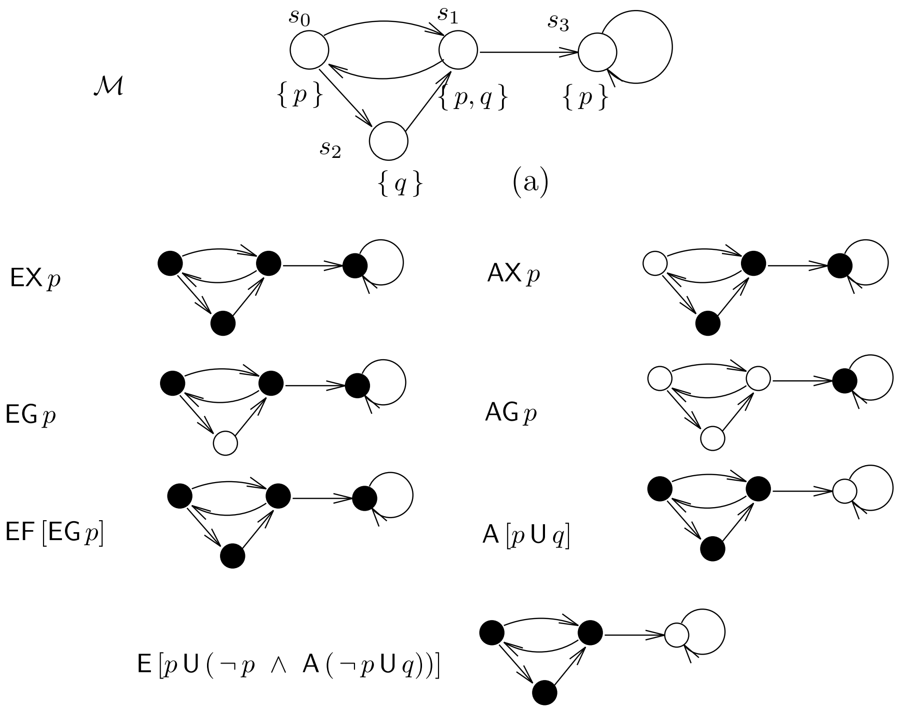


Figure 3.2: Interpretation of several CTL-formulas in an example of a model

- *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, which does not satisfy p. So, for these states AG p is not valid.*

- *EF [EG p] is valid for all states, since from each state a state (either s_0, s_1 or s_3) can be reached eventually from which some computation starts along which p is globally valid.*

- *A [p U q] is not valid in s_3 since its only computation (s_3^ω) never reaches a state for which q holds. For all other states this is, however true and in addition, the proposition p is valid before q is valid.*

- *Finally, E [p U ($\neg p$ $\wedge$ A ($\neg p$ 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 property 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 $\neg p$ 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 U ($\neg p$ $\wedge$ A ($\neg p$ U q)) since $p \in$ Label(s_0), $p \notin$ Label(s_2) and $q \in$ 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 next state is a q-state. Thus, the property $\neg\, p \ \wedge\ \mathsf{A}\,(\neg\, p\, \mathsf{U}\, q)$ is reached after a path of length 0.

(End of example.)

3.3 Expressiveness of CTL, CTL* and PLTL

In order to understand better the relationship between PLTL, CTL and CTL* we present an alternative characterization of the syntax of CTL and CTL* in terms of PLTL. We do this by explicitly distinguishing between state formulas, i.e. expressions over states, and path formulas, i.e. properties that are supposed to hold along paths. Table 3.1 summarizes the syntax of the three types of logic considered.[4] (Although we did not consider the formal semantics of CTL* in these lecture notes we assume that this interpretation is intuitively clear from the interpretation of CTL.)

Clearly, by inspecting the syntax it can be seen that CTL is a subset of CTL*: all CTL-formulas belong to CTL*, but the reverse does not hold (syntactically). Since CTL and CTL* are interpreted over branching models, whereas PLTL is interpreted over sequences, a comparison between these three logics is not straightforward. A comparison is facilitated by changing the definition of PLTL slightly, such that its semantics is also defined in terms of a branching model. Although there are different ways of interpreting PLTL over branching structures — for instance, should ϕ in case of $\mathsf{X}\,\phi$ hold for all or for some possible successor states of the current state? — the simplest and most common approach is to consider that a PLTL-formula ϕ holds for *all* paths. Since a PLTL-formula is usually implicitly universally quantified over all possible computations, this choice is well justified (Emerson and Halpern, 1986). This results in the following embedding of PLTL in terms of CTL*:

[4]Here we have taken an alternative way to define the syntax of CTL that facilitates the comparison. It is left to the reader to show that this alternative definition corresponds to Definition 18. Hereby, the reader should bear in mind that $\neg\,\mathsf{E}\,\neg\,\psi$ equals $\mathsf{A}\,\psi$ for path-formula ψ.

PLTL		$\phi ::= p \mid \neg\,\phi \mid \phi \;\vee\; \phi \mid \mathsf{X}\,\phi \mid \phi\,\mathsf{U}\,\phi$
CTL	state formulas	$\phi ::= p \mid \neg\,\phi \mid \phi \;\vee\; \phi \mid \mathsf{E}\,\psi$
	path formulas	$\psi ::= \;\neg\,\psi \mid \mathsf{X}\,\phi \mid \phi\,\mathsf{U}\,\phi$
CTL*	state formulas	$\phi ::= p \mid \neg\,\phi \mid \phi \;\vee\; \phi \mid \mathsf{E}\,\psi$
	path formulas	$\psi ::= \phi \mid \neg\,\psi \mid \psi \;\vee\; \psi \mid \mathsf{X}\,\psi \mid \psi\,\mathsf{U}\,\psi$

Table 3.1: Summary of syntax of PLTL, CTL and CTL*

Definition 23. (**Formulation of PLTL in terms of CTL***)
The state formulas of PLTL are defined by $\phi ::= \mathsf{A}\,\psi$ where ψ is a path formula. The path formulas are defined according to

$$\psi ::= p \mid \neg\,\psi \mid \psi \;\vee\; \psi \mid \mathsf{X}\,\psi \mid \psi\,\mathsf{U}\,\psi.$$

As a result, CTL, CTL* and PLTL are now interpreted in terms of the same model, and this common semantical basis allows a comparison. Let us first clarify how the expressiveness of two temporal logics is compared. Clearly, one logic is more expressive than another if it allows the expression of terms that cannot be expressed in the other. This syntactical criterion applies to CTL versus CTL*: the former is syntactically a subset of the latter. This criterion is in general too simple. More precisely, it does not exclude the fact that for some formula ϕ in one temporal logic $\mathcal{L}$, say, there does not exist an equivalent formula ψ — that syntactically differs from ϕ — in the temporal logic $\mathcal{L}'$. Here, by equivalent we mean:

Definition 24. (**Equivalence of formulas**)

Formulas ϕ and ψ are *equivalent* if and only if for all models $\mathcal{M}$ and states s:

$$\mathcal{M}, s \models \phi \text{ if and only if } \mathcal{M}, s \models \psi.$$

We are now in a position to define formally what it means for two temporal logics to be equally expressive.

Definition 25. (**Comparison of expressiveness**)
Temporal logic $\mathcal{L}$ and $\mathcal{L}'$ are *equally expressive* if for all models $\mathcal{M}$ and states s

$$\begin{array}{rl} & \forall\, \phi \in \mathcal{L}.\ (\exists\, \psi \in \mathcal{L}'.\, (\mathcal{M}, s \models \phi \;\Leftrightarrow\; \mathcal{M}, s \models \psi)) \\ \wedge & \forall\, \psi \in \mathcal{L}'.\ (\exists\, \phi \in \mathcal{L}.\, (\mathcal{M}, s \models \phi \;\Leftrightarrow\; \mathcal{M}, s \models \psi))\,. \end{array}$$

If only the first conjunct is valid, but not the second, then $\mathcal{L}$ is (strictly) *less expressive* than $\mathcal{L}'$.

Figure 3.3 depicts the relationship between the three logics considered in this

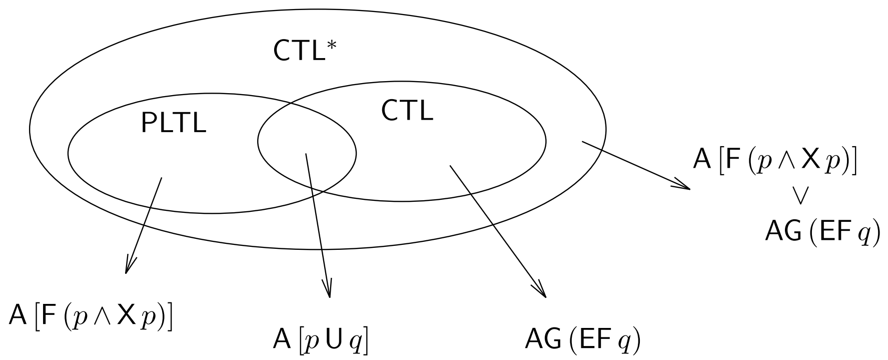


Figure 3.3: Relationship between PLTL, CTL and CTL^*

section. It follows that CTL^* is more expressive than both PLTL and CTL, whereas PLTL and CTL are incomparable. An example of a formula which distinguishes the part of the figure it belongs to is given for CTL, CTL^* and PLTL.

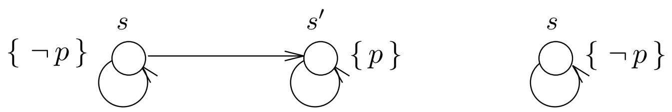
In PLTL, but not in CTL. For example, $\mathsf{A}\,[\mathsf{F}\,(p \wedge \mathsf{X}\,p)]$ is a PLTL-formula for which there does not exist an equivalent formula in CTL. The proof of this is by (Emerson & Halpern, 1986) and falls outside the scope of these lecture notes.

Another example of a PLTL-formula for which an equivalent formulation in CTL does not exist is:

$$\mathsf{A}\,[\mathsf{G}\,\mathsf{F}\,p \;\Rightarrow\; \mathsf{F}\,q]$$

which states that if p holds infinitely often, then q will be valid eventually. This is an interesting property which occurs frequently in proving the correctness of systems. For instance, a typical property for a communication protocol over an unreliable communication medium (such as a radio or infra-red connection) is that "if a message is being sent infinitely often, it will eventually arrive at the recipient".

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 that it is always possible to recover from a certain error. The proof sketch that for $\mathsf{AG}\,\mathsf{EF}\,p$ there does not exist an equivalent formulation in PLTL is as follows (Huth and Ryan, 1999). Let ϕ be a PLTL-formula such that $\mathsf{A}\,\phi$ is equivalent to $\mathsf{AG}\,\mathsf{EF}\,p$. Since $\mathcal{M}, s \models \mathsf{AG}\,\mathsf{EF}\,p$ in the left-hand figure below (a), it follows that $\mathcal{M}, s \models \mathsf{A}\,\phi$. Let $\mathcal{M}'$ be the sub-model of $\mathcal{M}$ shown in the right-hand diagram (b). The paths starting from s in $\mathcal{M}'$ are a subset of those starting from s in $\mathcal{M}$, so we have $\mathcal{M}', s \models \mathsf{A}\,\phi$. However, it is not the case that $\mathcal{M}', s \models \mathsf{AG}\,\mathsf{EF}\,p$, since p is never valid along the only path s^ω.



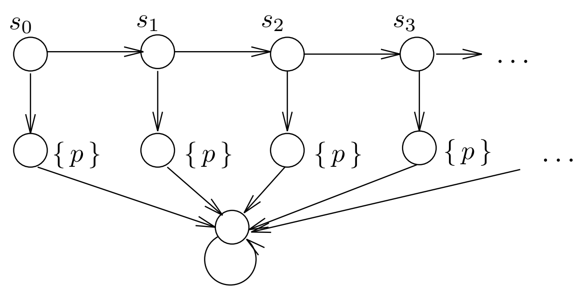
(a) (b)

Relationship between CTL* and PLTL. The following holds for the relationship between PLTL and CTL* (Clarke and Draghicescu, 1988):

> *For any* CTL^**-formula* ϕ*, an equivalent* PLTL*-formula (if such exists) must be of the form* $\mathsf{A}\, f(\phi)$ *where* $f(\phi)$ *equals* ϕ *where all path-quantifiers are eliminated.*

For instance, for $\phi = \mathsf{EF}\,\mathsf{EG}\,p \Rightarrow \mathsf{AF}\,q$ we obtain $f(\phi) = \mathsf{F}\,\mathsf{G}\,p \Rightarrow \mathsf{F}\,q$. Thus ϕ is a distinguishing formula for CTL^* if for ϕ its equivalent $\mathsf{A}\, f(\phi)$ is not a PLTL-formula.

As a final example of the difference in expressiveness of PLTL, CTL and CTL^* consider the PLTL-formula $\mathsf{G}\,\mathsf{F}\,p$, 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: $\mathsf{AG}\,\mathsf{F}\,p$ and $\mathsf{EG}\,\mathsf{F}\,p$ are CTL^*-formulas. $\mathsf{AG}\,\mathsf{F}\,p$ is equivalent to $\mathsf{AG}\,\mathsf{AF}\,p$ — for any model $\mathcal{M}$ the validity of these two formulas is identical — and thus for $\mathsf{AG}\,\mathsf{F}\,p$ an equivalent CTL-formula does exist, since $\mathsf{AG}\,\mathsf{AF}\,p$ is a CTL-formula. For $\mathsf{EG}\,\mathsf{F}\,p$, however, no equivalent CTL-formula does exist. This can be seen by considering the following example model.



We have $s \models \mathsf{EG}\,\mathsf{EF}\,p$, since for the path $s_0\, s_1\, s_2 \ldots$ for each s_j a p-state is eventually reachable. However, $s \not\models \mathsf{EG}\,\mathsf{F}\,p$, since there is no path starting in s such that p is infinitely often valid.

3.4 Specifying properties in CTL

In order to illustrate the way in which properties can be expressed in CTL we treat a simple two-process mutual exclusion program. Each process (P_1 and P_2) can be in one of the following three states: the critical section (C), the attempting section (T), and the non-critical section (N). A process starts in the non-critical section and indicates that it wants to enter its critical section by entering its attempting section. It remains in the attempting section until it obtains access to its critical section, and from its critical section it moves to the non-critical section. The state of process P_i is denoted $P_i.s$, for i=1, 2. Some required properties and their formal specification in CTL are as follows.

1. "It is not possible for both processes to be in their critical section at the same time".

$$\mathsf{AG}\,[\neg\,(P_1.s = C \;\wedge\; P_2.s = C)]$$

2. "A process that wants to enter its critical section is eventually able to do so".

$$\mathsf{AG}\,[P_1.s = T \;\Rightarrow\; \mathsf{AF}\,(P_1.s = C)]$$

3. "Processes must strictly alternate in having access to their critical section".

$$\mathsf{AG}\,[P_1.s = C \;\Rightarrow\; \mathsf{A}\,(P_1.s = C\,\mathsf{U}\,(P_1.s \neq C \;\wedge\; \mathsf{A}(P_1.s \neq C\,\mathsf{U}\,P_2.s = C)))]$$

3.5 Automaton-based approach for CTL?

The model checking problem for CTL is to check for a given CTL-model $\mathcal{M}$, state $s \in S$, and CTL-formula ϕ whether $\mathcal{M}, s \models \phi$, that is, to establish whether the property ϕ is valid in state s of model $\mathcal{M}$. Here, it is assumed that $\mathcal{M}$ is *finite.* This means that the set S of states is finite (but non-empty).

For linear temporal logic we have seen that the model checking algorithm is strongly based on Büchi automata, a kind of automata that accepts infinite words. The key result that makes this approach possible is the fact that for each PLTL-formula ϕ, a Büchi automaton can be constructed that accepts precisely those (infinite) sequences of atomic propositions that make ϕ valid. This allows the reduction of model checking PLTL to known automata-theoretic problems.

When dealing with CTL the question is whether a similar method can be used. The answer is positive, since an automaton-based approach is possible also for CTL. The difference between PLTL and CTL is that the former expresses properties related to infinite sequences of states, whereas the latter focuses on infinite *trees* of states. This suggests transforming a CTL-formula into a Büchi-like automaton that accepts infinite trees (of states) rather than infinite sequences. And, indeed, each CTL-formula can be transformed into a *Büchi-tree* automaton. The time complexity of this transformation is *exponential* in the length of the CTL-formula under consideration, as in the PLTL case.

Recent work of Bernholtz, Vardi and Wolper (1994) has led to a significant improvement of the automaton-based approach for branching temporal logic. They propose to use a variant of non-deterministic tree automata, called (weak) alternating tree automata. The time complexity of their approach is linear in the length of the CTL-formula and the size of the system specification. An interesting aspect of this approach is that the space complexity of their algorithm is NLOGSPACE in the size of the model $\mathcal{M}$. This means that model checking can be done in space which is polynomial in the size of the system specification (rather than exponential, as is usual). To our knowledge, no tools have been constructed yet based on alternating tree automata.

Although these current developments are interesting and quite promising, there does exist an efficient and well-established method for model checking CTL based on another paradigm that is conceptually simpler. As originally shown by Clarke and Emerson (1981) model checking for CTL can be performed in a time complexity that is linear in the size of the formula (and the system specification). In these lecture notes we discuss this traditional scheme that has its roots in *fixed point theory*. This enables us to use a tool, SMV, which is based on this approach.

3.6 Model checking CTL

Suppose we want to determine whether the CTL-formula ϕ is valid in the finite CTL-model $\mathcal{M} = (S, R, Label)$. The basic concept of the model checking algorithm is to "label" each state $s \in S$ with the sub-formulas of ϕ that are valid in s. The set of sub-formulas of ϕ is denoted by $Sub(\phi)$ and is inductively defined as follows.

Definition 26. (**Sub-formulas of a CTL-formula**)
Let $p \in AP$, and ϕ, ψ be CTL-formulas. Then

$$
\begin{array}{rcl}
Sub(p) & = & \{\, p \,\} \\
Sub(\neg\,\phi) & = & Sub(\phi) \cup \{\, \neg\,\phi \,\} \\
Sub(\phi \vee \psi) & = & Sub(\phi) \cup Sub(\psi) \cup \{\, \phi \vee \psi \,\} \\
Sub(\mathsf{EX}\,\phi) & = & Sub(\phi) \cup \{\, \mathsf{EX}\,\phi \,\} \\
Sub(\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]) & = & Sub(\phi) \cup Sub(\psi) \cup \{\, \mathsf{E}\,[\phi\,\mathsf{U}\,\psi] \,\} \\
Sub(\mathsf{A}\,[\phi\,\mathsf{U}\,\psi]) & = & Sub(\phi) \cup Sub(\psi) \cup \{\, \mathsf{A}\,[\phi\,\mathsf{U}\,\psi] \,\}.
\end{array}
$$

(Note that $\phi\,\mathsf{U}\,\psi$ is not a sub-formula of $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$.) The labelling procedure mentioned above is performed iteratively, starting by labelling the states with the sub-formulas of length 1 of ϕ, i.e. the atomic propositions (and true and false) that occur in ϕ. (Actually this step is easy, since the function *Label* already provides this labelling.) In the $(i{+}1)$-th iteration of the labelling algorithm sub-formulas of length $i{+}1$ are considered and the states are labelled accordingly. The labels already assigned to states are used for that purpose, being sub-formulas of ϕ of length at most i ($i \geqslant 1$). For instance, if $\phi = \phi_1 \vee \phi_2$ then state s is labelled with ϕ, a formula of length $i{+}1$, if s is already labelled with ϕ_1 or with ϕ_2 in some previous iteration. The labelling algorithm ends by considering the sub-formula of length $\mid \phi \mid$, ϕ itself.

The model checking problem for CTL, deciding whether $\mathcal{M}, s \models \phi$, for a given model $\mathcal{M}$ and CTL-formula ϕ, can now be solved for any state s in S by considering its labelling:

$\mathcal{M}, s \models \phi$ if and only if s is "labelled" with ϕ.

function *Sat* (ϕ : *Formula*) : **set of** *State*;
(* precondition: true *)
begin
    if $\phi = \text{true} \longrightarrow$ **return** S
    [] $\phi = \text{false} \longrightarrow$ **return** $\varnothing$
    [] $\phi \in AP \longrightarrow$ **return** $\{ s \mid \phi \in \mathit{Label}(s) \}$
    [] $\phi = \neg \phi_1 \longrightarrow$ **return** $S - \mathit{Sat}(\phi_1)$
    [] $\phi = \phi_1 \vee \phi_2 \longrightarrow$ **return** $(\mathit{Sat}(\phi_1) \cup \mathit{Sat}(\phi_2))$
    [] $\phi = \mathsf{EX}\, \phi_1 \longrightarrow$ **return** $\{ s \in S \mid \exists\, (s, s') \in R \ \wedge\ s' \in \mathit{Sat}(\phi_1) \}$
    [] $\phi = \mathsf{E}\, [\phi_1\, \mathsf{U}\, \phi_2] \longrightarrow$ **return** $\mathit{Sat}_{EU}(\phi_1, \phi_2)$
    [] $\phi = \mathsf{A}\, [\phi_1\, \mathsf{U}\, \phi_2] \longrightarrow$ **return** $\mathit{Sat}_{AU}(\phi_1, \phi_2)$
    fi
(* postcondition: $\mathit{Sat}(\phi) = \{ s \mid \mathcal{M}, s \models \phi \}$ *)
end

Table 3.2: Outline of main algorithm for model checking CTL

Actually, the model checking algorithm can be presented in a very compact and elegant way by determining for a given ϕ and $\mathcal{M}$:

$$\mathit{Sat}(\phi) = \{ s \in S \mid \mathcal{M}, s \models \phi \}$$

in an iterative way (as indicated above). By computing $\mathit{Sat}(\phi)$ according to the algorithm depicted in Table 3.2 the problem of checking $\mathcal{M}, s \models \phi$ reduces to checking $s \in \mathit{Sat}(\phi)$. (In the program text it is assumed that the model $\mathcal{M} = (S, R, \mathit{Label})$ is a global variable.)

Notice that by computing $\mathit{Sat}(\phi)$ a more general problem than just checking whether $\mathcal{M}, s \models \phi$ is solved. In fact, it checks for *any* state s in $\mathcal{M}$ whether $\mathcal{M}, s \models \phi$, and not just for a given one. In addition, since $\mathit{Sat}(\phi)$ is computed in an iterative way by considering the sub-formulas of ϕ, the sets $\mathit{Sat}(\psi)$ for any sub-formula ψ of ϕ are computed, and thus $\mathcal{M}, s \models \psi$ can be easily checked as well.

The computation of $\mathit{Sat}(\phi)$ is done by considering the syntactical structure of ϕ. For $\phi = \text{true}$ the program just returns S, the entire state space of $\mathcal{M}$, thus indicating that any state fulfills true. Accordingly, $Sat(\text{false}) = \varnothing$, since there is no state in which false is valid. For atomic propositions, the labelling $Label(s)$ 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. Finally, for $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$ and $\mathsf{A}\,[\phi\,\mathsf{U}\,\psi]$ the specific functions Sat_{EU} and Sat_{AU} are invoked that perform the computation of $Sat(\mathsf{E}\,[\phi\,\mathsf{U}\,\psi])$ and $Sat(\mathsf{A}\,[\phi\,\mathsf{U}\,\psi])$. These algorithms are slightly more involved, and their correctness is based on the computation of so-called *fixed points*. In order to understand these program fragments better, we first give a summary of the most important results and concepts of fixed point theory (based on partial orders[5]).

3.7 Fixed point theory based on posets

In this section we briefly recall some results and definitions from basic domain theory as far as they are needed to understand the fundamentals of model checking CTL.

Definition 27. (**Partial order**)
A binary relation $\sqsubseteq$ on set A is a *partial order* iff, 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,

[5]Another well-known variant of fixed-point theory is based on metric spaces — domains that are equipped with an appropriate notion of distance between any pair of its elements — where the existence of fixed points is guaranteed by Banach's contraction theorem for certain types of functions. This theory falls outside the scope of these lecture notes.

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 28. (**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. Let $\langle A, \sqsubseteq \rangle$ be a poset.

Definition 29. (**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 24. *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 upper 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 element of this example lattice are $\varnothing$ and S.* *(End of example.)*

Definition 30. (**Monotonic function**)
Function $F : A \longrightarrow A$ is *monotonic* if for each $a_1, a_2 \in A$ we have $a_1 \sqsubseteq a_2 \;\Rightarrow\; F(a_1) \sqsubseteq F(a_2)$.

Thus F is monotonic if it preserves the ordering $\sqsubseteq$. For instance, the function $F = S \cup \{ 0 \}$ is monotonic on $\langle 2^S, \subseteq \rangle$.

Definition 31. (**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. The following important result for complete lattices is from Knaster and Tarski (1955).

Theorem 32.
Every monotonic function over a complete lattice has a complete lattice of fixed points (and hence a unique greatest and unique least fixed point).

Notice that the lattice of fixed points is in general different from the lattice on which the monotonic function is defined.

The least fix-point of monotonic function F on the complete lattice $\langle A, \sqsubseteq \rangle$ can be computed by $\sqcup_i F^i(\bot)$, i.e. the least upper bound 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 roughly 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 $\sqcap_i F^i(\top)$, that is the greatest lower bound 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 following subsection we are interested in the construction of monotonic functions on lattices of CTL-formulas. 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 32). These fixed points can be easily computed and such computations form the key to the correctness of functions Sat_{EU} and Sat_{AU}.

3.8 Fixed point characterization of CTL-formulas

The labelling procedures for $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$ and $\mathsf{A}\,[\phi\,\mathsf{U}\,\psi]$ are based on a fixed point characterization of CTL-formulas. Here, the technique is to characterize $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$ as the least (or greatest) fixed point of a certain function (on CTL-formulas), and to apply an iterative algorithm — suggested by Knaster and Tarski's result — to compute such fixed points in order to carry out the labelling of the states. To do this basically two main issues need to be resolved.

1. First, a *complete lattice on* CTL*-formulas* 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-formulas.

2. Secondly, *monotonic functions on* CTL*-formulas* 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 it turns out that an axiomatization of CTL is useful.

In the sequel we start by defining a complete lattice of formulas, after which we introduce some axioms that are helpful in finding the monotonic functions mentioned in step 2. Finally, we show that $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$ and $\mathsf{A}\,[\phi\,\mathsf{U}\,\psi]$ are particular fixed points of these functions.

A complete lattice of CTL-formulas

The partial order $\sqsubseteq$ on CTL-formulas is defined by associating with each formula ϕ the set of states in $\mathcal{M}$ for which ϕ holds. Thus ϕ is identified with the set

$$[\![\,\phi\,]\!] = \{\, s \in S \mid \mathcal{M}, s \models \phi \,\}.$$

(Strictly speaking $[\![\ ]\!]$ is a function of $\mathcal{M}$ as well, i.e. $[\![\ ]\!]_{\mathcal{M}}$ would be a more correct notation, but since in all cases $\mathcal{M}$ is known from the context we omit this subscript.) The basic idea now is to define the order $\sqsubseteq$ by:

$$\phi \sqsubseteq \psi \text{ if and only if } [\![\phi]\!] \subseteq [\![\psi]\!].$$

Stated in words, $\sqsubseteq$ 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 $\langle \mathsf{CTL}, \sqsubseteq \rangle$ 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-formulas 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 $\langle \mathsf{CTL}, \sqsubseteq \rangle$ is false, since $[\![\text{false}]\!] = \varnothing$, which is the bottom element for 2^S. Similarly, true is the greatest element in $\langle \mathsf{CTL}, \sqsubseteq \rangle$ since $[\![\text{true}]\!] = S$.

Some CTL-axioms

In the previous chapter on linear temporal logic we have seen that axioms can be helpful in order to prove the equivalence between formulas: rather than proving the equivalence using the semantic definition, it is often sufficient to use the axioms that are defined on the syntax of the formulas. This facilitates proving the equivalence of the formulas. An important axiom for model checking PLTL (cf. Chapter 2) is the *expansion rule* for until:

$$\phi \,\mathsf{U}\, \psi \;\equiv\; \psi \;\vee\; (\phi \;\wedge\; \mathsf{X}\,[\phi \,\mathsf{U}\, \psi]).$$

By instantiating this rule with the definitions of F and G one obtains

$$\begin{aligned} \mathsf{G}\,\phi &\equiv \phi \;\wedge\; \mathsf{X}\,\mathsf{G}\,\phi \text{ and} \\ \mathsf{F}\,\phi &\equiv \phi \;\vee\; \mathsf{X}\,\mathsf{F}\,\phi. \end{aligned}$$

For CTL similar axioms do exist. Given that each linear temporal operator U, F, and G can be prefixed with either an existential or a universal quantification, we obtain the axioms as listed in Table 3.3. For all these axioms the basic idea 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 (statement about the successor states). The first

$\mathsf{EG}\,\phi$	$\equiv$	$\phi \;\wedge\; \mathsf{EX}\,[\mathsf{EG}\,\phi]$
$\mathsf{AG}\,\phi$	$\equiv$	$\phi \;\wedge\; \mathsf{AX}\,[\mathsf{AG}\,\phi]$
$\mathsf{EF}\,\phi$	$\equiv$	$\phi \;\vee\; \mathsf{EX}\,[\mathsf{EF}\,\phi]$
$\mathsf{AF}\,\phi$	$\equiv$	$\phi \;\vee\; \mathsf{AX}\,[\mathsf{AF}\,\phi]$
$\mathsf{E}\,[\phi \,\mathsf{U}\, \psi]$	$\equiv$	$\psi \;\vee\; (\phi \;\wedge\; \mathsf{EX}\,[\mathsf{E}\,(\phi \,\mathsf{U}\, \psi)])$
$\mathsf{A}\,[\phi \,\mathsf{U}\, \psi]$	$\equiv$	$\psi \;\vee\; (\phi \;\wedge\; \mathsf{AX}\,[\mathsf{A}\,(\phi \,\mathsf{U}\, \psi)])$

Table 3.3: Expansion axioms for CTL

four axioms can be derived from the latter two central ones. For instance, for $\mathsf{AF}\,\phi$ we derive

$$
\begin{array}{ll}
 & \mathsf{AF}\,\phi \\
\Leftrightarrow & \{ \text{ definition of } \mathsf{AF} \ \} \\
 & \mathsf{A}\,[\text{true}\,\mathsf{U}\,\phi] \\
\Leftrightarrow & \{ \text{ axiom for } \mathsf{A}\,[\phi\,\mathsf{U}\,\psi] \ \} \\
 & \phi \ \vee \ (\text{true} \ \wedge \ \mathsf{AX}\,[\mathsf{A}\,(\text{true}\,\mathsf{U}\,\phi)]) \\
\Leftrightarrow & \{ \text{ predicate calculus; definition of } \mathsf{AF} \ \} \\
 & \phi \ \vee \ \mathsf{AX}\,[\mathsf{AF}\,\phi].
\end{array}
$$

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

$$
\begin{array}{ll}
 & \mathsf{EG}\,\phi \\
\Leftrightarrow & \{ \text{ definition of } \mathsf{EG} \ \} \\
 & \neg\,\mathsf{AF}\,\neg\,\phi \\
\Leftrightarrow & \{ \text{ result of above derivation } \} \\
 & \neg\,(\,\neg\,\phi \ \vee \ \mathsf{AX}\,[\mathsf{AF}\,\neg\,\phi]) \\
\Leftrightarrow & \{ \text{ predicate calculus } \} \\
 & \phi \ \wedge \ \neg\,\mathsf{AX}\,[\mathsf{AF}\,\neg\,\phi] \\
\Leftrightarrow & \{ \text{ definition of } \mathsf{AX} \ \} \\
 & \phi \ \wedge \ \mathsf{EX}\,(\,\neg\,[\mathsf{AF}\,\neg\,\phi]) \\
\Leftrightarrow & \{ \text{ definition of } \mathsf{EG} \ \} \\
 & \phi \ \wedge \ \mathsf{EX}\,[\mathsf{EG}\,\phi].
\end{array}
$$

Similar derivations can be performed in order to find the axioms for EF and AG. These are left to the reader. As stated above, the elementary axioms are the last two expansion axioms. They can be proved using the semantics of CTL, and these proofs are very similar to the proof of the expansion law for U for PLTL as discussed in the previous chapter. We therefore omit these proofs here and leave them (again) to the reader.

CTL-formulas as fixed points

The expansion axiom

$$\mathsf{E}\,[\phi\,\mathsf{U}\,\psi] \equiv \psi \;\vee\; (\phi \;\wedge\; \mathsf{EX}\,[\mathsf{E}\,(\phi\,\mathsf{U}\,\psi)])$$

suggests considering the expression $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$ as a fixed point of the function $G : \mathsf{CTL} \longrightarrow \mathsf{CTL}$ defined by

$$G(z) = \psi \;\vee\; (\phi \;\wedge\; \mathsf{EX}\,z)$$

since clearly, one obtains $G(\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]) = \mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$ from the above expansion rule. 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 G (Huth and Ryan, 1999). More precisely, $[\![\, \mathsf{E}\,[\phi\,\mathsf{U}\,\psi]\,]\!]$ is a fixed point of the function $F : 2^S \longrightarrow 2^S$, where F is defined by

$$F(Z) = [\![\,\psi\,]\!] \;\cup\; ([\![\,\phi\,]\!] \;\cap\; \{\, s \in S \mid \exists\, s' \in R(s) \;\cap\; Z \,\}).$$

where $R(s)$ denotes the set of direct successors of s, that is, $R(s) = \{\, s' \in S \mid (s, s') \in R \,\}$. Similar formulations can be obtained for the other temporal operators. To summarize we obtain the following fixed point characterizations:

Theorem 33.

1. $[\![\,\mathsf{EG}\,\phi\,]\!]$ *is the greatest fix-point of* $F(Z) = [\![\,\phi\,]\!] \cap \{\, s \in S \mid \exists\, s' \in R(s) \cap Z \,\}$
2. $[\![\,\mathsf{AG}\,\phi\,]\!]$ *is the greatest fix-point of* $F(Z) = [\![\,\phi\,]\!] \cap \{\, s \in S \mid \forall\, s' \in R(s) \cap Z \,\}$
3. $[\![\,\mathsf{EF}\,\phi\,]\!]$ *is the least fix-point of* $F(Z) = [\![\,\phi\,]\!] \;\cup\; \{\, s \in S \mid \exists\, s' \in R(s) \;\cap\; Z \,\}$
4. $[\![\,\mathsf{AF}\,\phi\,]\!]$ *is the least fix-point of* $F(Z) = [\![\,\phi\,]\!] \;\cup\; \{\, s \in S \mid \forall\, s' \in R(s) \;\cap\; Z \,\}$

5. $[\![\, \mathsf{E}\,[\phi\,\mathsf{U}\,\psi]\,]\!]$ *is the least fix-point of*

$$F(Z) = [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cap \{\, s \in S \mid \exists\, s' \in R(s) \cap Z\,\})$$

6. $[\![\, \mathsf{A}\,[\phi\,\mathsf{U}\,\psi]\,]\!]$ *is the least fix-point of*

$$F(Z) = [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cup \{\, s \in S \mid \forall\, s' \in R(s) \cap Z\,\}).$$

It is not difficult to check using the axioms of Table 3.3 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.

Proving the fixed point characterizations

We illustrate the proof of Theorem 33 by checking the case for $[\![\, \mathsf{E}\,[\phi\,\mathsf{U}\,\psi]\,]\!]$; the proofs for the other cases are conducted in a similar way. The proof consists of two parts: first we prove that the function $F(Z)$ 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.

Proving monotonicity. Let $Z_1, Z_2 \in 2^S$ such that $Z_1 \subseteq Z_2$. It must be proven that $F(Z_1) \subseteq F(Z_2)$. Then we derive:

$$\begin{array}{ll}
 & F(Z_1) \\
= & \{ \text{ definition of } F \ \} \\
 & [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cap \{\, s \in S \mid \exists\, s' \in R(s) \cap Z_1\,\}) \\
\subseteq & \{ \ Z_1 \subseteq Z_2; \text{ set calculus } \} \\
 & [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cap \{\, s \in S \mid \exists\, s' \in R(s) \cap Z_2\,\}) \\
= & \{ \text{ definition of } F \ \} \\
 & F(Z_2).
\end{array}$$

This means that for any arbitrary ψ and ϕ we have that $F(Z_1) \subseteq F(Z_2)$, and thus F is monotonic on 2^S. Given this result and the fact that $\langle 2^S, \subseteq \rangle$ is a complete lattice, it follows by Knaster-Tarski's theorem that F has a complete lattice of fixed points, including a unique least and greatest fixed point.

Proving the least fixed point. Recall that the least element of $\langle 2^S, \subseteq \rangle$ is $\varnothing$. If the set of states S has $n{+}1$ states, it is not difficult to prove that the least fixed point of F equals $F^{n+1}(\varnothing)$ for monotonic F on S. We now prove that

$[\![\, \mathsf{E}\,[\phi \,\mathsf{U}\, \psi]\,]\!] = F^{n+1}(\varnothing)$ by induction on n.

By definition we have $F^0(\varnothing) = \varnothing$. For $F(\varnothing)$ we derive:

$$
\begin{array}{ll}
 & F(\varnothing) \\
= & \{ \text{ definition of } F \ \} \\
 & [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cap \{ s \in S \mid \exists\, s' \in R(s) \cap \varnothing \}) \\
= & \{ \text{ calculus } \} \\
 & [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cap \varnothing) \\
= & \{ \text{ calculus } \} \\
 & [\![\,\psi\,]\!].
\end{array}
$$

Thus $F(\varnothing) = [\![\,\psi\,]\!]$, the set of states that can reach $[\![\,\psi\,]\!]$ in 0 steps. Now

$$
\begin{array}{ll}
 & F^2(\varnothing) \\
= & \{ \text{ definition of } F \ \} \\
 & [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cap \{ s \in S \mid \exists\, s' \in R(s) \cap F(\varnothing) \}) \\
= & \{ \ F(\varnothing) = [\![\,\psi\,]\!] \ \} \\
 & [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cap \{ s \in S \mid \exists\, s' \in R(s) \cap [\![\,\psi\,]\!] \}).
\end{array}
$$

Thus $F(F(\varnothing))$ 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 $F^{k+1}(\varnothing)$ 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:

$$
\begin{array}{ll}
 & [\![\, \mathsf{E}\,[\phi \,\mathsf{U}\, \psi]\,]\!] \\
= & \{ \text{ by the above reasoning } \} \\
 & \bigcup_{k \geqslant 0} F^{k+1}(\varnothing) \\
= & \{ \ F^i(\varnothing) \subseteq F^{i+1}(\varnothing) \ \} \\
 & F^{k+1}(\varnothing).
\end{array}
$$

Computing least and greatest fixed points

What do these results mean for the labelling approach to CTL model checking? We discuss this by means of an example. Consider $\chi = \mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$. An iterative approach is taken to the computation of $Sat_{EU}(\phi, \psi)$. Since $[\![\,\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]\,]\!]$ is the least fixed point of

$$F(Z) = [\![\,\psi\,]\!] \cup ([\![\,\phi\,]\!] \cap \{\, s \in S \mid \exists\, s' \in R(s) \cap Z \,\})$$

the problem of computing $Sat_{EU}(\phi, \psi)$ boils down to computing the fixed point of F, which is (according to Knaster-Tarski's theorem) equal to $\bigcup_i F^i(\varnothing)$. $\bigcup_i F^i(\varnothing)$ is computed by means of iteration: $\varnothing$, $F(\varnothing)$, $F(F(\varnothing)), \ldots$. Since we deal with *finite* CTL-models, this procedure is guaranteed to terminate, i.e. there exists some k such that $F^{k+1}(\varnothing) = F^k(\varnothing)$. Then, $F^{k+1}(\varnothing) = \bigcup_i F^i(\varnothing)$.

Intuitively, this iterative procedure can be understood as follows: one starts with no state being labelled with $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$. This corresponds to the approximation for which the formula is nowhere valid, i.e. $F^0(\varnothing) = \varnothing$. Then in the first iteration we consider $F^1(\varnothing)$ which reduces to $[\![\,\psi\,]\!]$ (see proof above), and label all states for which ψ holds with $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$. In the second iteration we consider $F^2(\varnothing)$ and label — in addition to the states which have already been labelled — the states s for which ϕ holds and which have some direct successor state for which ψ holds. We continue in this way until the fixed point $F^k(\varnothing)$ is reached for some k. At this point of the computation all states are labelled that satisfy $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$.

The resulting procedures Sat_{EU} and Sat_{AU}, for the formula $\mathsf{A}\,[\phi\,\mathsf{U}\,\psi]$, are listed in Table 3.4 and Table 3.5.

At the beginning of the $(i{+}1)$-th iteration in these procedures we have as an invariant $Q = F^{i+1}(\varnothing)$ and $Q' = F^i(\varnothing)$ for the function F. The iterations end when $Q = Q'$, that is, when $F^{i+1}(\varnothing) = F^i(\varnothing)$. The difference between Sat_{EU} and Sat_{AU} is that for the latter a new state is labelled if ϕ holds in that state and if all successor states are already labelled, i.e. are member of Q'. This corresponds to AX.

function $Sat_{EU}\,(\phi, \psi : Formula)$: **set of** *State*;
(* precondition: true *)
begin var Q, Q' : **set of** *State*;
  $Q, Q' := Sat(\psi), \varnothing$;
  do $Q \neq Q' \longrightarrow$
    $Q' := Q$;
    $Q := Q \cup (\{\, s \mid \exists\, s' \in Q.\, (s, s') \in R \,\} \cap Sat(\phi))$
  od;
  return Q
(* postcondition: $Sat_{EU}(\phi, \psi) = \{\, s \in S \mid \mathcal{M}, s \models \mathsf{E}\,[\phi \,\mathsf{U}\, \psi] \,\}$ *)
end

Table 3.4: Labelling procedure for $\mathsf{E}\,[\phi \,\mathsf{U}\, \psi]$

function $Sat_{AU}\,(\phi, \psi : Formula)$: **set of** *State*;
(* precondition: true *)
begin var Q, Q' : **set of** *State*;
  $Q, Q' := Sat(\psi), \varnothing$;
  do $Q \neq Q' \longrightarrow$
    $Q' := Q$;
    $Q := Q \cup (\{\, s \mid \forall\, s' \in Q.\, (s, s') \in R \,\} \cap Sat(\phi))$
  od;
  return Q
(* postcondition: $Sat_{AU}(\phi, \psi) = \{\, s \in S \mid \mathcal{M}, s \models \mathsf{A}\,[\phi \,\mathsf{U}\, \psi] \,\}$ *)
end

Table 3.5: Labelling procedure for $\mathsf{A}\,[\phi \,\mathsf{U}\, \psi]$

The iterative procedures for all formulas that have a least fixed point characterisation are performed in a similar way. For $\mathsf{EG}\,\phi$ and $\mathsf{AG}\,\phi$, which are greatest fixed points rather than least fixed points, the procedure is slightly different. Greatest fixed points are equal to $\bigcap_i F^i(S)$, which is computed by the series $S, F(S), F(F(S)), \ldots$. Intuitively, this means that one starts by labelling all states with the formula under consideration, say $\mathsf{EG}\,\phi$. This corresponds to $F^0(S) = S$. In each subsequent iteration states are "unlabelled" until a fixed point is reached.

Example 25. *Let the* CTL*-model $\mathcal{M}$ be shown in the first row of Figure 3.4 and suppose we are interested in checking whether $\phi = \mathsf{E}\,[p\,\mathsf{U}\,q]$ is valid in state s_0 of $\mathcal{M}$. For this purpose we compute $Sat_{EU}(p, q)$, i.e. $[\![\,\mathsf{E}\,[p\,\mathsf{U}\,q]\,]\!]$. It follows from the theory developed in this chapter that this reduces to computing the series $\varnothing$, $F(\varnothing)$, $F(F(\varnothing))$, $\ldots$ until a fixed point is reached, where according to Theorem 33*

$$F^{i+1}(Z) = [\![\,q\,]\!] \cup ([\![\,p\,]\!] \cap \{\, s \in S \mid \exists\, s' \in R(s) \cap F^i(Z)\,\}).$$

We start the computation by:

$$F(\varnothing) = [\![\,q\,]\!] \cup ([\![\,p\,]\!] \cap \{\, s \in S \mid \exists\, s' \in R(s) \cap \varnothing\,\}) = [\![\,q\,]\!] = \{\, s_2\,\}.$$

This is the situation just before starting the first iteration in $Sat_{EU}(p, q)$. This situation is illustrated in the first row, right column, where states which are in Q are colored black, and the others white.

For the second iteration we obtain

$$\begin{array}{ll}
 & F(F(\varnothing)) \\
= & \{\ \textit{definition of } F\ \} \\
 & [\![\,q\,]\!] \cup ([\![\,p\,]\!] \cap \{\, s \in S \mid \exists\, s' \in R(s) \cap F(\varnothing)\,\}) \\
= & \{\ F(\varnothing) = [\![\,q\,]\!] = \{\, s_2\,\};\ [\![\,p\,]\!] = \{\, s_0, s_1\,\}\ \} \\
 & \{\, s_2\,\} \cup (\{\, s_0, s_1\,\} \cap \{\, s \in S \mid \exists\, s' \in R(s) \cap \{\, s_2\,\})\,\}) \\
= & \{\ \textit{direct predecessors of } s_2 \textit{ are } \{\, s_1, s_3\,\}\ \} \\
 & \{\, s_2\,\} \cup (\{\, s_0, s_1\,\} \cap \{\, s_1, s_3\,\}) \\
= & \{\ \textit{calculus}\ \}
\end{array}$$

$\{ s_1, s_2 \}$.

The states that are now colored are those states from which a q-state (s_2) can be reached via a p-path (s_1) of length at most one.

From this result and the fact that the direct predecessors of $\{ s_1, s_2 \}$ are $\{ s_0, s_1, s_2, s_3 \}$ we obtain for the next iteration:

$$F^3(\varnothing) = \{ s_2 \} \cup (\{ s_0, s_1 \} \cap \{ s_0, s_1, s_2, s_3 \}) = \{ s_0, s_1, s_2 \}$$

Intuitively, the state s_0 is now labelled in the second iteration since it can reach a q-state (s_2) via a p-path (i.e. $s_0\, s_1$) of length two.

Since there are no other states in $\mathcal{M}$ that can reach a q-state via a p-path, the computation is finished. This can be checked formally by computing $F^4(\varnothing)$. The interested reader can check that indeed $F^4(\varnothing) = F^3(\varnothing)$, that is, the fixed point computation by $Sat_{EU}(p, q)$ has terminated. *(End of example.)*

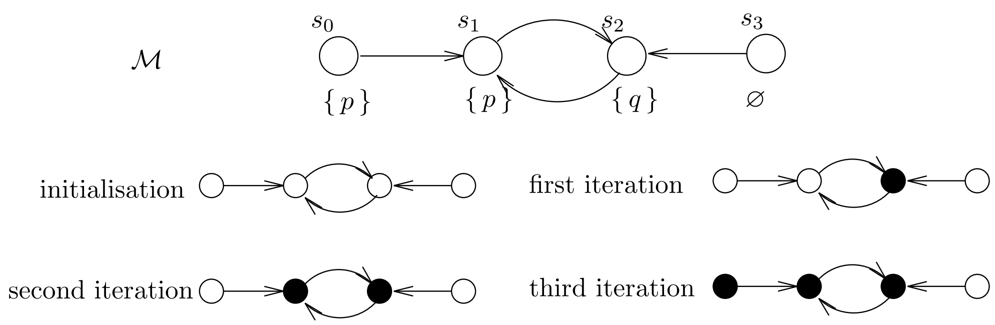


Figure 3.4: Example of iteratively computing $Sat_{EU}(p, q)$

Complexity analysis

The time complexity of model checking CTL is determined as follows. It is not difficult to see that Sat is computed for each sub-formula of ϕ, that is, $|\ Sub(\phi)\ |$ times. The size of $Sub(\phi)$ is proportional to the length of ϕ. The time complexity of the labelling procedures Sat_{AU} is proportional to $\mid S_{sys} \mid^3$, because the iteration is traversed $\mid S_{sys} \mid$ times in the worst case — starting with the empty set, in each iteration a single state is added to the set Q — while for the computation of each successive Q all transitions in R have to be considered, where R in the worst case equals $S_{sys} \times S_{sys}$. This entails that the worst time complexity of model checking CTL equals $\mathcal{O}(\mid \phi \mid \times \mid S_{sys} \mid^3)$ That is, the time complexity of model checking CTL is linear in the size of the formula to be checked and cubic in the number of states of the model of the system.

This time complexity can be improved by using a different, more efficient procedure for Sat_{EG} (Clarke, Emerson and Sistla, 1986). The concept behind this variant is to define CTL in terms of EX, $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$ and EG, and to proceed for checking $\mathsf{EG}\,\phi$ in state s as follows. First, consider only those states that satisfy ϕ and eliminate all other states and transitions. Then compute the maximal strong components in this reduced model that contain at least one transition, and check whether there is a strong component reachable from state s. If state s belongs to the reduced model and there exists such a path then — by construction of the reduced model — the property $\mathsf{EG}\,\phi$ is satisfied; otherwise, it is not. Using this variant, the complexity of model checking CTL can be reduced to

$$\mathcal{O}(\mid \phi \mid \times \mid S_{sys} \mid^2).$$

Thus, we conclude that

The worst-case time complexity of checking whether system-model sys satisfies the CTL*-formula* ϕ *is* $\mathcal{O}(\mid S_{sys} \mid^2 \times \mid \phi \mid)$

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, formulas in PLTL are never longer than, and mostly shorter than, their equivalent formulation in CTL. This follows directly from the fact that for formulas 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 (Clarke and Draghicescu, 1988). Even stronger, for each model $\mathcal{M}$ there does exist a PLTL-formula ϕ such that each CTL-formula equivalent to $\mathsf{E}\,\phi$ (or $\mathsf{A}\,\phi$) — if such a formula exists in CTL — has exponential length! This is nicely illustrated by the following example, which we adopted from (Kropf, 1997).

In summary, for a requirement 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 26. *We consider the problem of finding a Hamilton path in an arbitrary connected graph in terms of* PLTL *and* CTL*. Consider a graph $\mathcal{G} = (V, E)$ where V denotes the set of vertices and $E \subseteq V \times V$, the set of edges. Suppose $V = \{ v_0, \ldots, v_n \}$. A Hamilton path is a path through the graph which visits each state exactly once. (It is a travelling salesman problem where the cost of traversing an edge is the same for all edges.)*

We first describe the Hamilton path problem in PLTL*. The method used is to consider the graph $\mathcal{G}$ as a Büchi automaton which is obtained from $\mathcal{G}$ in the following way. We label each vertex v_i in V by a unique atomic proposition p_i, i.e.* $\text{Label}(v_i) = \{ p_i \}$*. In addition, we introduce a new (accepting) vertex w, such that $w \notin V$, with* $\text{Label}(w) = \{ q \}$*, where q is an atomic proposition different from any p_i. The vertex w is a direct successor of any node v_i, that is, the edges (v_i, w) are added to the graph, and w is a direct successor of itself, i.e. (w, w) is an edge. Figure 3.5 shows this construction for a connected graph with 4 vertices.*

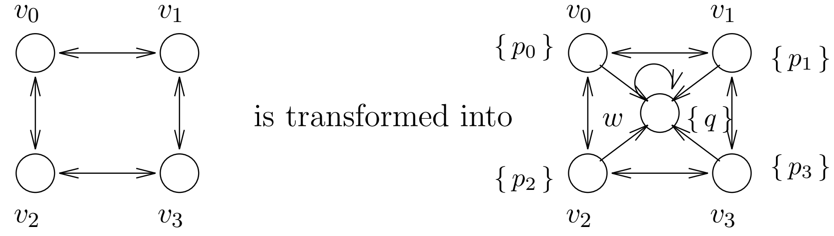


Figure 3.5: Encoding the Hamilton path problem in a model

Given this structure, the existence of a Hamilton path in a graph can be formulated in PLTL *as follows*[6]*:*

$$\mathsf{E}\left[(\forall i.\,\mathsf{F}\,p_i)\ \wedge\ \mathsf{X}^{\,n+1}q)\right].$$

where $\mathsf{X}^{\,1}q = \mathsf{X}\,q$ *and* $X^{n+1}q = \mathsf{X}\,(\mathsf{X}^{\,n}q)$*. 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. In order to obtain the desired infinite accepting run, such a path must have a suffix of the form* w^ω*, otherwise it would visit one or more states more than once. Notice that the length of the above formula is linear in the number of vertices in the graph.*

A formulation of the Hamilton path problem in CTL *exists and can be obtained in the following way. We start by constructing a* CTL*-formula* $g(p_0, \ldots, p_n)$ *which is valid when there exists a path from the current state that visits the states for which* $p_0 \ldots p_n$ *is valid in this order:*

$$g(p_0, \ldots, p_n) = p_0\ \wedge\ \mathsf{EX}\,(p_1\ \wedge\ \mathsf{EX}\,(\ldots\ \wedge\ \mathsf{EX}\,p_n)\ldots).$$

Because of the branching interpretation of CTL*, a formulation of the Hamilton path in* CTL *requires an explicit enumeration of all possible Hamilton paths. Let* $\mathcal{P}$ *be the set of permutations on* $\{\,0, \ldots n\,\}$*. Then we obtain:*

$$(\exists\,\theta \in \mathcal{P}.\,g(p_{\theta_0}, \ldots, p_{\theta_n}))\ \wedge\ \mathsf{EX}^{\,n+1}q.$$

By the explicit enumeration of all possible permutations we obtain a formula 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 Hamilton path problem, but this is impossible (Kropf, 1997). (End of example.)*

[6]Strictly speaking, 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 $\neg\,\mathsf{A}\,\neg\,\phi$ which can be checked by checking $\mathsf{A}\,\neg\,\phi$ which is a well-formed PLTL-formula according to Definition 23.

Overview of CTL model checking

We conclude this section with an overview of model checking CTL. This is shown in Figure 3.6.

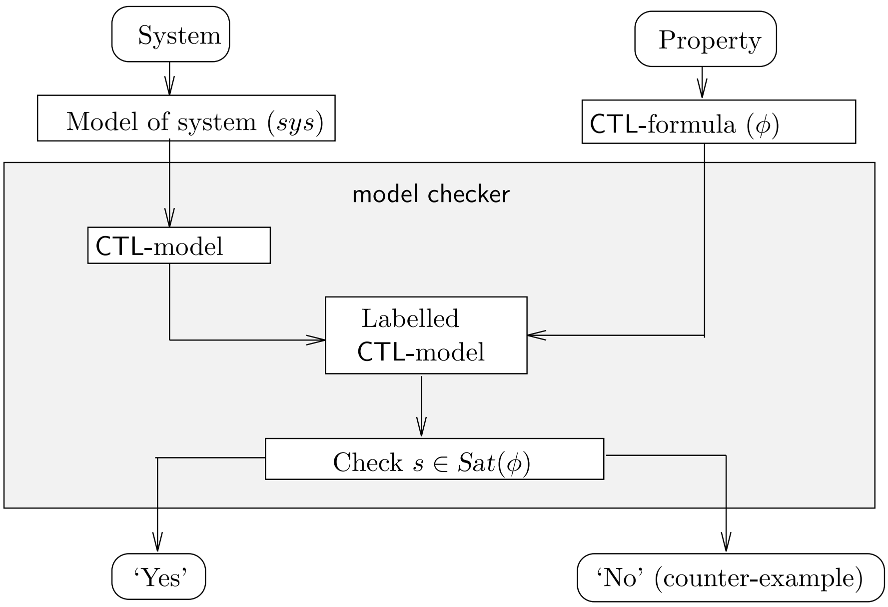


Figure 3.6: Overview of model checking CTL

3.9 Fairness

On the notion of fairness

As we have argued before, PLTL and CTL have incomparable expressiveness. An important category of properties that can be expressed in PLTL, but not in the branching temporal logic CTL, are so-called *fairness* constraints. We illustrate the concept of fairness by means of a frequently encountered problem in concurrent systems.

Example 27. Consider N processes $P_1, \ldots, P_N$ which require a certain service. There is one server process Server *which is expected to provide services to these processes. A possible strategy which* 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 which requires service. On finishing serving this process, repeat this selection procedure, once again starting with checking P_1. Now suppose that P_1 is always 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 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 is checked and (if needed) served.* *(End of example.)*

In verifying concurrent systems we are quite often only interested in execution sequences in which enabled transitions (statements) are executed in some fair way. In the next section, for instance, we will treat a mutual exclusion algorithm for two processes. In order to prove the absence of individual starvation — the situation in which a process which wants to enter its critical section has to wait infinitely long — we want to exclude those execution sequences in which a single 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 execution sequences 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.

Process fairness is a particular form of fairness. In general, fairness assumptions are needed for proving liveness properties ("something good will eventually happen") when the model to be checked considers non-determinism. 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 which can be potentially selected) is arbitrary. Other examples where non-determinism occurs are in sequential programs, when constructs like

$$\mathbf{do}\ \text{true} \longrightarrow S_1 [\,]\ \text{true} \longrightarrow S_2\ \mathbf{od}$$

are allowed. Here an unfair mechanism might always choose S_1 to be executed, and as a consequence, a property that is established by executing S_2 is never reached. Another prominent example where fairness is used to "resolve" nondeterminism is in modeling concurrent processes by means of interleaving. Interleaving boils down to modeling the concurrent execution of two independent processes by enumerating all the possible orders in which activities of the processes can be executed.

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

Types of fairness expressed in PLTL

In linear temporal logic such as PLTL, fairness can be expressed syntactically. We will briefly describe how three different forms of fairness can be formally specified in PLTL. 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). Then we distinguish between

- *Unconditional fairness.* A path is unconditionally fair with respect to ψ if

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

  holds. Such a property expresses, for instance, that a process enters its critical section infinitely often (regardless of any fairness constraint).

- *Weak fairness (justice).* A path is weakly fair with respect to ψ and fairness constraints ϕ if

  $$\mathsf{F}\,\mathsf{G}\,\phi \;\Rightarrow\; \mathsf{G}\,\mathsf{F}\,\psi.$$

  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, like a transition or an entire process, is continuously enabled ($\mathsf{F}\,\mathsf{G}\,\mathit{enabled}(a)$), then it will 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. In the literature, weak fairness is sometimes referred to as justice.

- *Strong fairness (compassion).* A path is strongly fair with respect to ψ and fairness constraints ϕ if

  $$\mathsf{G}\,\mathsf{F}\,\phi \;\Rightarrow\; \mathsf{G}\,\mathsf{F}\,\psi.$$

  The difference to weak fairness is that $\mathsf{F}\,\mathsf{G}$ is replaced by $\mathsf{G}\,\mathsf{F}$ in the premise. 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 computation 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.

Fairness constraints in CTL

Weak and strong fairness cannot be expressed syntactically In the branching temporal logic CTL. Additional constructs in the CTL-model are employed in order to be able to deal with fairness constraints. A CTL-model is extended such that it allows one to specify a set of fairness constraints $F_1, \ldots, F_k$. A fairness constraint is defined by (a predicate over) sets of states. For instance, in a mutual exclusion algorithm such a fairness constraint could be "process one is not in its critical section". This imposes the fairness constraint that there must be infinitely many states in a computation such that process one is not in its critical section. The basic approach is not to interpret CTL-formulas over all possible execution paths — as in the semantics of CTL which we have dealt with throughout this chapter — but rather to only consider the fair executions, i.e. those executions which satisfy all imposed fairness constraints $F_1, \ldots, F_k$. A fair CTL-model is defined as follows.

Definition 34. (**Fair model for CTL**)
A *fair* CTL-model is a quadruple $\mathcal{M} = (S, R, Label, \mathcal{F})$ where $(S, R, Label)$ is a CTL-model and $\mathcal{F} \subseteq 2^S$ is a set of fairness constraints.

Definition 35. (**An $\mathcal{F}$-fair path**)
A path $\sigma = s_0\, s_1\, s_2 \ldots$ is called $\mathcal{F}$-fair if for every set of states $F_i \in \mathcal{F}$ there are infinitely many states in σ that belong to F_i.

Formally, if $lim(\sigma)$ denotes the set of states which are visited by σ infinitely often, then σ is $\mathcal{F}$-fair if

$$lim(\sigma) \cap F_i \neq \varnothing \text{ for all } i.$$

Notice that this condition is identical to the condition for accepting runs of a generalized Büchi automaton (see Chapter 2). Indeed, a fair CTL-model is an ordinary CTL-model which is extended with a generalized Büchi acceptance condition.

The semantics of CTL in terms of fair CTL-models is identical to the semantics given earlier (cf. Definition 22), except that all quantifications over paths are interpreted over $\mathcal{F}$-fair paths rather than over all paths. Let $P^f_{\mathcal{M}}(s)$ be the set of $\mathcal{F}$-fair paths in $\mathcal{M}$ which start in state s. Clearly, $P^f_{\mathcal{M}}(s) \subseteq P_{\mathcal{M}}(s)$ if $\mathcal{F} \neq \varnothing$. The fair interpretation of CTL is defined in terms of the satisfaction relation $\models_f$. $(\mathcal{M}, s) \models_f \phi$ if and only if ϕ is valid in state s of fair model $\mathcal{M}$.

Definition 36. (**Fair semantics of CTL**)
Let $p \in AP$ be an atomic proposition, $\mathcal{M} = (S, R, Label, \mathcal{F})$ a fair CTL-model, $s \in S$, and ϕ, ψ CTL- formulas. The satisfaction relation $\models$ is defined by:

$$
\begin{array}{ll}
s \models_f p & \text{iff } p \in Label(s) \\
s \models_f \neg\,\phi & \text{iff } \neg\,(s \models_f \phi) \\
s \models_f \phi \,\vee\, \psi & \text{iff } (s \models_f \phi) \,\vee\, (s \models_f \psi) \\
s \models_f \mathsf{EX}\,\phi & \text{iff } \exists\,\sigma \in P^f_{\mathcal{M}}(s).\, \sigma[1] \models_f \phi \\
s \models_f \mathsf{E}\,[\phi \,\mathsf{U}\, \psi] & \text{iff } \exists\,\sigma \in P^f_{\mathcal{M}}(s).\, (\exists\, j \geqslant 0.\, \sigma[j] \models_f \psi \;\wedge \\
& \qquad\qquad (\forall\, 0 \leqslant k < j.\, \sigma[k] \models_f \phi)) \\
s \models_f \mathsf{A}\,[\phi \,\mathsf{U}\, \psi] & \text{iff } \forall\,\sigma \in P^f_{\mathcal{M}}(s).\, (\exists\, j \geqslant 0.\, \sigma[j] \models_f \psi \;\wedge \\
& \qquad\qquad (\forall\, 0 \leqslant k < j.\, \sigma[k] \models_f \phi)).
\end{array}
$$

The clauses for the propositional logic terms are identical to the semantics given earlier; for the temporal operators the difference lies in the quantifications which are over fair paths rather than over all paths. The expressiveness of fair CTL is strictly larger than that of CTL, and fair CTL is (like CTL) a subset of CTL^*. As CTL, fair CTL is incomparable to PLTL.

A few remarks are in order to see what type of fairness (unconditional, weak or strong fairness) can be supported by using this alternative interpretation for CTL. Suppose that $\phi_1, \ldots, \phi_n$ are the desired fairness constraints and ψ is the property to be checked for CTL-model $\mathcal{M}$. Let $\mathcal{M}_f$ be equal to $\mathcal{M}$ with the exception that $\mathcal{M}_f$ is a fair CTL-model where the fairness constraints $\phi_1, \ldots, \phi_n$ are realized by appropriate acceptance sets F_j. Then

$$\mathcal{M}_f, s \models_f \psi \text{ if and only if } \mathcal{M}, s \models \mathsf{A}\,[(\forall i.\, \mathsf{G}\,\mathsf{F}\,\phi_i) \ \Rightarrow \ \psi]$$

(Notice that $\mathsf{A}\,[(\forall i.\, \mathsf{G}\,\mathsf{F}\,\phi_i) \ \Rightarrow \ \psi]$ is a CTL^*-formula, and not a CTL-formula.) The intuition of this result is that the formulas $\mathsf{G}\,\mathsf{F}\,\phi_i$ exactly characterize those paths which visit F_j infinitely often.

For example, strong fairness (compassion) can now be imposed by checking

$$s \models_f \mathsf{AG}\,\mathsf{AF}\,\psi$$

where we use the result that the PLTL-formula $\mathsf{G}\,\mathsf{F}\,\psi$ is equivalent to the CTL-formula $\mathsf{AG}\,\mathsf{AF}\,\psi$.

Example 28. *Consider the* CTL*-model depicted in Figure 3.7 and suppose we are interested in checking* $\mathcal{M}, s_0 \models \mathsf{AG}\,[p \ \Rightarrow \ \mathsf{AF}\,q]$. *This property is invalid since there is a path* $s_0\, s_1\, (s_2\, s_4)^\omega$ *which never goes through a* q*-state. The reason for that this property is not valid is that at state* s_2 *there is a non-deterministic choice between moving either to* s_3 *or to* s_4*, and by always ignoring the possibility of going to* s_3 *we obtain a computation for which* $\mathsf{AG}\,[p \ \Rightarrow \ \mathsf{AF}\,q]$ *is invalid:*

$$\mathcal{M}, s_0 \not\models \mathsf{AG}\,[p \ \Rightarrow \ \mathsf{AF}\,q].$$

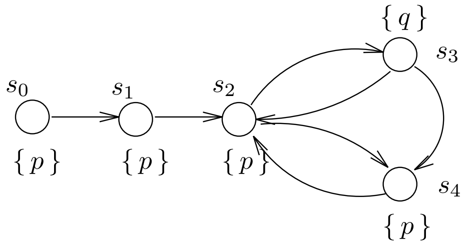


Figure 3.7: An example CTL-model

Usually, though, the intuition is that if there is infinitely often a choice of moving to s_3 then s_3 should be visited sometime (or infinitely often).

We transform the CTL*-model shown in Figure 3.7 into a fair* CTL*-model* $\mathcal{M}'$ *by defining* $\mathcal{F} = \{ F_1, F_2 \}$ *where* $F_1 = \{ s_3 \}$ *and* $F_2 = \{ s_4 \}$*. Let us now check* $\mathsf{AG}\,[p \Rightarrow \mathsf{AF}\,q]$ *on this fair model, that is, consider* $\mathcal{M}', s_0 \models_f \mathsf{AG}\,[p \Rightarrow \mathsf{AF}\,q]$*. Any* $\mathcal{F}$*-fair path starting at* s_0 *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. Such fair paths exclude paths like* $s_0\, s_1\, (s_2\, s_4)^\omega$*, since* s_3 *is never visited along this path. Thus we deduce that indeed*

$$\mathcal{M}', s_0 \models_f \mathsf{AG}\,[p \Rightarrow \mathsf{AF}\,q].$$

(End of example.)

Model checking fair CTL

It is beyond the scope of these lecture notes to treat in detail the extensions to the model checking algorithms which we have presented earlier which are concerned with the treatment of fairness. The fixed point characterizations of EF, EG and so on can be extended with fairness constraints. Based on these characterizations model checking of fair CTL can be performed in a similar way as model checking of CTL. The most important changes to the algorithms are that we have to consider the fair paths from a state, rather than all paths (like we did up to so far) for the procedures Sat_{EU} and Sat_{AU} and the treatment of $\mathsf{EX}\,\phi$. For this purpose, we need an algorithm to compute fair paths. In essence, fair paths can be determined as follows. Suppose we want to determine whether there is an $\mathcal{F}$-fair path in $\mathcal{M}$ starting at state s. Then:

1. We check whether we can reach from s some state $s' \in F_i$ for the fairness constraint $F_i \in \mathcal{F}$;
2. If such state exists for some i, we check whether we can reach s' from itself via a path which goes through a state in F_j for each $F_j \in \mathcal{F}$.

If both checks are positive then we have found a cycle which is reachable from the state s such that in the cycle each fairness constraint is satisfied. As for the calculation of emptiness for Büchi automata in the previous chapter, this strategy is the same as the computation of maximal strong components in a graph (cf. the programs *ReachAccept* and *DetectCycle* of Chapter 2). The standard algorithm to compute maximal strong components by Tarjan (1972) has a known worst case time-complexity of $\mathcal{O}(\mid S \mid + \mid R \mid)$.

A second major change to the model checking algorithms discussed before are the fixed point characterizations of $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$ and $\mathsf{A}\,[\phi\,\mathsf{U}\,\psi]$. These have to be changed in order to incorporate the fairness constraints $\mathcal{F}$. We do not provide the details here of this adaptation (see e.g. Clarke, Grumberg and Long, 1993) and only mention that the worst-case time-complexity of the model checking algorithms grows from $\mathcal{O}(\mid \phi \mid \times \mid S \mid^3)$ to

$$\mathcal{O}(\mid \phi \mid \times \mid S \mid^3 \times \mid \mathcal{F} \mid).$$

As for model checking plain CTL, this time complexity can be improved to being quadratic in the size of the state space, by using an alternative approach for EG.

3.10 The model checker SMV

The tool SMV (Symbolic Model Verifier) supports the verification of cooperating processes which "communicate" via shared variables. The tool has been developed at Carnegie-Mellon University by Ken McMillan and was originally developed for the automatic verification of synchronous hardware circuits. It is publically available, see `www.cs.cmu.edu/~modelcheck`. The model checker has been very useful for verifying hardware circuits and communication protocols. It has recently also been applied to large software systems, for instance, in the airplane-industry (Chan et. al, 1998). Processes can be executed in either a synchronous or an asynchronous fashion and model checking of CTL-formulas is supported. The model checking algorithms which are used by SMV are basically the algorithms which we have covered in this chapter and support the treatment of fair CTL. It is not our intention to give a complete introduction to the tool SMV in these lecture notes. For a more extensive introduction we refer to (McMillan, 1992/1993). We rather want to demonstrate by example how systems can be specified and verified using the model checker.

An SMV specification consists of process declarations, (local and global) variable declarations, formula declarations and a specification of the formulas that are to be verified. The main module is called `main`, as in C. The global structure of an SMV specification is:

```
MODULE main
VAR variable declarations
ASSIGN global variable assignments
DEFINITION definition of property to be verified   /* optional */
SPEC CTL-specification to verify

MODULE    /* submodule 1 */

MODULE    /* submodule 2 */

.............
```

The main ingredients of system specifications in SMV are:

- *Data types.* The only basic data types provided by SMV are bounded integer subranges and symbolic enumerated types.

- *Process declarations and initializations.* A process named `P` is defined as follows:

  ```
  MODULE P (formal parameters)
  VAR local definitions
  ASSIGN initial assignments to variables
  ASSIGN next assignments to variables
  ```

  This construct defines a module called `P` which can be instantiated by either

  ```
  VAR Pasync: process P(actual parameters)
  ```

  to obtain a process instantiation called `Pasync` which executes in asynchronous mode, or by

  ```
  VAR Psync: P(actual parameters)
  ```

  to obtain a process instantiation `Psync` which executes in synchronous mode. (The difference between asynchronous and synchronous is discussed below.)

- *Variable assignments.* In SMV a process is regarded as a finite-state automaton and is defined by listing for each (local and global) variable the initial value (i.e. the values in the initial state), and the value which is to be assigned to the variable in the next state. The latter value usually depends on the current values of the variables. For instance, `next(x) := x+y+2` assigns to `x` the value `x+y+2` in the next state. For variable `x` the assignment `init(x) := 3` denotes that `x` initially has the value 3. The assignment `next(x) := 0` assigns to `x` the value 0 in the next state. Assignments can be non-deterministic, e.g. the statement `next(x) := {0, 1}` means that the next value of `x` is either 0 or 1. Assignments can be conditional. For instance, the assignment

  ```
  next(x) := case b = 0: 2;
                  b = 1: {7, 12}
             esac;
  ```

assigns to variable `x` the value 2 if `b` equals 0 and (non-deterministically) the value 7 or 12 if `b` equals 1. If `x` is a variable in process instantiation `Q` then we write `Q.x`.

Assignments to global variables are only allowed if these variables occur as parameters of the process instantiation.

- *Synchronous versus asynchronous mode.* In the synchronous mode all assignments in any process are carried out in a single indivisible step. Intuitively this means that there is a single global clock and at each of its ticks each module performs a step. At any given time, a process is either running or not running. An assignment to the `next` value of a variable only occurs if the process is running. If not, the value of the variable remains unchanged in the next step.

  In asynchronous mode only the variables of the "active" process are assigned a new value. That is, at each tick of the global clock a single process is non-deterministically chosen for execution and a single step of this process is performed (while the other, not selected processes, keep their state). Synchronous composition is useful for e.g. modelling synchronous hardware circuits, whereas asynchronous composition is useful for modeling communication protocols or asynchronous hardware systems.

- *Specification of* CTL*-formulas.* For the specification of CTL-formulas, SMV uses the symbols `&` for conjunction, `|` for disjunction, `->` for implication and `!` for negation. The SMV model checker verifies that all possible initial states satisfy the specification.

Peterson and Fischer's mutual exclusion algorithm

The mutual exclusion algorithm of Peterson and Fischer is used by two processes which communicate via shared variables. It is intended to prevent the processes being simultaneously in their critical section. Each process i (i=1, 2) has local variables t_i and y_i which are readable by the other process but which can only be written by process i. These variables range over $\{ \perp, \text{false}, \text{true} \}$. The operator $\neg$ is not defined for $\perp$ and has its usual meaning for false and true. This means that in an expression like $\neg\, y_2 = y_1$ it is assumed that $y_1 \neq \perp$ and $y_2 \neq \perp$.

The initial values of the variables are $y_1, y_2, t_1, t_2 := \perp, \perp, \perp, \perp$. Process 1 is as follows:

$$\boxed{\begin{array}{l}
\textit{start}_1\text{: (* Process 1 *)} \\
t_1 := \textbf{if } y_2 = \text{false } \textbf{then} \text{ false } \textbf{else} \text{ true } \textbf{fi} \\
y_1 := t_1 \\
\textbf{if } y_2 \neq \perp \textbf{ then } t_1 := y_2 \textbf{ fi} \\
y_1 := t_1 \\
\textbf{loop while } y_1 = y_2 \\
\text{critical section 1; } y_1, t_1 := \perp, \perp \\
\textbf{goto } \textit{start}_1
\end{array}}$$

Process 2 is as follows:

$$\boxed{\begin{array}{l}
\textit{start}_2\text{: (* Process 2 *)} \\
t_2 := \textbf{if } y_1 = \text{true } \textbf{then} \text{ false } \textbf{else} \text{ true } \textbf{fi} \\
y_2 := t_2 \\
\textbf{if } y_1 \neq \perp \textbf{ then } t_2 := \neg\, y_2 \textbf{ fi} \\
y_2 := t_2 \\
\textbf{loop while } (\neg\, y_2) = y_1 \\
\text{critical section 2; } y_2, t_2 := \perp, \perp \\
\textbf{goto } \textit{start}_2
\end{array}}$$

The basic method of the algorithm is that when both processes compete to enter their critical section, it is guaranteed that $y_1 \neq \perp$ and $y_2 \neq \perp$, and therefore the conditions for entering the critical sections can never be valid simultaneously. Notice that the two processes are not symmetric.

Modeling the mutual exclusion algorithm in SMV

The translation of the algorithm just described into an SMV specification is fairly straightforward. The two process instantiations are created by

```
VAR prc1 : process P(t1,t2,y1,y2);
    prc2 : process Q(t1,t2,y1,y2);
```

The behavioral part of the first process of the previous section is given below. For our purpose we have labelled each statement in the algorithm starting from label `l1`. These labels are used to determine the next statement after the execution of a statement; they are a kind of program counter. The SMV code for the other process is rather similar, and is omitted here. Comments start with `--` and end with a carriage return.

```
MODULE P(t1,t2,y1,y2)
VAR label : {l1,l2,l3,l4,l5,l6,l7}; -- local variable label
ASSIGN init(label) := l1;           -- initial assignment
ASSIGN                              -- next assignments
   next(label) :=
       case
         label = l1              : l2;
         label = l2              : l3;
         label = l3              : l4;
         label = l4              : l5;
         label = l5 & y1 = y2    : l5;  -- loop
         label = l5 & !(y1 = y2) : l6;
         label = l6              : l7;
         label = l7              : l1;  -- goto start
       esac;

   next(t1) :=
       case
         label = l1 & y2 = false     : false;
         label = l1 & !(y2 = false)  : true;
         label = l3 & y2 = bottom    : t1;
         label = l3 & !(y2 = bottom) : y2;
         label = l6                  : bottom;
         1                           : t1;  -- otherwise keep
                                            -- t1 unchanged
```

```
        esac;

    next(y1) :=
        case
          label = l2 | label = l4 : t1;
          label = l6              : bottom;
          1                       : y1;  -- otherwise keep
                                         -- y1 unchanged
        esac;
```

Notice that the label `prc1.l6` corresponds to the critical section of process 1 and symmetrically, label `prc2.m6` corresponds to the critical section of process 2.

Model checking the SMV specification

There are two major properties which a mutual exclusion algorithm must possess. First there must be at most one process in its critical section at the same time. This is expressed in CTL as:

$$\mathsf{AG}\,\neg\,(prc1.label = l6\ \wedge\ prc2.label = m6)$$

In SMV this property is defined by

```
DEFINE MUTEX := AG !(prc1.label = l6 & prc2.label = m6)
```

where `MUTEX` is simply a macro. The result of the verification using `SPEC MUTEX` is positive:

```
-- specification MUTEX is true

resources used:
user time: 1.68333 s, system time: 0.533333 s
```

```
BDD nodes allocated: 12093
Bytes allocated: 1048576
BDD nodes representing transition relation: 568 + 1
reachable states: 157 (2^7.29462) out of 3969 (2^11.9546)
```

For the moment we ignore the messages concerning BDDs. SMV uses BDDs to represent (and store) the state space in a compact way in order to improve the capabilities of the model checker. These techniques will be treated in Chapter 5 of these lecture notes.

A second important property of a mutual exclusion algorithm is the absence of individual starvation. That means that it should not be possible that a process which wants to enter its critical section can never do so. This can be formalized in the current example in the following way

```
NST := AG ((prc1.label in {l1,l2,l3,l4,l5} -> AF prc1.label = l6) &
           (prc2.label in {m1,m2,m3,m4,m5} -> AF prc2.label = m6))
```

An alternative formulation of the intended property is

```
AG AF !(prc1.label in {l1,l2,l3,l4,l5}) &
AG AF !(prc2.label in {m1,m2,m3,m4,m5})
```

This states that from any state in the computation along any path the label of process 1 (and 2) will be different eventually from `l1` through `l5` (and `m1` through `m5`). This implies that process 1 and 2 are "regularly" in their critical section.

Checking the property `NST` using SMV yields an error as indicated by the following generated output:

```
-- specification NST is false
-- as demonstrated by the following execution sequence
-- loop starts here --
state 1.1:
```

```
NST = 0
MUTEX = 1
t1 = bottom
t2 = bottom
y1 = bottom
y2 = bottom
prc1.label = l1
prc2.label = m1

state 1.2:
[executing process prc2]

state 1.3:
[executing process prc2]
t2 = true
prc2.label = m2

state 1.4:
[executing process prc2]
y2 = true
prc2.label = m3

state 1.5:
[executing process prc2]
prc2.label = m4

state 1.6:
[executing process prc2]
prc2.label = m5

state 1.7:
[executing process prc2]
prc2.label = m6

state 1.8:
[executing process prc2]
```

```
t2 = bottom
y2 = bottom
prc2.label = m7

state 1.9:
prc2.label = m1
```

The counter-example is described as a sequence of changes to variables. If a variable is not mentioned, then its value has not been changed. The counterexample which the model checker produces indicates that there exists a loop in which `prc2` repeatedly obtains access to its critical section, whereas process `prc1` remains waiting forever. This should not surprise the reader, since the SMV system can, in each step of its execution, non-deterministically select a process (which can proceed) for execution. By always selecting one and the same process, we obtain the behavior illustrated above. One might argue that this result is not of much interest, since this unfair scheduling of processes is undesired. Stated differently, we would like to have a scheduling mechanism in which enabled processes — a process which can make a transition — are scheduled in a *fair* way. How this can be done is explained in the following subsection.

Model checking with fairness using SMV

SMV offers the possibility of specifying unconditional fairness requirements by stating (for *each* process):

```
FAIRNESS f
```

where `f` is an arbitrary CTL-formula. The interpretation of this statement is that the model checker, when checking some specification (which might differ from `f`), will ignore any path along which `f` does not occur infinitely often. Using this construct, we can establish process fairness in the following way. In SMV each process has a special variable `running` which is true if and only if the process is currently executing. When we add `FAIRNESS running` to our previous SMV specification for the mutual exclusion program, the previously obtained execution that showed the presence of individual starvation is indeed no longer possible (since it is unfair). We now obtain indeed absence of individual starvation. Therefore we can conclude that under the fairness assumption that each process is executed infinitely often, the algorithm of Peterson and Fischer does not lead to individual starvation.

As another example of the use of the `FAIRNESS` construct, let us modify the mutual exclusion algorithm slightly by allowing a process to stay in its critical section for some longer time. Recall that the label `prc1.l6` corresponds to process one being in its critical section. We now change the code of `MODULE P` so that we obtain:

```
ASSIGN
   next(label) :=
     case
     .........
     label = l6 : {l6, l7};
     .........
     esac;
```

Process one can now stay in its critical section infinitely long, while blocking process two from making any progress. Notice that adding `FAIRNESS running` to the above specification does not help: process two will then be scheduled for execution, but since it is blocked, it cannot make any progress. In order to avoid this situation we add the fairness constraint:

```
FAIRNESS  !(prc1.label = l6)
```

This statement means that when evaluating the validity of CTL-formulas, the path operators A and E range over the fair paths with respect to `!(prc1.label = l6)`. Stated differently, the runs for which `!(prc1.label = l6)` is not valid infinitely often are not considered. This indeed avoids that process one remains in its critical section forever.

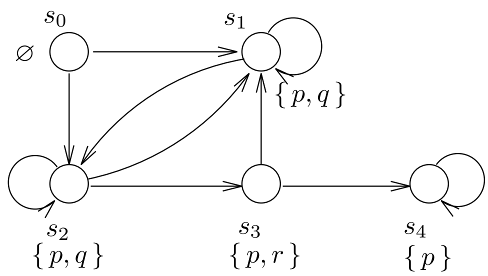
Exercises

EXERCISE 16. Prove the following equality:

$$\mathsf{A}\,[p\,\mathsf{U}\,q] \equiv \neg\,(\mathsf{E}\,[\neg\,q\,\mathsf{U}\,(\neg\,p\ \wedge\ \neg\,q)]\ \vee\ \mathsf{EG}\,\neg\,p)$$

where p and q are atomic propositions.

EXERCISE 17. Assume that we have the following model:



where states are represented by circles, for each state s the labelling $\pi(s)$ is given beside the circle representing s, and arrows denote relation R such that there is an arrow from s to s' if and only if $(s, s') \in R$. (a) Show that this model is a CTL-model and (b) check for each state in this model the validity of the following CTL-formulas in an informal way (that is, using the strategy of Example 23. Justify your answers.

1. $\mathsf{EG}\,p$
2. $\mathsf{AG}\,p$
3. $\mathsf{EF}\,[\mathsf{AG}\,p]$
4. $\mathsf{AF}\,[p\,\mathsf{U}\,\mathsf{EG}\,(p\ \Rightarrow\ q)]$
5. $\mathsf{EG}\,[((p\ \wedge\ q)\ \vee\ r)\,\mathsf{U}\,(r\,\mathsf{U}\,\mathsf{AG}\,p)]$

EXERCISE 18. We have defined in this chapter the logic CTL in terms of the basic operators EX, EU and AU. (EU and AU denote the combination of a path quantifier with the "until" operator.) Another possibility is to consider the operators AG, AX and AU as basic and to define all other operators (EX, EU and EG) as derivatives of these operators. Give a translation from an arbitrary CTL-formula into the basic operators AG, AX and AU.

EXERCISE 19. Give an example CTL-model which shows that the PLTL-formula $\mathsf{A}\,[\mathsf{F}\,\mathsf{G}\,p]$ (expressed as embedded CTL*-formula) and the CTL-formula $\mathsf{AF}\,\mathsf{AG}\,p$ are distinct.

EXERCISE 20.

1. Prove that the function $F(Z) = [\![\,\phi\,]\!] \cap \{\, s \mid \exists\, s' \in R(s) \cap Z \,\}$ is monotonic on the complete lattice $\langle 2^S, \subseteq \rangle$.
2. Given that $[\![\,\mathsf{EG}\,\phi\,]\!]$ is the greatest fixed point of the function $F(Z)$, give an algorithm Sat_{EG} which labels states in an arbitrary CTL-model $\mathcal{M} = (S, R, \pi)$ such that the postcondition of this procedure equals:

$$Sat_{EG}(\phi) = \{\, s \in S \mid \mathcal{M}, s \models \mathsf{EG}\,\phi \,\}.$$

EXERCISE 21. Let $\mathcal{M} = (S, P, \pi)$, where S is a set of states, P is a set of paths (where a path is an infinite sequence of states) and π is an assignment of atomic propositions to states. If we impose the following conditions on P, then (S, P, π) can be used as a CTL-model:

I $\sigma \in P \;\Rightarrow\; \sigma_{(1)} \in P$.

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

Here σ, σ' are paths, $\sigma_{(1)}$ is σ where the first element of σ is removed, and ρ, ρ' are finite sequences of states.

1. Give an intuitive interpretation of **I** and **II**.
2. Check whether the following sets of paths satisfy **I** and **II**. Here a^ω denotes an infinite sequence of a's, and a^* denotes a finite (possibly empty) sequence of a's. Justify 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) $\{ a\, a^*\, b^\omega \}$.

Exercise 22. Consider the CTL-model of Exercise 17 and compute the sets $[\![\mathsf{EF}\, p]\!]$ and $[\![\mathsf{EF}\, [\mathsf{AG}\, p]]\!]$ using the fixed point characterizations.

Exercise 23. Consider the mutual exclusion algorithm of 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 can take the values 1 and 2 and whose initial value is arbitrary. The i-th process (i=1, 2) is 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

Model this algorithm in SMV, and investigate whether this algorithm satisfies the following properties:

Mutual exclusion: two processes cannot be in their critical section at the same time.

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 to prove these properties in order to prohibit unfair executions (which might trivially violate these requirements).

EXERCISE 24. Dekker's algorithm is restricted to two processes. In the original mutual exclusion algorithm of 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 is non-deterministically chosen from the range $1, 2, \ldots, n$. The i-th process may be represented as follows:

var j : integer;
while true **do**
begin $b[i]$:= false;
    L_i : **if** $k \neq i$ **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 $(j \neq i \ \wedge \ \neg(c[j]))$ **then goto** L_i
            end
    $\langle$ critical section $\rangle$;
    $c[i]$:= true;
    $b[i]$:= true
end

Questions:

(a) Model this algorithm in SMV and check the mutual exclusion and individual starvation property. In case a property is not satisfied analyze the cause for this invalidity by checking the generated counter-example.

(b) Start with n=2, increase the number of processes gradually and compare the sizes of the state spaces.

EXERCISE 25. It is well-known that Dijkstra's solution for N processes is unfair, i.e. individual starvation is possible. In order to find a fair solution for N processes, Peterson proposed in 1981 the following variant. Let $Q[1 \ldots N]$ and $T[1 \ldots N{-}1]$ (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:

for $j := 1$ **to** $N - 1$ **do**
begin
    $Q[i] := j;$
    $T[j] := i;$
    wait until $(\forall\, k \neq i.\, Q[k] < j) \ \vee \ T[j] \neq i$
end;
$\langle$ critical Section $\rangle$;
$Q[i] := 0$

Check whether this is indeed a mutual exclusion program for $N = 2, 3, 4, \ldots$ (until the generated state space is too large to handle), and check whether Peterson's algorithm is indeed free from individual starvation.

3.11 Selected references

Kripke structures:

- S.A. KRIPKE. Semantical considerations on modal logic. *Acta Philosophica Fennica* **16**: 83–94, 1963.

Branching temporal logic:

- M. BEN-ARI, Z. MANNA, A. PNUELI. The temporal logic of branching time. *Acta Informatica* **20**: 207–226, 1983.
- E.A. EMERSON AND E.M. CLARKE. Using branching time temporal logic to synthesize synchronisation skeletons. *Science of Computer Programming* **2**: 241–266, 1982.

- D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science* **27**: 333–354, 1983.

Branching temporal logic versus linear temporal logic:

- 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.
- E.M. Clarke and I.A. Draghicescu. Expressibility results for linear time and branching time logics. In *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, LNCS 354, pages 428–437, 1988.
- T. Kropf. *Hardware Verifikation.* Habilitation thesis. University of Karlsruhe, 1997. (in German).

Model checking branching temporal logic using fixed point theory:

- E.M. Clarke, E.A. Emerson, 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.
- E.M. Clarke, O. Grumberg, D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency–Reflections and Perspectives*, LNCS 803, pages 124–175, 1993.
- M. Huth and M.D. Ryan. *Logic in Computer Science – Modelling and Reasoning about Systems.* Cambridge University Press, 1999 (to appear).

Fixed point theory:

- Z. Manna, S. Ness, J. Vuillemin. Inductive methods for proving properties of programs. *Communications of the ACM* **16**(8): 491–502, 1973.

Model checking branching temporal logic using tree automata:

- O. Bernholtz, M.Y. Vardi, P. Wolper. An automata-theoretic approach to branching-time model checking (extended abstract). In *Computer Aided Verification*, LNCS 818, pages 142–156, 1994.

Fairness:

- N. Francez. *Fairness*. Springer-Verlag, 1986.

SMV:

- K.L. McMillan. The SMV System. Technical Report CS-92-131, Carnegie-Mellon University, 1992.
- K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

Large case study with SMV:

- W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin and J.D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering* **24**(7): 498–519, 1998.

Chapter 4

Model Checking Real-Time Temporal Logic

Time-critical systems

Temporal logics like PLTL and CTL facilitate the specification of properties that focus on the temporal order of events. This temporal order is a qualitative notion; time is not considered quantitatively. For instance, let proposition p correspond to the occurrence of event A, and q correspond to the occurrence of event B. Then the PLTL-formula $\mathsf{G}\,[p \;\Rightarrow\; \mathsf{F}\,q]$ states that event A is always eventually followed by event B, but it does not state anything about how long the period between the occurrences of A and B will be. This absent notion of quantitative time is essential for the specification of *time-critical systems*. According to the "folk definition":

> *Time-critical systems are those 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.*

That is, time-critical systems must satisfy not only functional correctness requirements, but also timeliness requirements. Such systems are typically characterized by quantitative timing properties relating occurrences of events.

A typical example of a time-critical system is a train crossing: once the approach of a train is detected, the crossing needs to be closed within a certain time bound in order to halt car and pedestrian traffic before the train reaches the crossing. Communication protocols are another typical example of time-critical systems: after the transmission of a datum, a retransmission must be initiated if an acknowledgement is not received within a certain time bound. A third example of a time-critical system is a radiation machine where patients are imposed a high dosis of radiation during a limited time-period; a small exceed of this period is dangerous and can cause the patient's death.

Quantitative timing information is not only essential for time-critical systems, but is also a necessary ingredient in order to analyze and specify the performance aspects of systems. Typical performance-related statements like "job arrivals take place with an inter-arrival time of 30 seconds", can only be stated if one can *measure* the amount of elapsed time.

Quantitative time in temporal logics

In this chapter we treat the extension of a branching-time temporal logic with a quantitative notion of time. We have seen in the two previous chapters that linear and branching temporal logics are interpreted in terms of models where the notion of states plays a central role. The most important change when switching to quantitative time is the ability to measure time. That is, how is the notion of time incorporated in the state-based model and how is the relationship between states and time? As indicated by Koymans (1989) there are several issues that need to be addressed when incorporating time. The following issues, for example, need to be addressed:

- how should time elements be represented (explicitly or implicitly)?
- what is the notion of time reference (absolute or relative)?
- what is the semantical time domain (discrete or continuous)?
- how is time measure presented (additive or metric)?

As a result of the various ways in which these questions can be answered, a whole spectrum of real-time temporal logics has been developed. Some important real-time temporal logics are: Real-Time Temporal Logic (RTTL, Ostroff and Wonham, 1985), Metric Temporal Logic (Koymans, 1990), Explicit Clock Temporal Logic (Harel, Lichtenstein and Pnueli, 1990), Timed Propositional Temporal Logic (Alur and Henzinger, 1991), Timed Computational Tree Logic (Alur and Dill, 1989) and Duration Calculus (Chaochen, Hoare and Ravn, 1991). The first five are timed extensions of linear temporal logics, the one-but-last a real-time branching temporal logic, and the last an interval temporal logic, a timed extension of temporal logic that is suited for reasoning about time intervals. The purpose of these lecture notes is neither to discuss all possible variants nor to compare the merits or drawbacks of these approaches.

Choice of time domain

One of the most important issues is the choice of the semantical time domain. Since time — as in Newtonian physics — is of continuous nature, the most obvious choice is a continuous time domain like the real numbers. For synchronous systems, for instance, in which components proceed in a "lock-step" fashion, discrete time domains are appropriate. A prominent example of discrete-time temporal logics is RTTL. For synchronous systems one could alternatively consider to use ordinary temporal logic where the next operator is used as a means to "measure" the discrete advance of time. In a nutshell this works as follows. The basic idea is that a computation can be considered as a sequence of states where each transition from one state to the next state in the sequence can be thought of as a single tick of some computation clock. That is, $\mathsf{X}\,\phi$ is valid if after one tick of the clock ϕ holds. Let X^k be a sequence of k consecutive next operators defined by $\mathsf{X}^0\phi = \phi$ and $\mathsf{X}^{k+1}\phi = \mathsf{X}^k(\mathsf{X}\,\phi)$ for $k \geqslant 0$. The requirement that the maximal delay between event A and event B is at most 32 time-units then amounts to specifying

$$\mathsf{G}\,[p \;\Rightarrow\; \mathsf{X}^{<32}q]$$

where $\mathsf{X}^{<k}\phi$ is an abbreviation of the finite disjunction $\mathsf{X}^0\phi \ \vee \ \ldots \ \vee \ \mathsf{X}^{k-1}\phi$ and p, q correspond to the occurrences of A and B, respectively. (As an aside remark, the introduction of the X^k-operator makes the satisfiability problem for timed linear temporal logic EXPSPACE-hard.) Since we do not want to restrict ourselves to synchronous systems we consider a *real-valued* time-domain in these lecture notes. This allows for a more accurate specification of non-synchronous systems.

Model checking timed CTL

Suitability as specification language has originally been the driving force for the development of most real-time temporal logics. With the exception of the work by Alur and co-authors in the early nineties, automatic verification of real-time properties has not directly been the main motivation. This interest has increased significantly later (and is still increasing). Introducing time into a temporal logic must be carried out in a careful way in order to keep the model-checking problem be decidable. We concentrate on the temporal logic Timed CTL (TCTL, for short), for which model checking is decidable (Alur, Courcoubetis & Dill, 1993). This is a real-time branching temporal logic that represents time elements implicitly, supports relative time references, and is interpreted in terms of an additive, continuous time domain (i.e. $\mathbb{R}^+$, the non-negative real numbers). The basic idea of Timed CTL is to allow simple time constraints as parameters of the usual CTL temporal operators. In Timed CTL we can, for instance, specify that the maximal delay between event A and event B is 32 time-units by

$$\mathsf{AG}\,[p \ \Rightarrow \ \mathsf{AF}_{<32}\,q]$$

where as before proposition p (q) corresponds to the occurrence of event A (B). Timed CTL is a real-time extension of CTL for which model checking algorithms and several tools do exist.

The main difficulty of model-checking models in which the time domain is continuous is that the model to be checked has *infinitely* many states — for each time value the system can be in a certain state, and there are infinitely many of those values! This suggests that we are faced with an undecidable problem. In fact, the major question in model checking real-time temporal logics that must be dealt with is *how to cope with this infinite state space?* Since the solution to this key problem is rather similar for linear and branching temporal logics we treat one case, Timed CTL.

The essential idea to perform model checking on a continuous time-domain is to realize a discretization of this domain on-demand, i.e. depending on the property to be proven and the system model.

As a specification formalism for real-time systems we introduce the notion of *timed automata*, an extension of finite-state automata with clocks that are used to measure time. Since their conception (Alur & Dill, 1994), 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 model checking tools have been developed for timed automata. The central topic of the chapter will be to develop a model checking algorithm that determines the truth of a TCTL-formula with respect to a timed automaton.

Model checking real-time linear temporal logics

As we will see in this chapter the model-checking algorithm is strongly based on the labelling procedures that we introduced for automatically verifying CTL in Chapter 3. Although we will not deal with real-time *linear* temporal logics here, we like to mention that model checking of such logics — if decidable — is possible using the same paradigm as we have used for untimed linear temporal logics, namely Büchi automata. Alur and Henzinger (1989) have shown that using a timed extension of Büchi automata, model checking of an appropriate timed propositional linear temporal logic is feasible. Based on these theoretical results, Tripakis and Courcoubetis (1996) have constructed a prototype of a real-time extension of SPIN, the tool that we have used to illustrate model checking PLTL in Chapter 2. The crux of model checking real-time PLTL is to transform a timed Büchi automaton A_{sys} into a discretized variant, called a region automaton,

$\mathcal{R}(A_{sys})$. Then the scheme proceeds as for PLTL: a timed Büchi automaton $\mathcal{A}_{\neg\,\phi}$ is constructed for the property ϕ to be checked, and subsequently the emptiness of the product region automaton $\mathcal{R}(A_{sys} \otimes A_{\neg\,\phi})$ is checked. This algorithm uses polynomial space. The transformation of a Büchi automaton into its region automaton is based on an equivalence relation that is identical to the one used in the branching-time approach in this chapter. If A has n states (actually, locations) and k clocks, then $\mathcal{R}(A)$ has $n \cdot 2^{\mathcal{O}(^k \log c)} \cdot \mathcal{O}(2^k \cdot k!)$ states, where c is the largest integer constant occurring in A with which clocks are compared.

4.1 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. Clocks are used to measure the elapse of time.

Definition 37. (**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 can 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. The value of a clock thus denotes the amount of time that has been elapsed since it has been initialized. Conditions on the values of clocks are used as enabling conditions of transitions: only if the clock constraint is fulfilled, the transition is enabled, and can be taken; otherwise, the transition is blocked. 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 38. (**Clock constraints**)

[1]Note the (deliberate) distinction between location and state.

For set C of clocks with $x, y \in C$, the set of *clock constraints* over C, $\Psi(C)$, is defined by

$$\alpha ::= x \prec c \;\Big|\; x - y \prec c \;\Big|\; \neg\,\alpha \;\Big|\; (\alpha \;\wedge\; \alpha)$$

where $c \in \mathbb{N}$ and $\prec\; \in \{ <, \leqslant \}$.

Throughout this chapter we use typical abbreviations such as $x \geqslant c$ for $\neg\,(x < c)$ and $x = c$ for $x \leqslant c \;\wedge\; x \geqslant c$, and so on. We could allow addition of constants in clock constraints like $x \leqslant c{+}d$ for $d \in \mathbb{N}$, but addition of clock variables, like in $x{+}y < 3$, would make model checking undecidable. Also if c could be a real number, then model checking a timed temporal logic that is interpreted in a dense time domain — as in our case — becomes undecidable. Therefore, c is required to be a natural. Decidability 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, see also Example 36.

Definition 39. (**(Safety) timed automaton**)
A *timed automaton* $\mathcal{A}$ is a tuple $(L, l_0, E, \mathit{Label}, C, \mathit{clocks}, \mathit{guard}, \mathit{inv})$ with

- L, a non-empty, finite set of locations with initial location $l_0 \in L$
- $E \subseteq L \times L$, a set of edges
- $\mathit{Label} : L \longrightarrow 2^{AP}$, a function that assigns to each location $l \in L$ a set $\mathit{Label}(l)$ of atomic propositions
- C, a finite set of clocks
- $\mathit{clocks} : E \longrightarrow 2^C$, a function that assigns to each edge $e \in E$ a set of clocks $\mathit{clocks}(e)$
- $\mathit{guard} : E \longrightarrow \Psi(C)$, a function that labels each edge $e \in E$ with a clock constraint $\mathit{guard}(e)$ over C, and
- $\mathit{inv} : L \longrightarrow \Psi(C)$, a function that assigns to each location an *invariant*.

The function *Label* has the same role as for models for CTL and PLTL and associates to a location the set of atomic propositions that are valid in that location. As we will see, this function is only relevant for defining the satisfaction of atomic propositions in the semantics of TCTL. For edge e, the set $clocks(e)$ denotes the set of clocks that are to be reset when traversing e, and $guard(e)$ denotes the enabling condition that specifies when e can be taken. For location l, $inv(l)$ constrains the amount of time that can be spent in l.

For depicting timed automata we adopt the following conventions. Circles denote locations and edges 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 often omit the labelling *Label* in the sequel. 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.

Example 29. Figure 4.1(a) depicts a simple timed automaton with one clock x and one location l 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 4.1(b) gives an example execution of this timed automaton, by depicting the value of clock x. 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.

Changing the timed automaton of Figure 4.1(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 edge must be taken. This is illustrated in Figure 4.1(c) and (d).

Observe that the same effect is not obtained when strengthening the enabling constraint in Figure 4.1(a) into $2 \leqslant x \leqslant 3$ while keeping the invariant 'true' in l. In that case, the outgoing edge 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

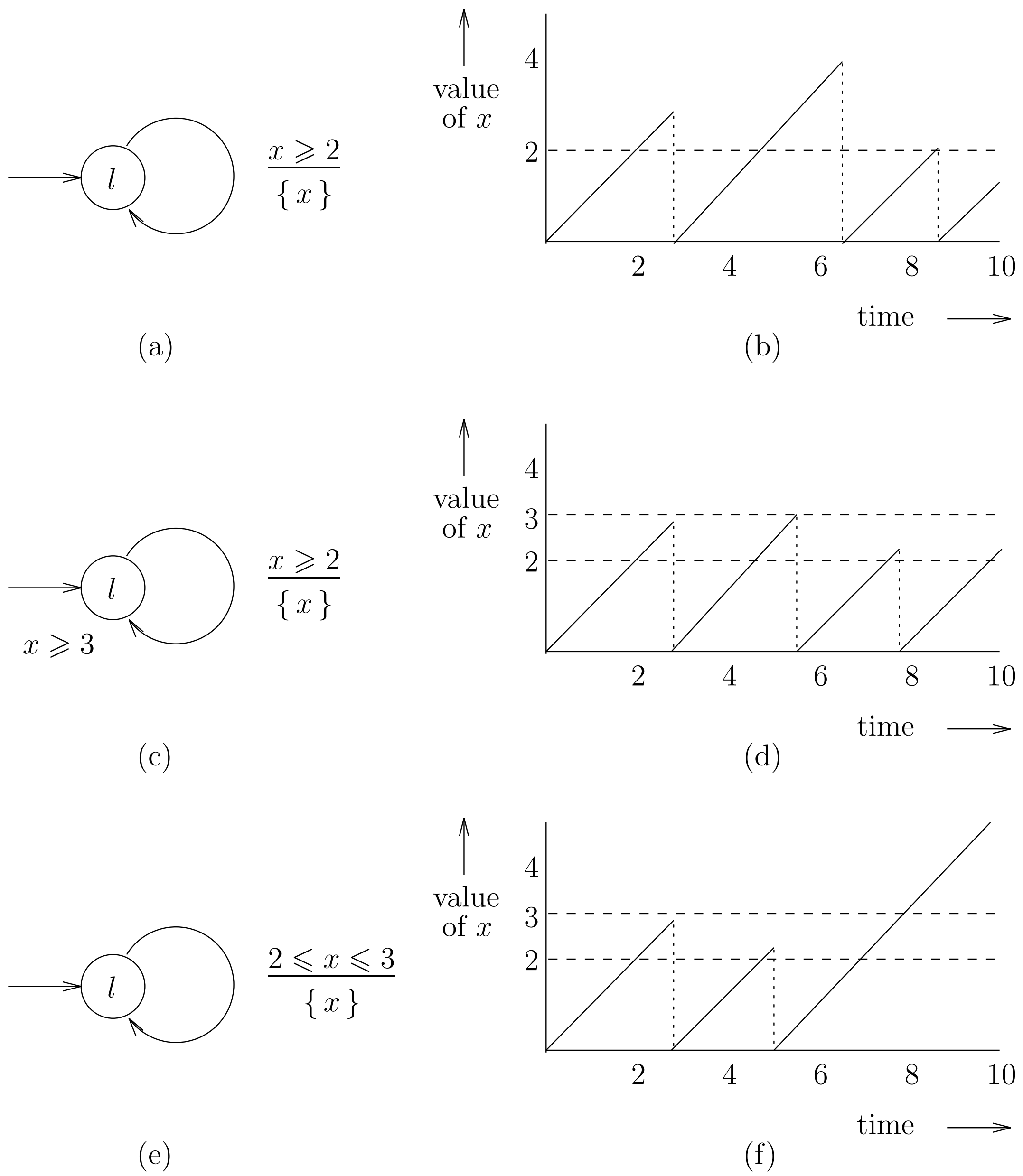


Figure 4.1: A timed automaton with one clock and a sample evolution of it

letting time pass while staying in l. This is illustrated in Figure 4.1(e) and (f). (End of example.)

We like to emphasize that 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 30. *Figure 4.2(a) depicts a timed automaton with two clocks, x and y. Initially, both clocks start running from value 0 on, until one time-unit has passed. From this point in time, both edges 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.)*

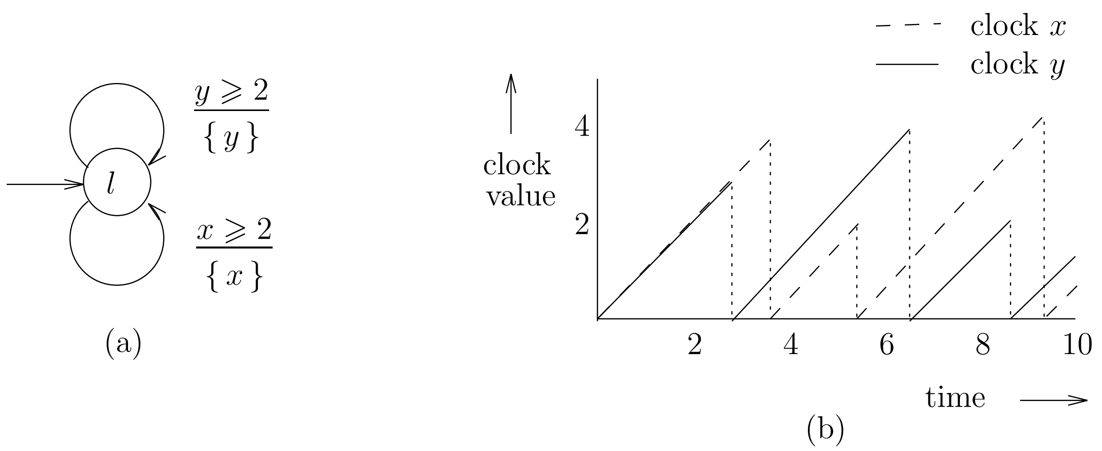


Figure 4.2: A timed automaton with two clocks and a sample evolution of it

Example 31. *Figure 4.3 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 4.3 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*

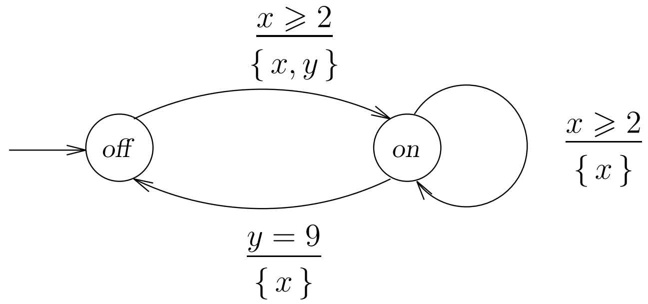


Figure 4.3: A timed automaton example: the switch

switches automatically off exactly 9 time-units after the most recent time the 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 edges 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.)*

In the above examples we have already shown how clocks evolve. The values of clocks are formally defined by *clock valuations*.

Definition 40. (**Clock valuation**)
A *clock valuation* v for a set of clocks C is a function $v : C \longrightarrow \mathbb{R}^+$, assigning to each clock $x \in C$ its current value $v(x)$.

Let $V(C)$ 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 32. *Consider the timed automaton of Figure 4.3. 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* $v''(x) = Label$ *and* $v''(y) = 3$. *Note that the latter state is not reachable.* *(End of example.)*

Let v be a clock valuation on set of clocks C. Clock valuation $v{+}t$ denotes that all clocks are increased by t with respect to valuation v. It is defined by $(v{+}t)(x) = v(x) + t$ 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, reset x in (reset y in v) is simply denoted reset x, y in v.

Example 33. *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* reset x in $(v{+}9)$, *clock x has value 0 and clock y reads 9. Clock valuation v' now equals* (reset x 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 41. (**Evaluation of clock constraints**)
For $x, y \in C$, $v \in V(C)$ and let $\alpha, \beta \in \Psi(C)$ we have

$$\begin{array}{ll} v \models x \prec c & \text{iff } v(x) \prec c \\ v \models x - y \prec c & \text{iff } v(x) - v(y) \prec c \\ v \models \neg\, \alpha & \text{iff } v \not\models \alpha \\ v \models \alpha \ \wedge\ \beta & \text{iff } v \models \alpha \ \wedge\ v \models \beta. \end{array}$$

For negation and conjunction, the rules are identical to those for propositional logic. In order to check whether $x \prec c$ is valid in v, it is simply checked whether $v(x) \prec c$. Similarly, constraints on differences of clocks are treated.

Example 34. *Consider clock valuation v, $v{+}9$ and* reset x in $(v + 9)$ *of the previous example and suppose we want to check the validity of $\alpha = x \leqslant 5$ and of $\beta = (x - y = 0)$. It follows $v \models \alpha$ and $v \models \beta$, since $v(x) = v(y) = 0$. We have*

$v+9 \not\models \alpha$ *since* $(v+9)(x) = 9 \not\leqslant 5$ *and* $v+9 \models \beta$ *since* $(v+9)(x) = (v+9)(y) = 9$. *The reader is invited to check that* $\mathsf{reset}\ x\ \mathsf{in}\ (v+9) \models \alpha$, *but* $\mathsf{reset}\ x\ \mathsf{in}\ (v+9) \not\models \beta$.

(End of example.)

4.2 Semantics of timed automata

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. There are two possible ways in which a timed automaton can proceed: by traversing an edge in the automaton, or by letting time progress while staying in the same location. Both ways are represented by a single transition relation $\longrightarrow$.

Definition 42. (**Transition system underlying a timed automaton**)
The transition system associated to timed automaton $\mathcal{A}$, denoted $\mathcal{M}(\mathcal{A})$, is defined as $(S, s_0, \longrightarrow)$ where:

- $S = \{ (l, v) \in L \times V(C) \mid v \models \mathit{inv}(l) \}$
- $s_0 = (l_0, v_0)$ where $v_0(x) = 0$ for all $x \in C$
- the transition relation $\longrightarrow \subseteq S \times (\mathbb{R}^+ \cup \{ * \}) \times S$ is defined by the rules:
  1. $(l, v) \xrightarrow{*} (l', \mathsf{reset}\ \mathit{clocks}(e)\ \mathsf{in}\ v)$ if the following conditions hold:
     (a) $e = (l, l') \in E$
     (b) $v \models \mathit{guard}(e)$, and
     (c) $(\mathsf{reset}\ \mathit{clocks}(e)\ \mathsf{in}\ v) \models \mathit{inv}(l')$
  2. $(l, v) \xrightarrow{d} (l, v+d)$, for positive real d, if the following condition holds:

     $$\forall\, d' \leqslant d.\, v+d' \models \mathit{inv}(l).$$

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 edge e in the timed automaton it must be that (b) v satisfies the clock constraint of e (otherwise the edge is disabled), and (c) the new clock valuation, that is obtained by resetting all clocks associated to e 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 $v{+}3 \models inv(l)$, the intermediate valuation $v{+}2$, for instance, invalidates the clock constraint.

Definition 43. (**Path**)
A *path* is an infinite sequence $s_0 \, a_0 \, s_1 \, a_1 \, s_2 \, a_2 \, \ldots$ of states alternated by transition labels such that $s_i \xrightarrow{a_i} s_{i+1}$ for all $i \geqslant 0$.

Note that labels can either be * in case of traversing an edge in the timed automaton, or (positive) real numbers in case of delay transition. The set of paths starting in a given state is defined as before. A *position* of a path is a pair (i, d) such that d equals 0 if $a_i = *$, i.e. in case of an edge-transition, and equals a_i otherwise. The set of positions of the form (i, d) characterizes the set of states visited by σ while going from state s_i to the successor s_{i+1}. Let $Pos(\sigma)$ denotes the set of positions in σ. For convenience, let $\sigma(i, d)$ denote the state $(s_i, v_i{+}d)$. A total order on positions is defined by:

$$(i, d) \ll (j, d') \text{ if and only if } i < j \; \vee \; (i = j \; \wedge \; d \leqslant d').$$

That is, position (i, d) precedes (j, d') if l_i is visited before l_j in σ, or if the positions point to the same location and d is at most d'.

In order to "measure" the amount of time that elapses on a path, we introduce:

Definition 44. (**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:

$$\begin{aligned}\Delta(\sigma, 0) &= 0 \\ \Delta(\sigma, i{+}1) &= \Delta(\sigma, i) + \begin{cases} 0 & \text{if } a_i = * \\ a_i & \text{if } a_i \in \mathbb{R}^+. \end{cases}\end{aligned}$$

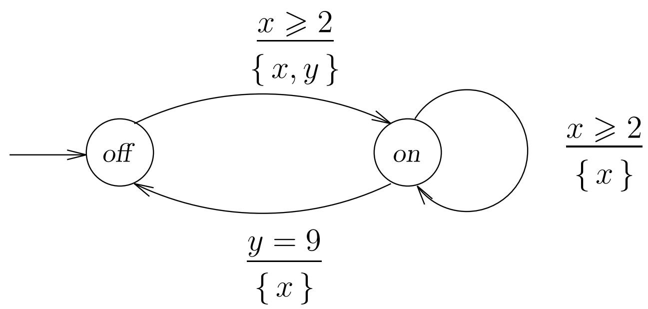
Path σ is called *time-divergent* if $\lim_{i \to \infty} \Delta(\sigma, i) = \infty$. The set of time-divergent paths starting at state s is denoted $P^{\infty}_{\mathcal{M}}(s)$. An example of a non time-divergent path is a path that visits an infinite number of states in a bounded amount of time. For instance, the path

$$s_0 \xrightarrow{2^{-1}} s_1 \xrightarrow{2^{-2}} s_2 \xrightarrow{2^{-3}} s_3 \ldots$$

is not time-divergent, since an infinite number of states is visited in the bounded interval $[\frac{1}{2}, 1]$. Since such paths are not realistic, usually non-Zeno timed automata are considered.

Definition 45. (**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.

Example 35. Recall the light switch from Example 31:



A prefix of an example path of the switch is

$$\begin{aligned}\sigma \;=\; & (\text{off}, v_0) \xrightarrow{3} (\text{off}, v_1) \xrightarrow{*} (\text{on}, v_2) \xrightarrow{4} (\text{on}, v_3) \xrightarrow{*} (\text{on}, v_4) \\ & \xrightarrow{1} (\text{on}, v_5) \xrightarrow{2} (\text{on}, v_6) \xrightarrow{2} (\text{on}, v_7) \xrightarrow{*} (\text{off}, v_8) \ldots\end{aligned}$$

with $v_0(x) = v_0(y) = 0$, $v_1 = v_0{+}3$, $v_2 = \textsf{reset}\ x, y\ \textsf{in}\ v_1$, $v_3 = v_2{+}4$, $v_4 = \textsf{reset}\ x\ \textsf{in}\ v_3$, $v_5 = v_4{+}1$, $v_6 = v_5{+}2$, $v_7 = v_6{+}2$ *and* $v_8 = \textsf{reset}\ x\ \textsf{in}\ v_7$. *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 $(\text{off}, v_1) \xrightarrow{*} (\text{on}, v_2)$ *is, for instance, possible since (a) there is an edge e from* off *to* on, *(b)* $v_1 \models x \geqslant 2$ *since* $v_1(x) = 3$, *and (c)* $v_2 \models \text{inv}(\text{on})$. *The positions of* σ *are (0,3), (1,0), (2,4), (3,0), (4,1), (5,2), (6,2), (7,0), respectively. We have, for instance,* $(1, 0) \ll (5, 2)$. *In addition* $\Delta(\sigma, 3) = 7$ *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 edge 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* $\text{inv}(\text{off}) = \text{inv}(\text{on}) = true$. *If we modify the switch such that* $\text{inv}(\text{off})$ *becomes* $y \leqslant 9$ *while* $\text{inv}(\text{on})$ *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.)*

4.3 Syntax of TCTL

Let $\mathcal{A}$ be a timed automaton, AP a set of atomic propositions and D a non-empty set of clocks that is disjoint from the clocks of $\mathcal{A}$, i.e. $C \cap D = \varnothing$. $z \in D$

is sometimes called a *formula clock*. Since clocks form a part of the syntax of the logic, the logic is sometimes referred to as explicit-clock temporal logic.

Definition 46. (**Syntax of timed computation tree logic**)
For $p \in AP$, $z \in D$, and $\alpha \in \Psi(C \cup D)$, the set of TCTL-formulas is defined by:

$$\boxed{\phi ::= p \;\big|\; \alpha \;\big|\; \neg\,\phi \;\big|\; \phi \;\vee\; \phi \;\big|\; z \;\mathsf{in}\; \phi \;\big|\; \mathsf{E}\,[\phi\,\mathsf{U}\,\phi] \;\big|\; \mathsf{A}\,[\phi\,\mathsf{U}\,\phi].}$$

Notice that α is a clock constraint over formula clocks and clocks of the timed automaton; it allows, for instance, the comparison of a formula clock and an automaton clock. The boolean operators true, false, $\wedge$, $\Rightarrow$ and $\Leftrightarrow$ are defined in the usual way (see Chapter 2). Clock z in $z \;\mathsf{in}\; \phi$ is called a *freeze identifier* and bounds the formula clock z in ϕ. The intuitive interpretation is that $z \;\mathsf{in}\; \phi$ is valid in state s if ϕ holds in s where clock z starts with value 0 in s. For instance, $z \;\mathsf{in}\; (z = 0)$ is a valid statement, while $z \;\mathsf{in}\; (z > 1)$ is not. The use of this freeze identifier is very useful in combination with until-constructions to specify typical timeliness requirements like punctuality, bounded response, and so on. Typically, one is not interested in formulas where formula clocks from D are free, that is, not bound. For simplicity we therefore consider all TCTL-formulas to be closed from now on. This means, for instance, that formulas like $x \leqslant 2$ and $z \;\mathsf{in}\; (z - y = 4)$ are illegal if x, y are formula clocks (i.e. $x, y \in D$); $x \;\mathsf{in}\; (x \leqslant 2)$ and $z \;\mathsf{in}\; (y \;\mathsf{in}\; (z - y = 4))$ are legal formulas, however.

Like for CTL, the basic operators of TCTL are until-formulas with existential and universal quantification over paths. EF, EG, and so on, are derivatives of these until-formulas like before. Notice that there are no timed variants of EX and AX in TCTL. The reason for this will become clear when we treat the semantics of the logic. The binding of the operators is identical to that of CTL.

Using the freeze identifier, temporal operators of CTL like $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$, $\mathsf{EF}\,\phi$ and so on, can be equipped with a time constraint in a succinct way. For instance, the formula

$$\mathsf{A}\,[\phi\,\mathsf{U}_{\leqslant 7}\,\psi]$$

intuitively means that along any path starting from the current state, the property ϕ holds continuously until within 7 time-units ψ becomes valid. It can be defined by

$$z \text{ in } \mathsf{A}\,[(\phi \;\wedge\; z \leqslant 7)\,\mathsf{U}\,\psi].$$

Alternatively, the formula

$$\mathsf{EF}_{<5}\,\phi$$

means that along some path starting from the current state, the property ϕ becomes valid within 5 time-units, and is defined by

$$z \text{ in } \mathsf{EF}\,(z < 5 \;\wedge\; \phi)$$

where EF is defined as before. Stated otherwise, $\mathsf{EF}_{<c}\,\phi$ denotes that a ϕ-state is reachable within c time-units. The dual expression, $\mathsf{AF}_{<c}\,\phi$ is valid if all executions lead to a ϕ-state within c time-units.

Example 36. Let $AP = \{\, b = 1, b < 2, b \geqslant 3 \,\}$ *be a set of atomic propositions. Example* TCTL*-formulas are:* $\mathsf{E}\,[(b < 2)\,\mathsf{U}_{\leqslant 21}\,(b \geqslant 3)]$, $\mathsf{AF}_{\geqslant 0}\,(b < 2)$, *and* $\mathsf{EF}_{<7}\,[\mathsf{EF}_{<3}\,(b = 1)]$. *Formula* $\mathsf{AX}_{<2}(b = 1)$ *is not a* TCTL*-formula since there is no next operator in the logic. The formula* $\mathsf{AF}_{\leqslant \pi}\,(b < 2)$ *is not a legal* TCTL*-formula, since* $\pi \notin \mathbb{N}$. *For the same reason* $\mathsf{AF}_{=\frac{2}{3}}\,[\mathsf{AG}_{<\frac{4}{5}}\,(b < 2)]$ *is also not allowed. Using appropriate scaling, however, this formula can be converted into the legal* TCTL*-formula* $\mathsf{AF}_{=5}\,[\mathsf{AG}_{<6}\,(b < 2)]$. *(End of example.)*

Notice that it is not allowed to write a bounded response time property like "there exists some unknown time t such that if ϕ holds, then before time t property ψ holds". For instance,

$$\exists\, t.\, z \text{ in } (\mathsf{AG}_{\geqslant 0}\,[(b = 1) \;\Rightarrow\; \mathsf{AF}\,(z < t \;\wedge\; b \geqslant 3)]).$$

Such quantifications over time makes model checking undecidable.

4.4 Semantics of TCTL

Recall that the interpretation of temporal logic is defined in terms of a model. For PLTL such a model consists of a (non-empty) set S of states, a total successor function R on states, and an assignment *Label* of atomic propositions to states. For CTL the function R is changed into a relation in order to take the branching nature into account. For TCTL, in addition, the notion of real-time has to be incorporated. This is needed in order, for instance, to be able to obtain a formal interpretation of formulas of the form $z \mathsf{in} \phi$ that contain time constraints. The basic idea is to consider a *state*, i.e. a location and a clock valuation. The location determines which atomic propositions are valid (using the labelling *Label*), while the clock valuation is used to determine the validity of the clock constraints in the formula at hand. Since clock constraints may contain besides clocks of the automaton, formula clocks, an additional clock valuation is used to determine the validity of statements about these clocks. Thus, the validity is defined for a state $s = (l, v)$ and formula clock valuation w. We will use $v \cup w$ to denote the valuation that for automata clock x equals $v(x)$ and for formula clock z equals $w(z)$.

The semantics of TCTL is defined by a satisfaction relation (denoted $\models$) that relates a transition system $\mathcal{M}$, state (i.e. a location plus a clock valuation over the clocks in the automaton), a clock valuation over the formula clocks occurring in ϕ, and a formula ϕ. We write $(\mathcal{M}, (s, w), \phi) \in \models$ by $\mathcal{M}, (s, w) \models \phi$. We have $\mathcal{M}, (s, w) \models \phi$ if and only if ϕ is valid in state s of model $\mathcal{M}$ under formula clock valuation w.

State $s = (l, v)$ satisfies ϕ if the "extended" state (s, w) satisfies ϕ where w is a clock valuation such that $w(z) = 0$ for all formula clocks z.

Definition 47. (**Semantics of TCTL**)
Let $p \in AP$ be an atomic proposition, $\alpha \in \Psi(C \cup D)$ a clock constraint over $C \cup D$, $\mathcal{M} = (S, \longrightarrow)$ an infinite transition system, $s \in S$, $w \in V(D)$, and ϕ, ψ

TCTL-formulae. The satisfaction relation $\models$ is defined by:

$$
\begin{array}{ll}
s, w \models p & \text{iff } p \in \mathit{Label}(s) \\
s, w \models \alpha & \text{iff } v \cup w \models \alpha \\
s, w \models \neg\, \phi & \text{iff } \neg\, (s, w \models \phi) \\
s, w \models \phi \;\vee\; \psi & \text{iff } (s, w \models \phi) \;\vee\; (s, w \models \psi) \\
s, w \models z \;\mathsf{in}\; \phi & \text{iff } s, \mathsf{reset}\; z \;\mathsf{in}\; w \models \phi \\
s, w \models \mathsf{E}\,[\phi \,\mathsf{U}\, \psi] & \text{iff } \exists\, \sigma \in P^{\infty}_{\mathcal{M}}(s).\, \exists\, (i, d) \in \mathit{Pos}(\sigma). \\
& \qquad (\sigma(i, d), w{+}\Delta(\sigma, i) \models \psi \;\wedge \\
& \qquad (\forall\, (j, d') \ll (i, d).\, \sigma(j, d'), w{+}\Delta(\sigma, j) \models \phi \;\vee\; \psi)) \\
s, w \models \mathsf{A}\,[\phi \,\mathsf{U}\, \psi] & \text{iff } \forall\, \sigma \in P^{\infty}_{\mathcal{M}}(s).\, \exists\, (i, d) \in \mathit{Pos}(\sigma). \\
& \qquad ((\sigma(i, d), w{+}\Delta(\sigma, i)) \models \psi \;\wedge \\
& \qquad (\forall\, (j, d') \ll (i, d).\, (\sigma(j, d'), w{+}\Delta(\sigma, j)) \models \phi \;\vee\; \psi)).
\end{array}
$$

For atomic propositions, negations and disjunctions, the semantics is defined in a similar way as before. Clock constraint α is valid in (s, w) if the values of the clocks in v, the valuation in s, and w satisfy α. Formula $z \;\mathsf{in}\; \phi$ is valid in (s, w) if ϕ is valid in (s, w') where w' is obtained from w by resetting z. The formula $\mathsf{E}\,[\phi \,\mathsf{U}\, \psi]$ is valid in (s, w) if there exists a time-divergent path σ starting in s, that satisfies ψ at some position, and satisfies $\phi \;\vee\; \psi$ at all preceding positions. The reader might wonder why it is not required — like in the untimed variants of temporal logic we have seen before — that just ϕ holds at all preceding positions. The reason for this is the following. Assume $i = j$, $d \leqslant d'$ and suppose ψ does not contain any clock constraint. Then, by definition $(i, d) \ll (j, d')$. Moreover, if $i = j$ it means that the positions (i, d) and (j, d') point to the same location in the timed automaton. But then, the same atomic propositions are valid in (i, d) and (j, d'), and, since ψ does not contain any clock constraint, the validity of ψ in (i, d) and (j, d') is the same. Thus, delaying in locations forces us to use this construction.

Consider now, for instance, the formula $\mathsf{E}\,[\phi \,\mathsf{U}_{<12}\, \psi]$ and state s in $\mathcal{M}$. This formula means that there exists an s-path which has an initial prefix that lasts at most 12 units of time, such that ψ holds at the end of the prefix and ϕ holds at all intermediate states. Formula $\mathsf{A}\,[\phi \,\mathsf{U}_{<12}\, \psi]$ requires this property to be valid for all paths starting in s. The formulae $\mathsf{E}\,[\phi \,\mathsf{U}_{\geqslant 0}\, \psi]$ and $\mathsf{A}\,[\phi \,\mathsf{U}_{\geqslant 0}\, \psi]$ are abbreviated by

$\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$ and $\mathsf{A}\,[\phi\,\mathsf{U}\,\psi]$, respectively. This corresponds to the fact that the timing constraint $\geqslant 0$ is not restrictive.

For PLTL the formula $\mathsf{F}\,(\phi \,\wedge\, \psi)$ implies $\mathsf{F}\,\phi \,\wedge\, \mathsf{F}\,\psi$, but the reverse implication does not hold (the interested reader is invited to check this). Similarly, in CTL we have that $\mathsf{AF}\,(\phi \,\wedge\, \psi)$ implies $\mathsf{AF}\,\phi \,\wedge\, \mathsf{AF}\,\psi$, but not the other way around. Due to the possibility of requiring punctuality, the validity of a property at a precise time instant, we obtain for $\mathsf{AF}_{=c}\,(\phi \,\wedge\, \psi)$ also the implication in the other direction:

$$\mathsf{AF}_{=c}\,(\phi \,\wedge\, \psi) \equiv (\mathsf{AF}_{=c}\,\phi \,\wedge\, \mathsf{AF}_{=c}\,\psi)$$

This can be formally proven as follows:

$$
\begin{array}{ll}
 & s, w \models \mathsf{AF}_{=c}\,(\phi \,\wedge\, \psi) \\
\Leftrightarrow & \{ \text{ definition of } \mathsf{AF}_{=c} \ \} \\
 & s, w \models \mathsf{A}\,[\mathrm{true}\,\mathsf{U}_{=c}\,(\phi \,\wedge\, \psi)] \\
\Leftrightarrow & \{ \text{ definition of } \ \mathsf{U}_{=c}\,\text{; let } z \text{ be a `fresh' clock } \} \\
 & s, w \models z \ \mathsf{in}\ \mathsf{A}\,[\mathrm{true}\,\mathsf{U}\,(z = c \,\wedge\, \phi \,\wedge\, \psi)] \\
\Leftrightarrow & \{ \text{ semantics of } z \ \mathsf{in}\ \phi \ \} \\
 & s, \mathsf{reset}\ z\ \mathsf{in}\ w \models \mathsf{A}\,[\mathrm{true}\,\mathsf{U}\,(z = c \,\wedge\, \phi \,\wedge\, \psi)] \\
\Leftrightarrow & \{ \text{ semantics of } \mathsf{A}\,[\phi\,\mathsf{U}\,\psi]\text{; true is valid in any state } \} \\
 & \forall\,\sigma \in P^{\infty}_{\mathcal{M}}(s).\,\exists\,(i,d) \in \mathit{Pos}(\sigma). \\
 & \qquad (\sigma(i,d), (\mathsf{reset}\ z\ \mathsf{in}\ w){+}\Delta(\sigma,i) \models (z = c \,\wedge\, \phi \,\wedge\, \psi)) \\
\Leftrightarrow & \{ \text{ predicate calculus } \} \\
 & \forall\,\sigma \in P^{\infty}_{\mathcal{M}}(s).\,\exists\,(i,d) \in \mathit{Pos}(\sigma). \\
 & \qquad (\sigma(i,d), (\mathsf{reset}\ z\ \mathsf{in}\ w){+}\Delta(\sigma,i) \models (z = c \,\wedge\, \phi) \,\wedge\, (z = c \,\wedge\, \psi)) \\
\Leftrightarrow & \{ \text{ semantics of } \wedge \ \} \\
 & \forall\,\sigma \in P^{\infty}_{\mathcal{M}}(s).\,\exists\,(i,d) \in \mathit{Pos}(\sigma). \\
 & \qquad\qquad ((\sigma(i,d), (\mathsf{reset}\ z\ \mathsf{in}\ w){+}\Delta(\sigma,i) \models (z = c \,\wedge\, \phi)) \\
 & \qquad\quad \wedge\ (\sigma(i,d), (\mathsf{reset}\ z\ \mathsf{in}\ w){+}\Delta(\sigma,i) \models (z = c \,\wedge\, \psi)) \\
\Leftrightarrow & \{ \text{ semantics of } \mathsf{A}\,[\phi\,\mathsf{U}\,\psi]\text{; true is valid in any state } \} \\
 & \qquad s, \mathsf{reset}\ z\ \mathsf{in}\ w \models \mathsf{A}\,[\mathrm{true}\,\mathsf{U}\,(z = c \,\wedge\, \phi)]
\end{array}
$$

$$\wedge \;\; s, \mathsf{reset}\ z\ \mathsf{in}\ w \models \mathsf{A}\,[\mathrm{true}\,\mathsf{U}\,(z = c \;\wedge\; \psi)]$$

$$\Leftrightarrow \quad \{ \text{ semantics of } z \text{ in } \phi \ \}$$

$$s, w \models z\ \mathsf{in}\ \mathsf{A}\,[\mathrm{true}\,\mathsf{U}\,(z = c \;\wedge\; \phi)] \;\wedge\; s, w \models z\ \mathsf{in}\ \mathsf{A}\,[\mathrm{true}\,\mathsf{U}\,(z = c \;\wedge\; \psi)]$$

$$\Leftrightarrow \quad \{ \text{ definition of } \mathsf{U}_{=c}\text{; definition of } \mathsf{AF}_{=c} \ \}$$

$$s, w \models \mathsf{AF}_{=c}\,\phi \;\wedge\; s, w \models \mathsf{AF}_{=c}\,\psi$$

$$\Leftrightarrow \quad \{ \text{ semantics of } \wedge \ \}$$

$$s, w \models (\mathsf{AF}_{=c}\,\phi \;\wedge\; \mathsf{AF}_{=c}\,\psi).$$

The reader is invited to check that this equivalence only holds for the timing constraint $= c$, but not for the others. Intuitively this stems from the fact that in $\mathsf{AF}_{=c}\,(\phi \;\wedge\; \psi)$ we not only require that eventually $\phi \;\wedge\; \psi$ hold, but also at exactly the same time instant. But then also $\mathsf{AF}_{=c}\,\phi$ and $\mathsf{AF}_{=c}\,\psi$. Since conditions like $< c$, $\geqslant c$, and so on, do not single out one time instant a similar reasoning does not apply to the other cases.

Extending the expressivity of CTL with real-time aspects in a dense time-setting has a certain price to pay. Recall that the satisfiability problem of a logic is as follows: does there exist a model $\mathcal{M}$ (for that logic) such that $\mathcal{M} \models \phi$ for a given formula ϕ? For PLTL and CTL satisfiability is decidable, but for TCTL it turns out that the satisfiability problem is *highly undecidable*. Actually, for most real-time logics the satisfiability problem is undecidable: Alur & Henzinger (1993) showed that only a very weak arithmetic over a discrete time domain can result in a logic for which satisfiability is decidable.

Another difference with the untimed variant is the extension with *past* operators, operators that refer to the history rather than to the future (like F, U and G). Some example past operators are previous (the dual of next), and since (the dual of until). It is known that extending CTL and PLTL with past operators does not enhance their expressivity; the major reason for considering these operators for these logics is to improve specification convenience — some properties are easier to specify when past operators are incorporated. For TCTL, however, this is not the case: extending TCTL (actually, any temporal logic interpreted over a continuous domain) with past operators extends its expressivity (Alur & Henzinger, 1992). Model checking such logics is however still possible using a natural accommodation, and the worst case complexity is not increased.

4.5 Specifying timeliness properties in TCTL

In this section we treat some typical timeliness requirements for time-critical systems and formally specify these properties in TCTL.

1. *Promptness* requirement: specifies a maximal delay between the occurrence of an event and its reaction. For example, every transmission of a message is followed by a reply within 5 units of time. Formally,

   $$\mathsf{AG}\,[\mathit{send}(m) \;\Rightarrow\; \mathsf{AF}_{<5}\,\mathit{receive}\,(r_m)]$$

   where it is assumed that m and r_m are unique, and that $\mathit{send}(m)$ and $\mathit{receive}(r_m)$ are valid if m is being sent, respectively if r_m is being received, and invalid otherwise.

2. *Punctuality* requirement: specifies an exact delay between events. For instance, there exists a computation during which the delay between transmitting m and receiving its reply is exactly 11 units of time. Formally:

   $$\mathsf{EG}\,[\mathit{send}(m) \;\Rightarrow\; \mathsf{AF}_{=11}\,\mathit{receive}\,(r_m)]$$

3. *Periodicity* requirement: specifies that an event occurs with a certain period. For instance, consider a machine that puts boxes on a moving belt that moves with a constant speed. In order to maintain an equal distance between successive boxes on the belt, the machine is required to put boxes periodically with a period of 25 time-units, say. Naively, one is tempted to specify this periodic behavior by

   $$\mathsf{AG}\,[\mathsf{AF}_{=25}\,\mathit{putbox}]$$

   where the atomic proposition *putbox* is valid if the machine puts a box on the belt, and invalid otherwise. This formulation does, however, allow to put additional boxes on the belt in between two successive box placings that have a spacing of 25 time-units. For instance, a machine that puts boxes at time instant 25, 50, 75, ... and at time 10, 35, 60, ... on the belt, satisfies the above formulation. A more accurate formulation of our required periodicity requirement is:

$$\mathsf{AG}\,[\mathit{putbox} \;\Rightarrow\; \neg\,\mathit{putbox}\,\mathsf{U}_{=25}\,\mathit{putbox}]$$

This requirement specifies a delay of 25 time-units between two successive box placings, and in addition, requires that no such event occurs in between[2].

4. *Minimal delay* requirement: specifies a minimal delay between events. For instance, in order to ensure the safety of a railway system, the delay between two trains at a crossing should be at least 180 time units. Let *tac* be an atomic proposition that holds when the train is at the crossing. This minimal delay statement can be formalized by:

$$\mathsf{AG}\,[\mathit{tac} \;\Rightarrow\; \neg\,\mathit{tac}\,\mathsf{U}_{\geqslant 180}\,\mathit{tac}]$$

The reason for using the until-construct is similar to the previous case for periodicity.

5. *Interval delay* requirement: specifies that an event must occur within a certain interval after another event. Suppose, for instance, that in order to improve the throughput of the railway system one requires that trains should have a maximal distance of 900 time-units. The safety of the train system must be remained. Formally, we can easily extend the previous minimal delay requirement:

$$\mathsf{AG}\,[\mathit{tac} \;\Rightarrow\; (\neg\,\mathit{tac}\,\mathsf{U}_{\geqslant 180}\,\mathit{tac} \;\wedge\; \neg\,\mathit{tac}\,\mathsf{U}_{\leqslant 900}\,\mathit{tac})]$$

Alternatively, we could write

$$\mathsf{AG}\,[\mathit{tac} \;\Rightarrow\; \neg\,\mathit{tac}\,\mathsf{U}_{=180}\,(\mathsf{AF}_{\leqslant 720}\,\mathit{tac})]$$

It specifies that after a train at the crossing it lasts 180 time units (the safety requirement) before the next train arrives, and in addition this next train arrives within 720+180=900 time-units (the throughput requirement).

[2]The reader might wonder why such construction is not used for the punctuality requirement. This is due to the fact that r_m is unique, so $\mathit{receive}(r_m)$ can be valid at most once. This is not true for the predicate *putbox*, since putting boxes on the belt is supposed to be a repetitive activity.

4.6 Clock equivalence: the key to model checking real time

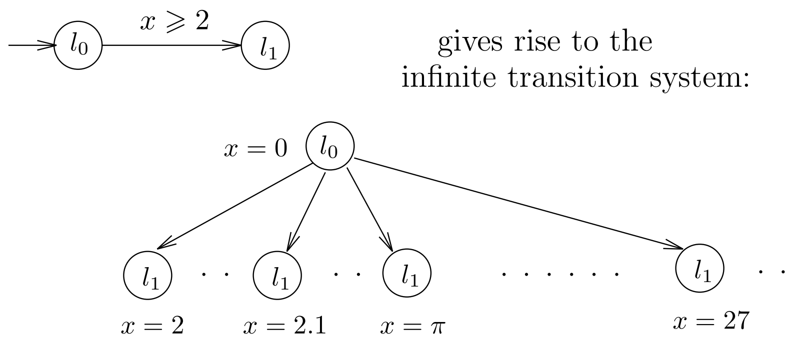
The satisfaction relation $\models$ for TCTL is not defined in terms of timed automata, but in terms of infinite transition systems. The model checking problem for timed automata can easily be defined in terms of $\mathcal{M}(\mathcal{A})$, the transition system underlying $\mathcal{A}$ (cf. Definition 42). Let the initial state of $\mathcal{M}(\mathcal{A})$ be $s_0 = (l_0, v_0)$ where l_0 is the initial location of $\mathcal{A}$ and v_0 the clock valuation that assigns value 0 to all clocks in $\mathcal{A}$. We can now formalize what it means for a timed automaton to satisfy a TCTL-formula:

Definition 48. (**Satisfiability for a timed automaton**)
For TCTL-formula ϕ and timed automaton $\mathcal{A}$ let

$$\mathcal{A} \models \phi \text{ if and only if } \mathcal{M}(\mathcal{A}), (s_0, w_0) \models \phi$$

where $w_0(y) = 0$ for all formula clocks y.

The model checking problem that we are interested in is to check whether a given timed automaton $\mathcal{A}$ satisfies some TCTL-formula ϕ. According to the last definition, this amounts to check whether $\mathcal{M}(\mathcal{A}), (s_0, w_0) \models \phi$. The basis for the model checking of $\mathcal{A}$ is thus the transition system $\mathcal{M}(\mathcal{A})$. The main problem, however, is that the state space of $\mathcal{M}(\mathcal{A})$, the set $L \times V(C)$, is *infinite*! This is exemplified by means of the following example:



How can we effectively check whether a given formula is valid for $\mathcal{M}(\mathcal{A})$, if an infinite number of states has to be considered?

From the above treatment of timed automata it is evident that the number of states of a timed automaton is infinite. The key idea by Alur and Dill (1990) that facilitates model checking of timed automata (and similar structures) is to introduce an appropriate equivalence relation, $\approx$ say, on clock valuations such that

Correctness: equivalent clock valuations satisfy the same TCTL-formulas, and

Finiteness: the number of equivalence classes under $\approx$ is finite.

The second property allows to replace an infinite set of clock valuations by a finite set of (sets of) clock valuations. In addition, the first property guarantees that for model $\mathcal{M}$ these clock valuations satisfy the same TCTL-formulas. That is to say, there does not exist a TCTL-formula that distinguishes the infinite-state and its equivalent finite-state clock valuation. Formally, for any timed automaton $\mathcal{A}$, this amounts to:

$$\mathcal{M}(\mathcal{A}), ((l, v), w) \models \phi \text{ if and only if } \mathcal{M}(\mathcal{A}), ((l, v'), w') \models \phi$$

for $v \cup w \approx v' \cup w'$.

The key observation that leads to the definition of this equivalence relation $\approx$ is that paths of timed automaton $\mathcal{A}$ starting at states which

- agree on the integer parts of all clock values, and
- agree on the ordering of the fractional parts of all clocks

are very similar. In particular, since time constraints in TCTL-formulas only refer to natural numbers, there is no TCTL-formula that distinguishes between these "almost similar" runs. Roughly speaking, two clock valuations are equivalent if they satisfy the aforementioned two constraints. Combined with the observation that

- if clocks exceed the maximal constant with which they are compared, their precise value is not of interest

the amount of equivalence classes thus obtained is not only denumerable, but also finite!

Since the number of equivalence classes is finite, and the fact that equivalent TCTL-models of timed automata satisfy the same formulas, this suggests to perform model checking on the basis of these classes. The finite-state model that thus results from timed automaton $\mathcal{A}$ is called its *region automaton*. Accordingly, equivalence classes under $\approx$ are called *regions*. Model checking a timed automaton thus boils down to model checking its associated finite region automaton. This reduced problem can be solved in a similar way as we have seen for model checking the branching temporal logic CTL in the previous chapter — by iteratively labelling states with sub-formulas of the property to be checked, as we will see later on. Thus, roughly speaking

Model checking a timed automaton against a TCTL*-formula amounts to model checking its region automaton against a* CTL*-formula.*

In summary we obtain the scheme in Table 4.1 for model checking the TCTL-formula ϕ over the timed automaton $\mathcal{A}$. Here we denote the equivalence class of clock valuation v under $\approx$ by $[v]$. (As we will see later on, the region automaton also depends on the clock constraints in the formula to be checked, but this dependency is omitted here for reasons of simplicity.)

1.	*Determine the equivalence classes (i.e. regions) under* $\approx$
2.	*Construct the region automaton* $\mathcal{R}(\mathcal{A})$
3.	*Apply the* CTL*-model checking procedure on* $\mathcal{R}(\mathcal{A})$
4.	$\mathcal{A} \models \phi$ *if and only if* $[s_0, w_0] \in Sat^R(\phi)$.

Table 4.1: Basic recipe for model checking TCTL over timed automata

In the next section we address the construction of the region automaton $\mathcal{R}(\mathcal{A})$. We start by discussing in detail the notion of equivalence $\approx$ indicated above. Since this notion of equivalence is of crucial importance for the approach we extensively justify its definition by a couple of examples.

In the sequel we adopt the following notation for clock valuations. For clock valuation $v : C \longrightarrow \mathbb{R}^+$ and clock $x \in C$ let $v(x) = \lfloor x \rfloor + \mathit{frac}(x)$, that is, $\lfloor x \rfloor$ is the integral part of the clock value of x and $\mathit{frac}(x)$ the fractional part. For instance, for $v(x) = 2.134$ we have $\lfloor x \rfloor = 2$ and $\mathit{frac}(x) = 0.134$. The justification of the equivalence relation $\approx$ on clock valuations proceeds by four observations, that successively lead to a refinement of the notion of equivalence. Let v and v' be two clock valuations defined on the set of clocks C.

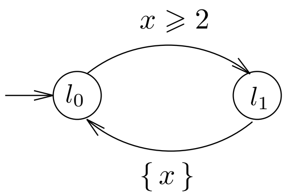


Figure 4.4: Clock valuations that agree on integer parts are equivalent

First observation: consider the timed automaton depicted in Figure 4.4 and the following states of this automaton: (l_0, v) and (l_0, v') with $v(x) = 3.2$ and $v'(x) = 3.7$. Clearly, in both states the edge from location l_0 to l_1 is enabled for any time instant exceeding 2. The fractional parts of $v(x)$ and $v'(x)$ do not determine its enabled-ness. Similarly, if $v(x) = 1.2$ and $v'(x) = 1.7$ the edge is disabled for both clock valuations and once more the fractional parts are irrelevant. Since in general clock constraints are of the form $x \sim c$ with $c \in \mathbb{N}$, only the integral parts of the clocks seem to be important. This leads to the first suggestion for clock equivalence:

$$v \approx v' \text{ if and only if } \lfloor v(x) \rfloor = \lfloor v'(x) \rfloor \text{ for all } x \in C. \tag{4.1}$$

This notion is rather simple, leads to a denumerable (but still infinite) number of equivalence classes, but is too coarse. That is, it identifies clock valuations that can be distinguished by means of TCTL-formulas.

Second observation: consider the timed automaton depicted in Figure 4.5 and the following states of this automaton: $s = (l_0, v)$ and $s' = (l_0, v')$ with $v(x) = 0.4$,

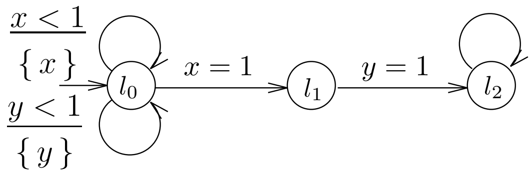


Figure 4.5: The ordering of clock valuations of different clocks is relevant

$v'(x) = 0.2$ and $v(y) = v'(y) = 0.3$. According to suggestion (4.1) we would have $v \approx v'$, since $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor = 0$ and similarly for clock y. But from state s the location l_2 can be reached, whereas this is not possible starting from state s'. This can be seen as follows. Consider s at 6 fractions of a time-unit later. Its clock valuation $v{+}0.6$ reads $v(x) = 1$ and $v(y) = 0.9$. Clearly, in this clock valuation the edge leading from l_0 to l_1 is enabled. If, after taking this edge, successively in location l_1 time progresses by 0.1 time-units, then the edge leading to location l_2 is enabled. A similar scenario does not exist for s'. In order to reach from s' location l_1 clock x needs to be increased by 0.8 time-units. But then $v'(y)$ equals 1.1, and the edge leading to l_2 will be permanently disabled. The important difference between v and v' is that $v(x) \geqslant v(y)$, but $v'(x) \leqslant v'(y)$. This suggests the following extension to suggestion (4.1):

$$v(x) \leqslant v(y) \text{ if and only if } v'(x) \leqslant v'(y) \text{ for all } x, y \in C. \tag{4.2}$$

Again, the resulting equivalence leads to a denumerable number of equivalence classes, but it is still too coarse.

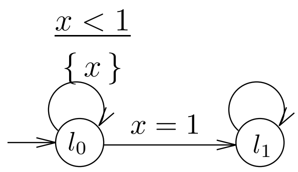


Figure 4.6: Integer-valued clocks should match

Third observation: consider the timed automaton of Figure 4.6 and the following states $s = (l_0, v)$ and $s' = (l_0, v')$ with $v(x) = 0$ and $v'(x) = 0.1$. According to suggestions (4.1) and (4.2) we would have $v \approx v'$, but, for instance, $s, w \models \mathsf{EF}_{=1}\, p$, but $s, w not \models \mathsf{EF}_{=1}\, p$, where p is an atomic proposition that is only valid in location l_1. The main difference between v and v' is that clock x in s is exactly

0 whereas in state s' it just passed 0. This suggests to extend suggestions (4.1) and (4.2) with

$$\mathit{frac}(v(x)) = 0 \text{ if and only if } \mathit{frac}(v'(x)) = 0 \text{ for all } x \in C. \tag{4.3}$$

The resulting equivalence is not too coarse anymore, but still leads to an infinite number of equivalence classes and not to a finite one.

Fourth observation: let c_x be the maximum constant with which clock x is compared in a clock constraint (that occurs either in an enabling condition associated to an edge, or in an invariant) in the timed automaton at hand. Since c_x is the largest constant with which x is compared it follows that if $v(x) > c_x$ then the actual value of x is irrelevant; the fact that $v(x) > c_x$ suffices to decide the enabled-ness of all edges in the timed automaton.

In combination with the three conditions above, this observation leads to a *finite* number of equivalence classes: only the integral parts of clocks are of importance up to a certain bound ((4.1) plus the current observation) plus their fractional ordering (4.2) and the fact whether fractional parts are zero or not (4.3). These last two points now also are of importance only if the clocks have not exceed their bound. For instance, if $v(y) > c_y$ then the ordering with clock x is not of importance, since the value of y will not be tested anymore anyway.

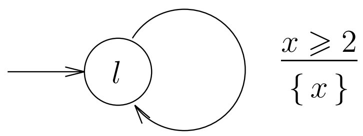
Finally, we come to the following definition of clock equivalence (Alur and Dill, 1994).

Definition 49. (**Clock equivalence**)
Let $\mathcal{A}$ be a timed automaton with set of clocks C and $v, v' \in V(C)$. Then $v \approx v'$ if and only if

1. $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ or $v(x) > c_x$ and $v'(x) > c_x$, for all $x \in C$, and
2. $\mathit{frac}(v(x)) \leqslant \mathit{frac}(v(y))$ iff $\mathit{frac}(v'(x)) \leqslant \mathit{frac}(v'(y))$ for all $x, y \in C$ with $v(x) \leqslant c_x$ and $v(y) \leqslant c_y$, and
3. $\mathit{frac}(v(x)) = 0$ iff $\mathit{frac}(v'(x)) = 0$ for all $x \in C$ with $v(x) \leqslant c_x$.

This definition can be modified in a straightforward manner such that $\approx$ is defined on a set of clocks C' such that $C \subseteq C'$. This would capture the case where C' can also included formula clocks (beside clocks in the automaton). For formula clock z in ϕ, the property at hand, c_z is the largest constant with which z is compared in ϕ. For instance, for formula z in $(AF[(p \wedge z \leqslant 3) \vee (q \wedge z > 5)])$, $c_z = 5$.

Example 37. Consider the simple timed automaton depicted by:



This automaton has a set of clocks $C = \{ x \}$ *with* $c_x = 2$*, since the only clock constraint is* $x \geqslant 2$*. We gradually construct the regions of this automaton by considering each constraint of the definition of* $\approx$ *separately. Clock valuations* v *and* v' *are equivalent if* $v(x)$ *and* $v'(x)$ *belong to the same equivalence class on the real line. (In general, for* n *clocks this amounts to considering an* n*-dimensional hyper-space on* $\mathbb{R}^+$*.) For convenience let* $[x \sim c]$ *abbreviate* $\{ x \mid x \sim c \}$ *for natural* c *and comparison operator* $\sim$*.*

1. *The requirement that* $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ *leads to the following partition of the real line:*

$$[0 \leqslant x < 1], [1 \leqslant x < 2], [2 \leqslant x < 3], [3 \leqslant x < 4], \ldots$$

2. *Since* $c_x = 2$*, it is not interesting for the timed automaton at hand to distinguish between the valuations* $v(x) = 3$ *and* $v'(x) = 27$*. In summary, the equivalence classes that result from considering the first constraint of Definition 49 is*

$$[0 \leqslant x < 1], [1 \leqslant x < 2], [x = 2], \text{ and } [x > 2]$$

3. *According to the second constraint, the ordering of clocks should be maintained. In our case, this constraint trivially holds since there is only one clock. So, there are no new equivalence classes introduced by considering this constraint.*

4. *The constraint that* $\mathrm{frac}(v(x)) = 0$ *iff* $\mathrm{frac}(v'(x)) = 0$ *if* $v(x) \leqslant c_x$ *leads to partitioning, for instance, the equivalence class* $[0 \leqslant x < 1]$*, into the classes* $[x = 0]$ *and* $[0 < x < 1]$*. Similarly, the class* $[1 \leqslant x < 2]$ *is partitioned. The class* $[x > 2]$ *is not partitioned any further since for this class the condition* $v(x) \leqslant c_x$ *is violated. As a result we obtain for the simple timed automaton the following 6 equivalence classes:*

$$[x = 0], [0 < x < 1], [x = 1], [1 < x < 2], [x = 2], \text{ and } [x > 2].$$

(End of example.)

Example 38. *Consider the set of clocks* $C = \{ x, y \}$ *with* $c_x = 2$ *and* $c_y = 1$*. In Figure 4.7 we show the gradual construction of the regions by considering each constraint of the definition of* $\approx$ *separately. The figure depicts a partition of the two-dimensional space* $\mathrm{I\!R}^+ \times \mathrm{I\!R}^+$*. Clock valuations* v *and* v' *are equivalent if the real-valued pairs* $(v(x), v(y))$ *and* $(v'(x), v'(y))$ *are elements of the same equivalence class in the hyper-space.*

1. *The requirement that* $\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor$ *for all clocks in* C *leads, for instance, to the equivalence classes* $[(0 \leqslant x < 1), (0 \leqslant y < 1)]$ *and* $[(1 \leqslant x < 2), (0 \leqslant y < 1)]$ *and so on. The constraint that* $v(x) > c_x$ *and* $v'(x) > c_x$ *for all clocks in* C *leads to the equivalence class* $[(x > 2), (y > 1)]$*. This means that for any clock valuation* v *for which* $v(x) > 2$ *and* $v(y) > 1$ *the exact values of* x *and* y *are irrelevant. The obtained equivalence classes are:*

$$\begin{array}{ll}
[(0 \leqslant x < 1), (0 \leqslant y < 1)] & [(1 \leqslant x < 2), (0 \leqslant y < 1)] \\
[(0 \leqslant x < 1), (y = 1)] & [(1 \leqslant x < 2), (y = 1)] \\
[(0 \leqslant x < 1), (y > 1)] & [(1 \leqslant x < 2), (y > 1)] \\
[(x = 2), (0 \leqslant y < 1)] & [(x > 2), (0 \leqslant y < 1)] \\
[(x = 2), (y = 1)] & [(x > 2), (y = 1)] \\
[(x = 2), (y > 1)] & [(x > 2), (y > 1)]
\end{array}$$

These 12 classes are depicted in Figure 4.7(a).

2. *Consider the equivalence class* $[(0 \leqslant x < 1), (0 \leqslant y < 1)]$ *that we obtained in the previous step. Since the ordering of the clocks now becomes important,*

this equivalence class is split into the classes $[(0 \leqslant x < 1), (0 \leqslant y < 1), (x < y)]$, $[(0 \leqslant x < 1), (0 \leqslant y < 1), (x = y)]$, *and* $[(0 \leqslant x < 1), (0 \leqslant y < 1), (x > y)]$. *A similar reasoning applies to the class* $[(1 \leqslant x < 2), (0 \leqslant y < 1)]$. *Other classes are not further partitioned. For instance, class* $[(0 \leqslant x < 1), (y = 1)]$ *does not need to be split, since the ordering of clocks* x *and* y *in this class is fixed,* $v(x) \leqslant v(y)$. *Class* $[(1 \leqslant x < 2), (y > 1)]$ *is not split, since for this class the condition* $v(x) \leqslant c_x$ *and* $v_y \leqslant c_y$ *is violated. Figure 4.7(b) shows the resulting equivalence classes.*

3. *Finally, we apply the third criterion of Definition 49. As an example consider the class* $[(0 \leqslant x < 1), (0 \leqslant y < 1), (x = y)]$ *that we obtained in the previous step. This class is now partitioned into* $[(x = 0), (y = 0)]$ *and* $[(0 < x < 1), (0 < y < 1), (x = y)]$. *Figure 4.7(c) shows the resulting 28 equivalence classes: 6 corner points, 14 open line segments, and 8 open regions.*

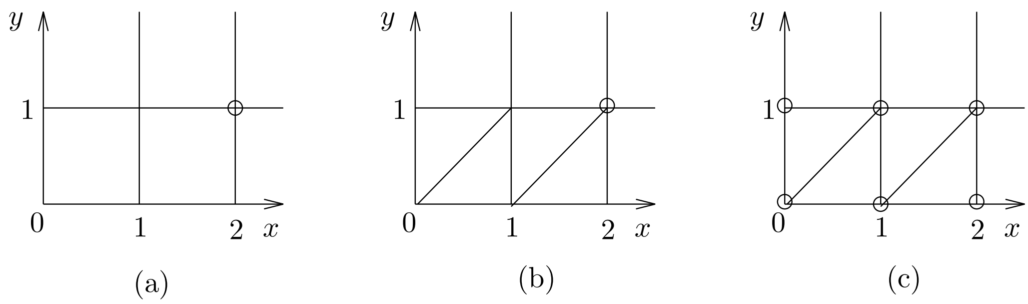


Figure 4.7: Partitioning of $\mathbb{R}^+ \times \mathbb{R}^+$ according to $\approx$ for $c_x = 2$ and $c_y = 1$

(End of example.)

4.7 Region automata

The equivalence classes under $\approx$ will be the basis for model checking timed automata. The combination of such class with a location is called a *region*.

Definition 50. (**Region**)
A *region* r is a pair $(l, [v])$ with location $l \in L$ and valuation $v \in V(C)$.

In the sequel we abbreviate $(l, [v])$ by $[s]$ for $s = (l, v)$. In addition we write $[s, w]$ as a shorthand for $(l, [v \cup w])$ for w a clock valuation over formula clocks.

Since there are finitely many equivalence classes under $\approx$, and any timed automaton possesses only a finite number of locations, the number of regions is finite. The next result says that states belonging to the same region satisfy the same TCTL-formulas. This important result for real-time model checking is due to Alur, Dill & Courcoubetis (1993).

Theorem 51.
Let $s, s' \in S$ such that $s, w \approx s', w'$. For any TCTL-formula ϕ, we have:

$$\mathcal{M}(\mathcal{A}), (s, w) \models \phi \text{ if and only if } \mathcal{M}(\mathcal{A}), (s', w') \models \phi.$$

According to this result we do not have to distinguish the equivalent states s and s', since there is no TCTL-formula that can distinguish between the two. This provides the correctness criterion for using equivalence classes under $\approx$, i.e. regions, as basis for model checking. Using regions as states we construct a finite-state automaton, referred to as the *region automaton*. This automaton consists of regions as states and transitions between regions. These transitions either correspond to the evolvement of time, or to the edges in the original timed automaton.

We first consider the construction of a region automaton by means of an example. Two types of transitions can appear between augmented regions. They are either due to (1) the passage of time (depicted as solid arrows), or (2) transitions of the timed automaton at hand (depicted as dotted arrows).

Example 39. *Let us consider our simple timed automaton with a single edge, cf. Figure 4.8(a). Since the maximum constant that is compared with x is 2, it follows $c_x = 2$. The region automaton is depicted in Figure 4.8(b). Since there is only one location, in each reachable region the timed automaton is in location l. The region automaton contains two transitions that correspond to the edge of the timed automaton. They both reach the initial region* **A**. *The dotted transitions from* **E** *and* **F** *to* **A** *represent these edges. Classes in which there is a possibility* *to let time pass without bound while remaining in that class, are called unbounded classes, see below. In this example there is a single unbounded class, indicated by a grey shading, namely* **F**. *In* **F**, *the clock x can grow without bound while staying in the same region. (The reader is invited to investigate the changes to the region automaton when the simple timed automaton is changed as in Figure 4.1(c) and (e).)*

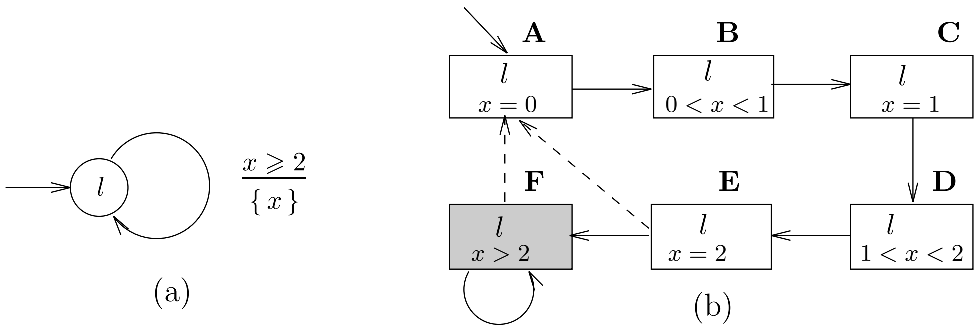


Figure 4.8: Region automaton of a simple timed automaton

Suppose now that we would like to model check a TCTL*-formula over the timed automaton of Figure 4.8(a) that contains a single formula clock z with $c_z = 2$, i.e. clock z is not compared in the formula with a constant larger than 2. The region automaton over $\{ x, z \}$ is depicted in Figure 4.9. Notice that it is in fact the region automaton of before (cf. Figure 4.8(b)) extended with two "copies" of it: regions* **G** *through* **L** *and regions* **M** *through* **R**. *These "copies" are introduced for the constraints $z-x = 2$ and $z-x > 2$. Notice that the formula clock z is never reset. This is typical for a formula clock as there is no means in a formula to reset clocks once they are introduced.* *(End of example.)*

Definition 52. (**Delay-successor region**)
Let r, r' be two distinct regions (i.e. $r \neq r'$). r' is the *delay successor* of r, denoted $r' = \mathit{delsucc}(r)$, if there exists a $d \in \mathbb{R}^+$ such that for each $r = [s]$ we have $s \xrightarrow{d} s'$ and $r' = [s{+}d]$ and for all $0 \leqslant d' < d : [s{+}d'] \in r \cup r'$.

Here, for $s = (l, v)$, state $s{+}d$ denotes $(l, v{+}d)$. In words: $[s{+}d]^*$ is the delay-successor of $[s]^*$ if for some *positive* d any valuation in $[s]^*$ moves to a valuation in $[s{+}d]^*$, without having the possibility to leave these two regions at any earlier

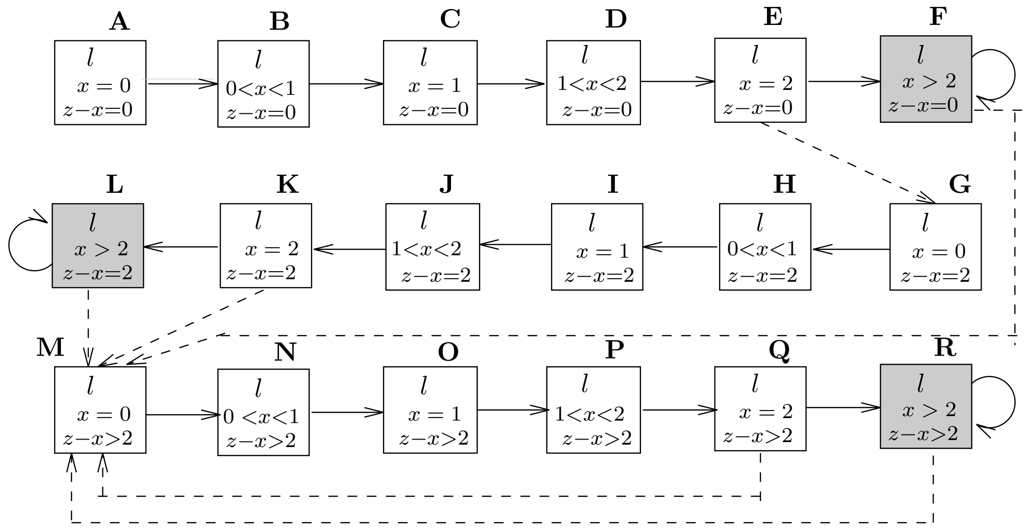


Figure 4.9: Region automaton of a simple timed automaton for formula clock z with $c_z = 2$

point in time. Observe that for each region there is at most one delay-successor. This should not surprise the reader, since delay-successors correspond to the advancing of time, and time can advance only deterministically.

Example 40. *Consider the equivalence classes of the timed automaton of Figure 4.8(a). The regions containing clock x and extra clock z (cf. Figure 4.9) are partitions of the two-dimensional real-space. Timed transitions correspond to upward moves along the diagonal line $x = z$. For instance, $[x = z = 1]$ is the delay-successor of $[(0 < x < 1), (x = z)]$, and the delay-successor of $[(1 < x < 2), (x = z)]$ has successor $[x = z = 2]$. Class $[(x = 1), (z > 2)]$ is not the delay-successor of $[(0 < x < 1), (z = 2)]$ — a unreachable region that is not depicted — since there is some real value d' such that $[(0 < x < 1), (z > 2)]$ is reached in between.* *(End of example.)*

Definition 53. (**Unbounded region**)
Region r is an *unbounded region* if for all clock valuations v such that $r = [v]$ we have $v(x) > c_x$ for all $x \in C$.

In an unbounded region all clocks have exceed the maximum constant with which they are compared, and hence all clocks may grow without bound. In

Figure 4.8(b) and 4.9 we have indicated the unbounded region by a grey shading of the region.

A region automaton now consists of a set of states (the regions), an initial state, and two transition relations: one corresponding to delay transitions, and one corresponding to the edges of the timed automaton at hand.

Definition 54. (**Region automaton**)
For timed automaton $\mathcal{A}$ and set of time constraints Ψ (over C and the formula clocks), the *region automaton* $\mathcal{R}(\mathcal{A}, \Psi)$ is the transition system $(R, r_0, \longrightarrow)$ where

- $R = S/_{\approx} = \{ [s] \mid s \in S \}$
- $r_0 = [s_0]$
- $r \longrightarrow r'$ if and only if $\exists\, s, s'.(r = [s] \;\wedge\; r' = [s'] \;\wedge\; s \overset{*}{\longrightarrow} s')$
- $r \longrightarrow r'$ if and only if
  1. r is an unbounded region and $r = r'$, or
  2. $r \neq r'$ and $r' = \mathit{delsucc}(r)$.

Observe that from this definition it follows that unbounded regions are the only regions that have a self-loop consisting of a delay transition.

Example 41. To illustrate the construction of a region automaton we consider a

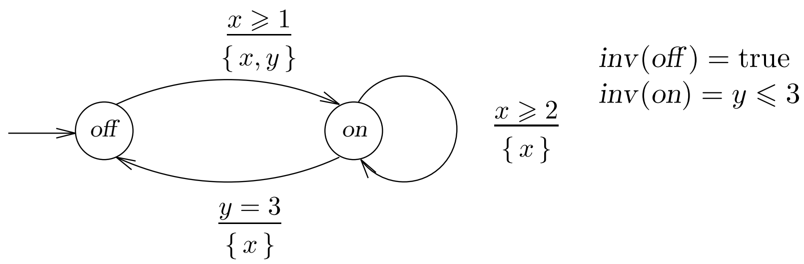


Figure 4.10: Timed automaton for modified light switch

slight variant of the light switch of Example 31, that is depicted in Figure 4.10. In order to show the effect of invariants we have equipped location on with invariant

$y \leqslant 3$. In order to keep size of the region automaton manageable we have adapted the enabling conditions of the transitions that change location; in particular we have made the constants smaller.

The region automaton $\mathcal{R}(\mathcal{A}, \Psi)$ where $\mathcal{A}$ is the modified light switch, is depicted in Figure 4.11. The set of time constraints is $\Psi = \{ x \geqslant 1, x \geqslant 2, y = 3, y \leqslant 3 \}$, i.e. the set of all enabling constraints and invariants in $\mathcal{A}$. For simplicity no formula clocks are considered. We have for example $\mathbf{D} \longrightarrow \mathbf{E}$ since there is a transition from state $s = (\mathit{off}, v)$ with $v(x) = v(y) > 1$ to state $s' = (\mathrm{on}, v')$ with $v'(x) = v'(y) = 0$ and $\mathbf{D} = [s]$, $\mathbf{E} = [s']$. There is a delay transition $\mathbf{D} \longrightarrow \mathbf{D}$, since the region $[v]$ where $v(x) = v(y) > 1$ is an unbounded region. This stems from the fact that in location off time can gro without any bound, i.e. $\mathrm{inv}(\mathit{off}) = \mathit{true}$.

The reader is invited to check how the region automaton has to be changed when changing $\mathrm{inv}(\mathit{off})$ into $y \leqslant 4$. *(End of example.)*

4.8 Model checking region automata

Given the region automaton $\mathcal{R}(\mathcal{A}, \Psi)$ for timed automaton $\mathcal{A}$ and TCTL-formula ϕ with clock constraints Ψ in $\mathcal{A}$ and ϕ, the model checking algorithm now proceeds as for untimed CTL, see previous chapter. Let us briefly summarise the idea of labelling. The basic idea of the model checking algorithm is to *label* each state (i.e. augmented region) in the region automaton with the sub-formulas of ϕ that are valid in that state. This labelling procedure is performed iteratively starting by labelling the states with the sub-formulas of length 1 of ϕ, i.e. the atomic propositions (and true and false) that occur in ϕ. In the $(i{+}1)$-th iteration of the labelling algorithm sub-formulas of length $i{+}1$ are considered and the states are labelled accordingly. To that purpose the labels already assigned to states are used, being sub-formulas of ϕ of length at most i ($i \geqslant 1$). The labelling algorithm ends by considering the sub-formula of length $\mid \phi \mid$, ϕ itself. This algorithm is listed in Table 4.2.

The model checking problem $\mathcal{A} \models \phi$, or, equivalently, $\mathcal{M}(\mathcal{A}), (s_0, w_0) \models \phi$, is

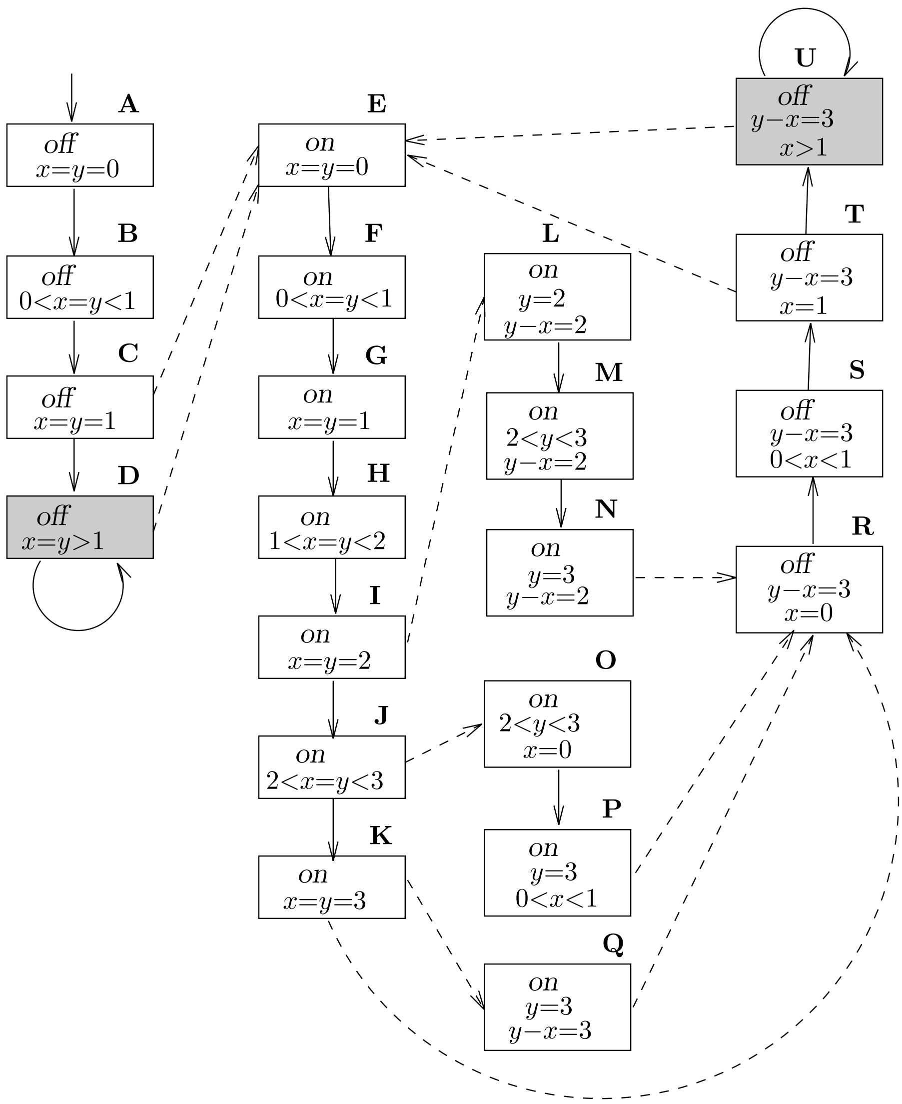


Figure 4.11: Region automaton of the modified light switch

function $Sat^R(\phi : Formula)$: **set of** *Region*;
(* precondition: true *)
begin
    if $\phi = \text{true} \longrightarrow$ **return** $S/_{\approx}$
    [] $\phi = \text{false} \longrightarrow$ **return** $\varnothing$
    [] $\phi \in AP \longrightarrow$ **return** $\{ [s, w]_{\approx} \mid \phi \in Label(s) \}$
    [] $\phi = \alpha \longrightarrow$ **return** $\{ [s, w]_{\approx} \mid s = (l, v) \ \wedge \ v \cup w \models \alpha \}$
    [] $\phi = \neg \phi_1 \longrightarrow$ **return** $S/_{\approx} - Sat^R(\phi_1)$
    [] $\phi = \phi_1 \ \vee \ \phi_2 \longrightarrow$ **return** $(Sat^R(\phi_1) \cup Sat^R(\phi_2))$
    [] $\phi = z.\phi_1 \longrightarrow$ **return** $\{ [s, w]_{\approx} \mid (s, [z]w) \in Sat^R(\phi_1) \}$
    [] $\phi = \mathsf{E}\,[\phi_1 \,\mathsf{U}\, \phi_2] \longrightarrow$ **return** $Sat^R_{EU}(\phi_1, \phi_2)$
    [] $\phi = \mathsf{A}\,[\phi_1 \,\mathsf{U}\, \phi_2] \longrightarrow$ **return** $Sat^R_{AU}(\phi_1, \phi_2)$
    fi
(* postcondition: $Sat^R(\phi) = \{ [s, w]_{\approx} \mid \mathcal{M}, (s, w) \models \phi \}$ *)
end

Table 4.2: Outline of main algorithm for model checking TCTL

now solved by considering the labelling:

$$\mathcal{M}(\mathcal{A}), (s_0, w_0) \models \phi \text{ if and only if } [s_0, w_0] \in Sat^R(\phi).$$

(The correctness of this statement follows from Theorem 55).

The computation of $Sat^R(\phi)$ is done by considering the syntactical structure of ϕ. The cases for the propositional formulas (true, false, negation, disjunction and atomic propositions) is identical to the case for CTL, cf. Chapter 3. For clock constraint α, $Sat^R(\alpha)$ is the set of regions $[s, w]$ such that α is satisfied by w and v, the clock valuation of the automata clocks in s. Region $[s, w]$ belongs to set $Sat^R(z \textsf{ in } \phi)$ if it satisfies ϕ for w' where $w'(z) = 0$ (start the formula clock at value 0) and $w'(x) = w(x)$ for $x \neq z$. For the until-formulas, auxiliary functions are defined. These functions are listed in Table 4.3 and 4.4. The code for the function Sat^R_{EU} is identical to the untimed case. For Sat^R_{AU} there is small, though essential, difference with the untimed algorithm for $\mathsf{A}\,[\phi \,\mathsf{U}\, \psi]$. In analogy to the case for CTL, the iteration would contain the statement:

$$Q := Q \cup \big(\{ s \mid \forall s' \in Q.\, s \longrightarrow s' \} \cap Sat^R(\phi)\big)\,.$$

The correctness of this statement, however, assumes that each state s has some successor under $\longrightarrow$. This is indeed valid for CTL, since in a CTL-model each state has at least one successor. In region automata, regions may exist that have no outgoing transition, neither a delay transition nor a transition that corresponds to an edge in the timed automaton. Therefore, we take a different approach and extend Q with those regions that have at least one transition into Q, and for which all direct successors are in Q. This yields the code listed in Table 4.4. An example of a region automaton containing a region without any successors is depicted in Figure 4.12. Region **C** does not have any successors: delaying is not possible due to the invariant of l, and there is no edge that is enabled for valuation $x = 1$.

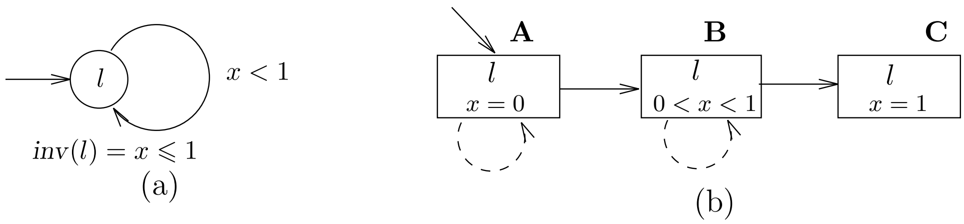


Figure 4.12: Region automaton with a region without successors

Example 42. Consider the region automaton (cf. Figure 4.11) of the modified light switch, and suppose location on *is labelled with atomic proposition q and* off *with p. We can now check that it is impossible to reach* on *from* off *in less than one time-unit:*

$$\mathcal{M}(\mathcal{A}), (s_0, w_0) \not\models \mathsf{E}\,[p\, \mathsf{U}_{<1}\, q]$$

since there is no path through the region automaton that starts in **A** *(which corresponds to $[s_0, w_0]$) to some region* **E** *through* **K***, where the light is switched on, and that lasts less than one time-unit. Formally, this can be seen as follows. Formula $\mathsf{E}\,[p\, \mathsf{U}_{<1}\, q]$ abbreviates z* in *$\mathsf{E}\,[(p \,\wedge\, z < 1)\, \mathsf{U}\, q]$. Equip the region automaton*

function $Sat^R_{EU}\,(\phi, \psi : \mathit{Formula}) :$ **set of** *Region*;
(* precondition: true *)
begin var $Q, Q' :$ **set of** *Region*;
  $Q, Q' := Sat^R(\psi), \varnothing;$
  do $Q \neq Q' \longrightarrow$
    $Q' := Q;$
    $Q := Q \,\cup\, \big(\{\, s \mid \exists\, s' \in Q.\, s \longrightarrow s' \,\} \,\cap\, Sat^R(\phi)\big)$
  od;
  return Q
(* postcondition: $Sat_{EU}(\phi, \psi) = \{\, [s, w] \mid s, w \models \mathsf{E}\,[\phi\,\mathsf{U}\,\psi] \,\}$ *)
end

Table 4.3: Model checking $\mathsf{E}\,[\phi\,\mathsf{U}\,\psi]$ over regions

function $Sat^R_{AU}\,(\phi, \psi : \mathit{Formula}) :$ **set of** *Region*;
(* precondition: true *)
begin var $Q, Q' :$ **set of** *Region*;
  $Q, Q' := Sat^R(\psi), \varnothing;$
  do $Q \neq Q' \longrightarrow$
    $Q' := Q;$
    $Q := Q \,\cup\, \big(\{\, s \mid \exists\, s' \in Q.\, s \longrightarrow s' \;\wedge\; (\forall\, s' \in Q.\, s \longrightarrow s') \,\} \,\cap\, Sat^R(\phi)\big)$
  od;
  return Q
(* postcondition: $Sat_{AU}(\phi, \psi) = \{\, [s, w] \mid s, w \models \mathsf{A}\,[\phi\,\mathsf{U}\,\psi] \,\}$ *)
end

Table 4.4: Model checking $\mathsf{A}\,[\phi\,\mathsf{U}\,\psi]$ over regions

with a formula clock z, that is reset in **A**, *and keeps advancing ad infinitum. The earliest time to switch the light on (i.e. to make q valid) is to take the transition from* **C** *to* **E**. *However, then z equals 1, and thus the property is invalid. (The reader is invited to check this informal reasoning by applying the algorithms to the region automaton.)* *(End of example.)*

Recall that a timed automaton is non-Zeno if from every state there is at least one time-divergent path, a path where time grows ad infinitum, does exist. Non-Zenoness is a necessary pre-condition for the model checking procedures given above, as follows from the following result (Yovine, 1998).

Theorem 55.
For $\mathcal{A}$ a non-Zeno timed automaton: $(s, w) \in Sat(\phi)$ if and only if $[s, w] \in Sat^R(\phi)$.

Thus the correctness of the model checking algorithm is only guaranteed if the timed automaton at hand is non-Zeno. Non-Zenoness of $\mathcal{A}$ can be checked by showing that from each state of $\mathcal{M}(\mathcal{A})$ time can advance by exactly one time-unit:

Theorem 56.
$\mathcal{A}$ is non-Zeno if and only if for all states $s \in S : \mathcal{M}(\mathcal{A}), s \models \mathsf{EF}_{=1}$ true.

Time complexity

The model checking algorithm labels the region automaton $\mathcal{R}(\mathcal{A}, \Psi)$ for timed automaton $\mathcal{A}$ and set of clock constraint Ψ. We know from Chapter 3 that the worst-case time complexity of such algorithm is proportional to $|\,\phi\,| \times |\,S\,|^2$, where S is the set of states of the automaton to be labelled. The number of states in $\mathcal{R}(\mathcal{A}, \Psi)$ equals the number of regions of $\mathcal{A}$ with respect to Ψ. The number of regions is proportional to the product of the number of locations in $\mathcal{A}$ and the number of equivalence classes under $\approx$. Let Ψ be a set of clock constraints over C' where $C \subseteq C'$, i.e. C' contains the clocks of $\mathcal{A}$ plus some formula clocks. For

$n = |\, C' \,|$, the number of regions is

$$\mathcal{O}\left(n! \times 2^n \times \prod_{x \in \Psi} c_x \times |\, L \,| \right).$$

Thus,

> *The worst-case time complexity of model checking* TCTL*-formula* ϕ *over timed automaton* $\mathcal{A}$*, with the clock constraints of* ϕ *and* $\mathcal{A}$ *in* Ψ *is:*
> $\mathcal{O}\left(|\, \phi \,| \times (n! \times 2^n \times \prod_{x \in \Psi} c_x \times |\, L \,|^2) \right).$

That is, model checking TCTL is

(i) linear in the length of the formula ϕ

(ii) exponential in the number of clocks in $\mathcal{A}$ and ϕ

(iii) exponential in the maximal constants with which clocks are compared in $\mathcal{A}$ and ϕ.

Using the techniques that we have discussed in Chapter 3, the time complexity can be reduced to being quadratic in the number of locations.

The lower-bound for the complexity of model checking TCTL for a given timed automaton is known to be PSPACE-hard (Alur, Courcoubetis & Dill, 1993). This means that at least a memory is needed of a size that is polynomial in the size of the system to be checked.

Fairness

As for untimed systems, in verifying time-critical systems we are often only interested in considering those executions in which enabled transitions are taken in a fair way, see Chapter 3. In order to be able to deal with fairness, we have seen that for the case with CTL the logic is interpreted over a fair CTL-model. A fair model has a new component, a set $\mathcal{F} \subseteq 2^L$ of fairness constraints. An analogous recipe can be followed for TCTL. We will not work out all details here, but the basic idea is to enrich a TCTL-model with a set of fairness constraints, in a similar way as for CTL. A fair path of a timed automaton is defined as a path in which for every set $F_i \in \mathcal{F}$ there are infinitely many locations in the run that are in F_i (like a generalized Büchi acceptance condition). The semantics of TCTL is now similar to the semantics provided earlier in this chapter, except that all path quantifiers are interpreted over $\mathcal{F}$-fair paths rather than over all paths. In an analogous manner as before, the region automaton is constructed. The fairness is incorporated in the region automaton by replacing each $F_i \in \mathcal{F}$ by the set $\{ (l, [v]) \mid l \in F_i \}$. The labelling algorithms can now be adapted to cover only $\mathcal{F}$-fair paths in the region automaton. Due to the consideration of fairness constraints, the worst-case time complexity of the labelling algorithm increases by a multiplicative factor that is proportional to the cardinality of $\mathcal{F}$.

Overview of TCTL model checking

We conclude this section by providing an overview of model checking a timed automaton against a TCTL-formula. Notice that the region automaton depends only on the clock constraints (i.e. the formula and automata clocks and the maximal constants with which clocks are compared) appearing in the formula ϕ, it does not depend on the formula itself. The labelled region automaton, of course, depends on the entire formula ϕ. The reader is invited to check the resemblance of this schema with the overview of CTL model checking in Chapter 3.

4.9 The model checker UPPAAL

UPPAAL is a tool suite for symbolic model checking of real-time systems (Larsen, Pettersson & Yi, 1997) developed at the University of Uppsala (Sweden) and Aalborg (Denmark). Besides model checking, it also supports simulation of timed automata and has some facilities to detect deadlocks. UPPAAL has been applied

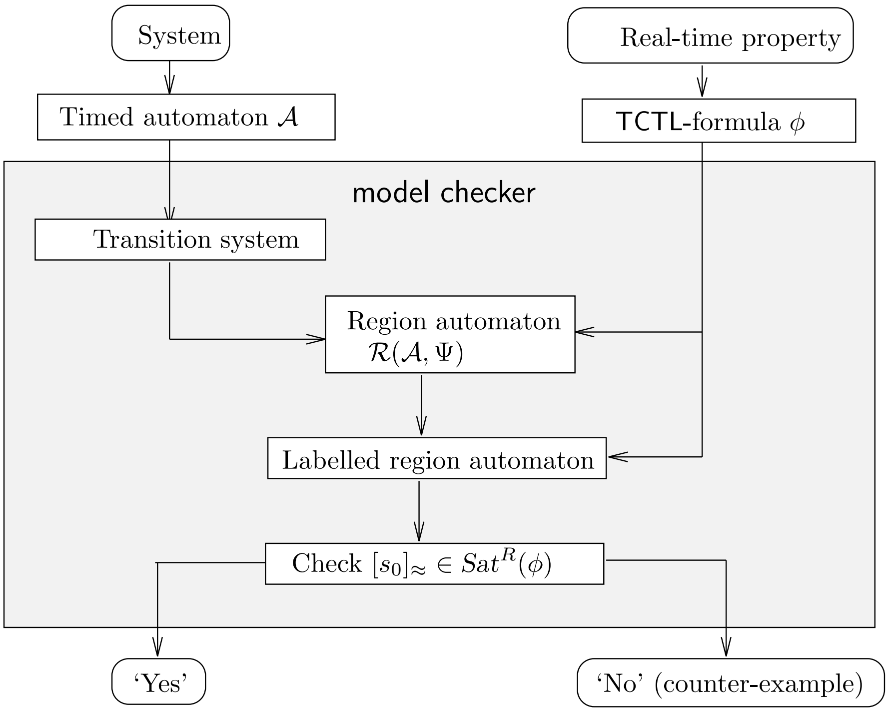


Figure 4.13: Overview of model checking TCTL over a timed automaton

to several industrial case studies such as real-time protocols, multi-media synchronization protocols and proving the correctness of electric power plants. The model checker is available under `http://www.docs.uu.se/docs/rtmv/uppaal/`. An overview of the model checker Uppaal is given in Figure 4.14.

For reasons of efficiency, the model checking algorithms that are implemented in Uppaal are based on (sets of) clock constraints [3], rather than on explicit (sets of) regions. By dealing with (disjoint) sets of clock constraints, a coarser partitioning of the (infinite) state space is obtained. Working with clock constraints allows to characterize $Sat(\phi)$ without explicitly building the region automaton a priori (Yovine, 1998).

Uppaal facilitates the graphical description of timed automata by using the tool Autograph. The output of Autograph is compiled into textual format (using component `atg2ta`), which is checked (by `checkta`) for syntactical correctness. This textual representation is one of the inputs to Uppaal's verifier

[3]Here, the property that each region can be expressed by a clock constraint over the clocks involved, is exploited.

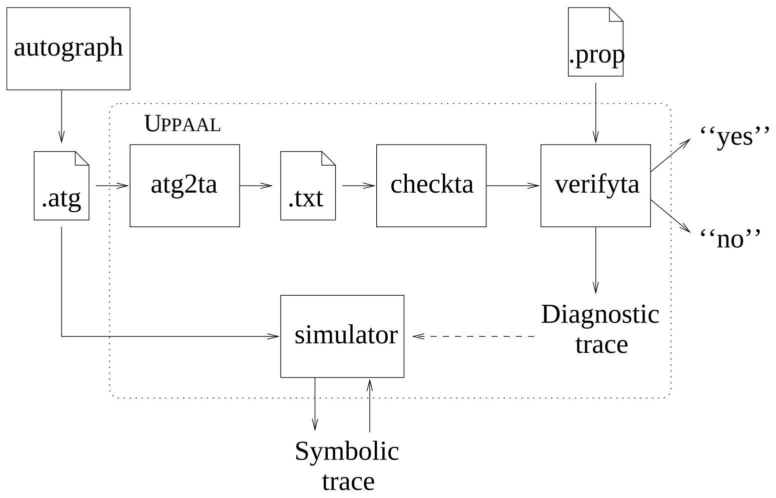


Figure 4.14: The structure of the model checker UPPAAL

`verifyta`. The verifier can be used to determine the satisfaction of a given real-time property with respect to a timed automaton. If a property is not satisfied, a diagnostic trace can be generated that indicates how the property may be violated. UPPAAL also provides a simulator that allows a graphical visualization of possible dynamic behaviors of a system description (i.e., a symbolic trace). The diagnostic trace, generated in case a property is violated, can be fed back to the simulator so that it can be analyzed with the help of the graphical presentation.

To improve the efficiency of the model checking algorithms, UPPAAL does not support the full expressivity of TCTL, but concentrates on a subset of it that is suitable for specifying safety properties. Properties are terms in the language defined by the following syntax:

$$\phi ::= \mathsf{AG}\,\psi \;\Big|\; \mathsf{EF}\,\psi \qquad\qquad \psi ::= a \;\Big|\; \alpha \;\Big|\; \psi \wedge \psi \;\Big|\; \neg\,\psi$$

where a is a location of a timed automaton (like `A.l` for automaton `A` and location `l`), and α a simple linear constraint on clocks or integer variables of the form $\mathtt{x} \sim \mathtt{n}$ for $\sim$ a comparison operator, `x` a clock or integer variable, and `n` an integer. Notice the restricted use of AG and EF as operators: they are only allowed as "top-level" operators, and cannot be nested. This use of path operators limits the expressiveness of the logic, but turns out to pose no significant practical restrictions and makes the model checking algorithm easier. Also notice that formula clocks are not part of the supported logic; only automata clocks can be referred to. Bounded liveness properties can be checked by checking the timed automaton versus a test automaton.

In a nutshell, the most important ingredients for specifying systems in UPPAAL are:

- *Networks of timed automata.* The system specification taken by UPPAAL as input, consists of a *network* of timed automata. Such network is described as the composition of timed automata that can communicate via (equally labelled) channels. If $\mathcal{A}_1$ through $\mathcal{A}_N$ ($N \geqslant 0$) are timed automata then

  $$\mathcal{A} = \mathcal{A}_1 \parallel \mathcal{A}_2 \parallel \ldots \parallel \mathcal{A}_N$$

  specifies the network of timed automata composed of $\mathcal{A}_1$ through $\mathcal{A}_N$. The communication between components is synchronous. Send and receive statements are labels of edges of a timed automaton. A send statement over channel `a`, denoted `a!`, is enabled if the edge is enabled (that is, the associated clock constraint is satisfied in the current clock valuation), and if there is a corresponding transition (in another timed automaton) labelled with the input statement `a?` that is enabled as well. For instance, the two timed automata depicted in Figure 4.15 can communicate along channel `a` if clock `x` reads more than value 2 and clock `y` reads at most 4. If there is no execution of the two timed automata that fulfills this conjunction of constraints, the entire system — including the components that are not involved in the interaction — halts. Since communication is synchronous,

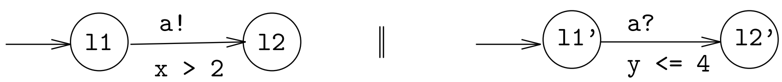
  

  Figure 4.15: Example of communication between timed automata

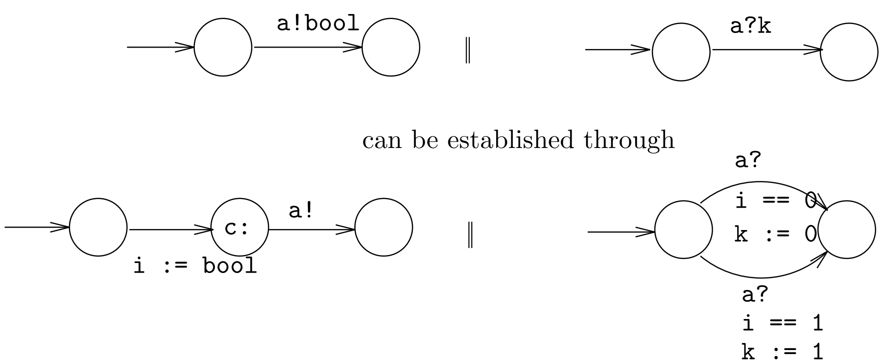
  in case of a synchronization both automata involved take their transition labelled `a!` and `a?` as one atomic transition, and afterwards the system is in locations `l2` and `l2'`, respectively.

- *Clock constraints and invariants* are conjunctions of atomic constraints which have the form `x` $\sim$ `n` where `x` is a variable (a clock or an integer), `n` a non-negative integer, and $\sim \in$ { `<, <=, =, >=, >` } a comparison operator.

- *Data.* The use of data in UPPAAL (version 1.99) is restricted to clocks and integers. Variables can be either local or global, i.e. shared variables are possible. Assignments to integer variable `i` must have the form `i:=n1*i+n2`. Notice that for the latter assignments the variable on the right-hand side of the assignment should be the same as the variable on the left-hand side.

- *Committed locations* are locations that are executed *atomically* and *urgently*. Atomicity of a committed location forbids interference in the activity that is taking place involving the committed location, i.e. the entering of a committed location followed by leaving it constitutes a single atomic event: no interleaving can take place of any other transition (of any other timed automaton). Urgency means that it is not allowed to delay for any positive amount of time in a committed location. Committed locations can thus never be involved in a delay transition. Committed locations are indicated by the prefix `c:`.

- *Value passing.* UPPAAL (version 1.99) does not include mechanisms for value passing at synchronization. Value passing can be effectively modeled by means of assignments to variables. Committed locations are used to ensure that the synchronization between components and the assignment to variables — that establishes the value passing — is carried out atomically. This is exemplified by:



Here, the distributed assignment `k := bool` is established.

Philips' bounded retransmission protocol

To illustrate the capabilities of real-time model checking we treat (part of) the model checking of an industrially relevant protocol, known as the *bounded retransmission protocol* (BRP, for short). This protocol has been developed by Philips Electronics B.V. is currently under standardization, and is used to transfer bulks of data (files) via an infra-red communication medium between audio/video equipment and a remote control unit. Since the communication medium is rather unreliable, the data to be transferred is split into small units, called chunks. These chunks are transmitted separately, and on failure of transmission — as indicated by the absence of an acknowledgement — a retransmission is issued. If, however, the number of retransmissions exceeds a certain threshold, it is assumed that there is a serious problem in the communication medium, and the transmission of the file is aborted. The timing intricacies of the protocol are twofold:

1. firstly, the sender has a timer that is used to initiate a retransmission in case an acknowledgement comes "too late", and
2. secondly, the receiver has a timer to detect the abortion (by the sender) of a transmission in case a chunk arrives "too late".

The correctness of the protocol clearly depends on the precise interpretation of "too late", and in this case study we want to determine clear and tight bounds on these timers in order for the BRP to work correctly. This case study is more extensively described in (D'Argenio, Katoen, Ruys & Tretmans, 1997).

What is the protocol supposed to do?

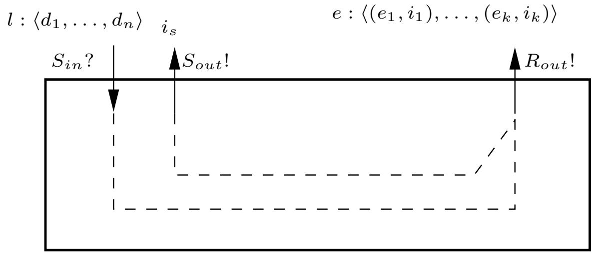
As for many transmission protocols, the service delivered by the BRP behaves like a buffer, i.e., it reads data from one client to be delivered at another one. There are two features that make the behavior much more complicated than a simple buffer. Firstly, the input is a *large file* (that can be modeled as a list), which is delivered in small chunks. Secondly, there is a *limited amount of time* for each chunk to be delivered, so we cannot guarantee an eventually successful delivery within the given time bound. It is assumed that either an initial part of the file or the whole file is delivered, so the chunks will not be garbled and their order will not be changed. Both the sender and the receiver obtain an *indication* whether the whole file has been delivered successfully or not.

The input (the list $l = d_1, \ldots, d_n$) is read on the "input" port. Ideally, each d_i is delivered on the "output" port. Each chunk is accompanied by an indication. This indication can be I_FST, I_INC, I_OK, or I_NOK. I_OK is used if d_i is the last element of the file. I_FST is used if d_i is the first element of the file *and more will follow*. All other chunks are accompanied by I_INC. However, when something goes wrong, a "not OK" indication (I_NOK) is delivered without datum. 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 after transmission of the whole file, or when the protocol gives up. An indication is sent out on the "input" port. This indication can be 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 may occur after delivery of the last-but-one chunk d_{n-1}. This situation arises, because no realistic implementation can ensure whether the last chunk got lost. The reason is that information about a successful delivery has to be transported back somehow over the same unreliable medium. In case the last acknowledgement fails to come, there is no way to know whether the last chunk d_n has been delivered or not. After this indication, the protocol is ready to transmit a subsequent file.

This completes the informal FTS (*File Transfer Service*) description. Remark that it is unclear from this description which indication the sending client receives in case the receiving client does not receive any chunk. Since something went wrong an I_NOK indication is required, but from this indication the sending client may not deduce that the receiving client has the corresponding indication. This is because the receiving client does not receive an I_NOK indication before delivery of the first chunk. So, if the sending client receives an I_NOK either the receiving client received the same or did not receive anything at all.

Formal specification of the FTS (optional)



Schematic view of the FTS

Signatures of the input and output:

$S_{in} : l = \langle d_1, \ldots, d_n \rangle$ for $n > 0$

$S_{out} : i_s \in \{ \text{I\_OK}, \text{I\_NOK}, \text{I\_DK} \}$

$R_{out} : \langle (e_1, i_1), \ldots, (e_k, i_k) \rangle$ for $0 \leqslant k \leqslant n, i_j \in \{ \text{I\_FST}, \text{I\_INC}, \text{I\_OK}, \text{I\_NOK} \}$ for $0 < j \leqslant k$.

The FTS is considered to have two "service access points": one at the sender side and the other at the receiver side. The sending client inputs its file via S_{in} as a list of chunks $\langle d_1, \ldots, d_n \rangle$. We assume that $n > 0$, i.e., the transmission of empty files is not considered. The sending client receives indications i_s via S_{out}, while the receiving client receives pairs (e_j, i_j) of chunks and indications via R_{out}. We assume that all outputs with respect to previous files have been completed when a next file is input via S_{in}.

In Table 4.5 we specify the FTS in a logical way, i.e., by stating properties that should be satisfied by the service. These properties define relations between input and output. Note that a distinction is made between the case in which the receiving client receives at least one chunk ($k > 0$) and the case that it receives none ($k = 0$). A protocol conforms to the FTS if it satisfies all listed properties.

For $k > 0$ we have the following requirements. **(1.1)** states that each correctly received chunk e_j equals d_j, the chunk sent via S_{in}. In case the notification i_j indicates that an error occurred, no restriction is imposed on the accompanying

Table 4.5: Formal specification of the FTS

$k > 0$	
(1.1)	$\forall 0 < j \leqslant k : i_j \neq \text{I\_NOK} \;\Rightarrow\; e_j = d_j$
(1.2)	$n > 1 \;\Rightarrow\; i_1 = \text{I\_FST}$
(1.3)	$\forall 1 < j < k : i_j = \text{I\_INC}$
(1.4.1)	$i_k = \text{I\_OK} \vee i_k = \text{I\_NOK}$
(1.4.2)	$i_k = \text{I\_OK} \;\Rightarrow\; k = n$
(1.4.3)	$i_k = \text{I\_NOK} \;\Rightarrow\; k > 1$
(1.5)	$i_s = \text{I\_OK} \;\Rightarrow\; i_k = \text{I\_OK}$
(1.6)	$i_s = \text{I\_NOK} \;\Rightarrow\; i_k = \text{I\_NOK}$
(1.7)	$i_s = \text{I\_DK} \;\Rightarrow\; k = n$
$k = 0$	
(2.1)	$i_s = \text{I\_DK} \;\Leftrightarrow\; n = 1$
(2.2)	$i_s = \text{I\_NOK} \;\Leftrightarrow\; n > 1$

chunk e_j. **(1.2)** through **(1.4)** address the constraints concerning the received indications via R_{out}, i.e., i_j. If the number n of chunks in the file exceeds one then **(1.2)** requires i_1 to be I_FST, indicating that e_1 is the first chunk of the file and more will follow. **(1.3)** requires that the indications of all chunks, apart from the first and last chunk, equal I_INC. The requirement concerning the last chunk (e_k, i_k) consists of three parts. **(1.4.1)** requires e_k to be accompanied with either I_OK or I_NOK. **(1.4.2)** states that if i_k = I_OK then k should equal n, indicating that all chunks of the file have been received correctly. **(1.4.3)** requires that the receiving client is not notified in case an error occurs before delivery of the first chunk. **(1.5)** through **(1.7)** specify the relationship between indications given to the sending and receiving client. **(1.5)** and **(1.6)** state when the sender and receiver have corresponding indications. **(1.7)** requires a "don't know" indication to only appear after delivery of the last-but-one chunk d_{n-1}. This means that the number of indications received by the receiving client must equal n. (Either this last chunk is received correctly or not, and in both cases an indication (+ chunk) is present at R_{out}.)

For $k = 0$ the sender should receive an indication I_DK if and only if the file to be sent consists of a single chunk. This corresponds to the fact that a "don't know" indication may occur after the delivery of the last-but-one chunk only. For $k = 0$ the sender is given an indication I_NOK if and only if n exceeds one. This gives rise to **(2.1)** and **(2.2)**.

Remark that there is no requirement concerning the limited amount of time available to deliver a chunk to the receiving client as mentioned in the informal service description. The reason for this is that this is considered as a protocol requirement rather than a service requirement.

How is the protocol supposed to work?

The protocol consists of a sender S equipped with a timer T_1, and a receiver R equipped with a timer T_2 which exchange data via two unreliable (lossy) channels, K and L.

Sender S reads a file to be transmitted and sets the retry counter to 0. Then it starts sending the elements of the file one by one over K to R. Timer T_1 is set and a frame is sent into channel K. This frame consists of three bits and a datum (= chunk). The first bit indicates whether the datum is the first element of the file. The second bit indicates whether the datum is the last item of the file. The third bit is the so-called alternating bit, that is used to guarantee that data is not duplicated. After having sent the frame, the sender waits for an acknowledgement from the receiver, or for a timeout. In case an acknowledgement arrives, the timer T_1 is reset and (depending on whether this was the last element of the file) the sending client is informed of correct transmission, or the next element of the file is sent. If timer T_1 times out, the frame is resent (after the counter for the number of retries is incremented and the timer is set again), or the transmission of the file is broken off. The latter occurs if the retry counter exceeds its maximum value MAX.

Receiver R waits for a first frame to arrive. This frame is delivered at the receiving client, timer T_2 is started and an acknowledgement is sent over L to S. Then the receiver simply waits for more frames to arrive. The receiver remembers whether the previous frame was the last element of the file and the expected value of the alternating bit. Each frame is acknowledged, but it is handed over to the receiving client only if the alternating bit indicates that it is new. In this case timer T_2 is reset. Note that (only) if the previous frame was last of the file, then a fresh frame will be the first of the subsequent file and a repeated frame will still be the last of the old file. This goes on until T_2 times out. This happens if for a long time no new frame is received, indicating that transmission of the file has been given up. The receiving client is informed, provided the last element of the file has not just been delivered. Note that if transmission of the next file starts before timer T_2 expires, the alternating bit scheme is simply continued. This scheme is only interrupted after a failure.

Timer T_1 times out if an acknowledgement does not arrive "in time" at the sender. It is set when a frame is sent and reset after this frame has been acknowledged. (Assume that premature timeouts are not possible, i.e., a message must not come *after* the timer expires.)

Timer T_2 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. So its delay must exceed MAX times the delay of T_1.[4] Assume that the sender does not start reading and transmitting the next file before the receiver has properly reacted to the failure. This is necessary, because the receiver has not yet switched its alternating bit, so a new frame would be interpreted as a repetition.

This completes the informal description of the BRP. Two significant assumptions are made in the above description, referred to as **(A1)** and **(A2)** below.

(A1) Premature timeouts are not possible

Let us suppose that the maximum delay in the channel K (and L) is TD and that timer T_1 expires if an acknowledgement has not been received within $\mathsf{T1}$ time units since the first transmission of a chunk. Then this assumption requires that $\mathsf{T1} > 2 \times \mathsf{TD} + \delta$ where δ denotes the processing time in the receiver R. **(A1)** thus requires knowledge about the processing speed of the receiver and the

[4]Later on we will show that this lower bound is not sufficient.

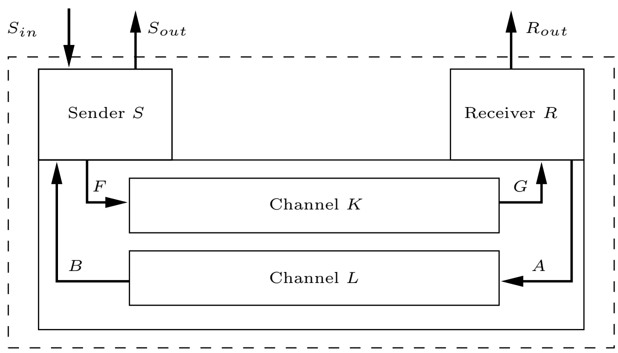
delay in the line.

> **(A2)** In case of abortion, S waits before starting a new file until R reacted properly to abort

Since there is no mechanism in the BRP that notifies the expiration of timer T_2 (in R) to the sender S this is a rather strong and unnatural assumption. It is unclear how S "knows" that R has properly reacted to the failure, especially in case S and R are geographically distributed processes — which apparently is the case in the protocol at hand. We, therefore, consider **(A2)** as an unrealistic assumption. In the sequel we ignore this assumption and adapt the protocol slightly such that this assumption appears as a property of the protocol (rather than as an assumption!).

Modeling the protocol as a network of timed automata

The BRP consists of a sender S and a receiver R communicating through channels K and L, see the figure below. S sends chunk d_i via F to channel K accompanied with an alternating bit ab, an indication b whether d_i is the first chunk of a file (i.e., $i = 1$), and an indication b' whether d_i is the last chunk of a file (i.e., $i = n$). K transfers this information to R via G. Acknowledgements ack are sent via A and B using L.



Schematic view of the BRP.

The signatures of A, B, F, and G are:

$$F, G : (b, b', ab, d_i) \text{ with } ab \in \{\,0, 1\,\}, b, b' \in \{\,\text{true}, \text{false}\,\} \text{ and } 0 < i \leqslant n$$

In addition, $A, B : ack$.

We adopt the following notational conventions. States are represented by labelled circles and the initial state as double-lined labelled circle. State invariants are denoted in brackets. Transitions are denoted by directed, labelled arrows. A list of guards denotes the conjunction of its elements. For the sake of simplicity, in this example we use value passing and variable assignments in an unrestricted way.

Channels K and L are simply modeled as first-in first-out queues of unbounded capacity with possible loss of messages. We assume that the maximum latency of both channels is TD time units.

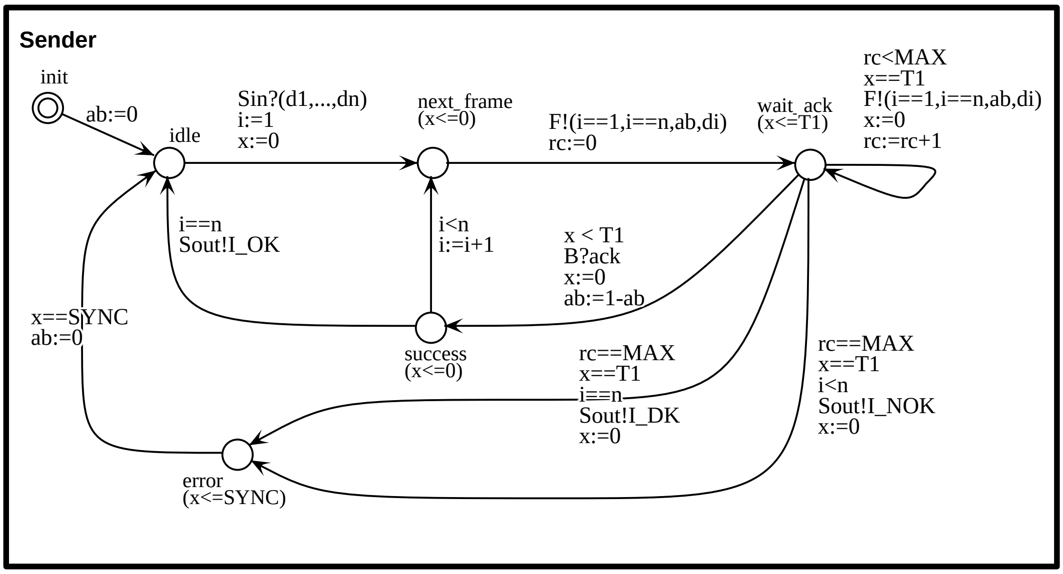


Figure 4.16: Timed automaton for sender S.

Modeling the sender

The sender S (see Figure 4.16) has three system variables: $ab \in \{0, 1\}$ indicating the alternating bit that accompanies the next chunk to be sent, i, $0 \leqslant i \leqslant n$, indicating the subscript of the chunk currently being processed by S, and rc, $0 \leqslant rc \leqslant \mathsf{MAX}$, indicating the number of attempts undertaken by S to retransmit d_i. Clock variable x is used to model timer T_1 and to make certain transitions urgent (see below). In the `idle` location S waits for a new file to be received via S_{in}. On receipt of a new file it sets i to one, and resets clock x. Going from location `next_frame` to `wait_ack`, chunk d_i is transmitted with the corresponding information and rc is reset. In location `wait_ack` there are several possibilities: in case the maximum number of retransmissions has been reached (i.e., $rc = \mathsf{MAX}$), S moves to an `error` location while resetting x and emitting an I_DK or I_NOK indication to the sending client (via S_{out}) depending on whether d_i is the last chunk or not; if $rc < \mathsf{MAX}$, either an `ack` is received (via B) within time (i.e., $x < \mathsf{T1}$) and S moves to the `success` location while alternating ab, or timer x expires (i.e., $x = \mathsf{T1}$) and a retransmission is initiated (while incrementing `rc`, but keeping the same alternating bit). If the last chunk has been acknowledged, S moves from location `success` to location `idle` indicating the successful transmission of the file to the sending client by I_OK. If another chunk has been acknowledged, i is incremented and x reset while moving from location `success` to location `next_frame` where the process of transmitting the next chunk is initiated.

Two remarks are in order. First, notice that transitions leaving location s, say, with location invariant $x \leqslant 0$ are executed without any delay with respect to the previous performed action, since clock x equals 0 if s is entered. Such transitions are called *urgent*. Urgent transitions forbid S to stay in location s arbitrarily long and avoid that receiver R times out without abortion of the transmission by sender S. Urgent transitions will turn out to be necessary to achieve the correctness of the protocol. They model a maximum delay on processing speed, cf. assumption **(A1)**. Secondly, we remark that after a failure (i.e., S is in 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 failure. This timer will make it possible to satisfy assumption **(A2)**. In case of failure the alternating bit scheme is restarted.

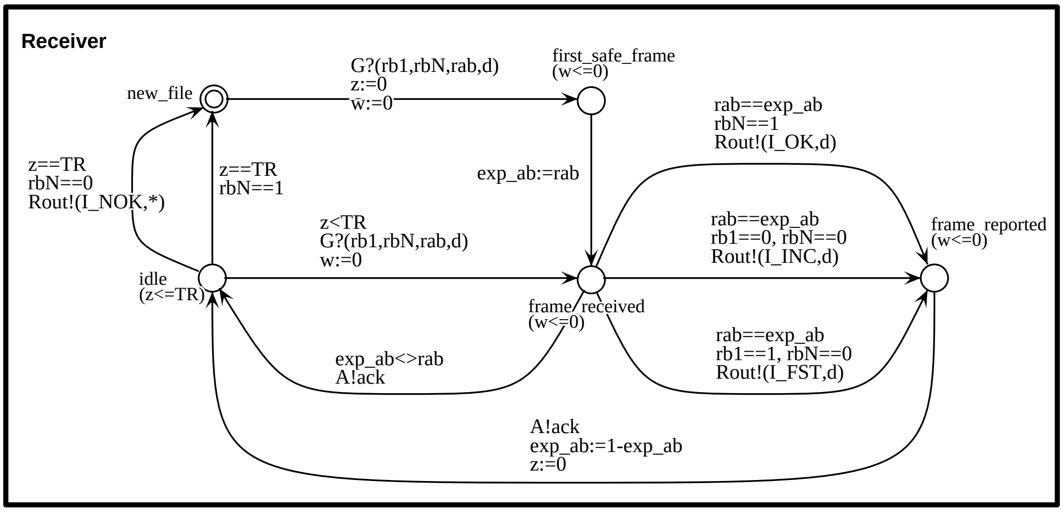


Figure 4.17: Timed automaton for receiver R.

Modeling the receiver

The receiver is depicted in Figure 4.17. System variable `exp_ab` $\in \{0, 1\}$ in receiver R models the expected alternating bit. Clock z is used to model timer T_2 that determines transmission abortions of sender S, while clock w is used to make some transitions urgent. 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.

Most of the transitions in R are made urgent in order to be able to fulfill assumption **(A1)**. For example, if we allowed an arbitrary delay in location `frame_received` then the sender S could generate a timeout (since it takes too long for an acknowledgement to arrive at S) while an acknowledgement generated by R is possibly still to come.

Model checking the specification

An important aspect of the BRP is the question what the relationships between the timeout values, transmission delays and synchronization delays (like SYNC) are in order for the protocol to work correctly.

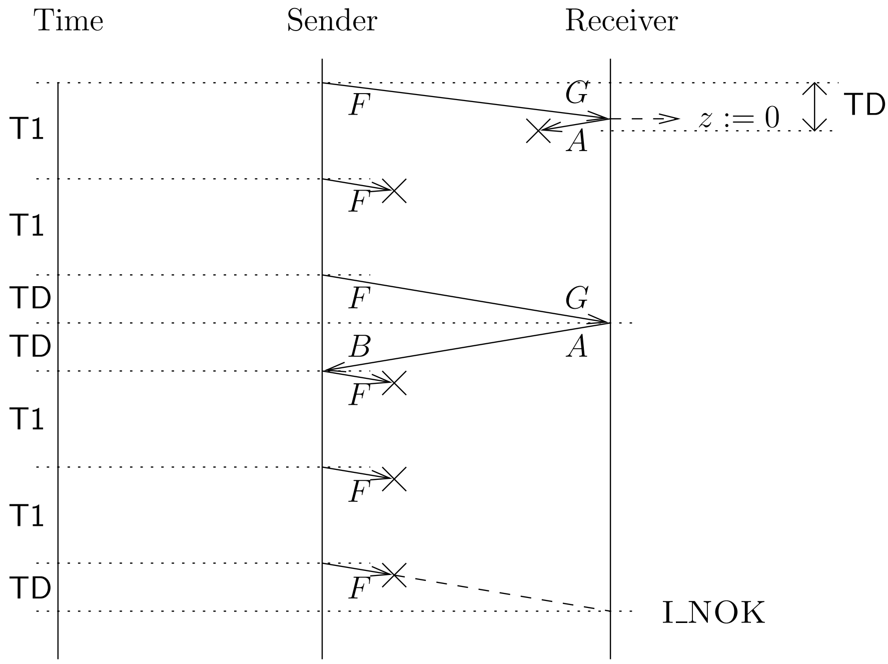
Premature timeouts

Assumption **(A1)** states that no premature timeouts should occur. The BRP contains two types of timeouts: one for detecting the absence of a promptly acknowledgement (by the sender), and one for detecting the abortion of the transmission (by the receiver). It can easily be seen that timer T_1 (i.e., clock x) of sender S does not violate assumption **(A1)** if it respects the two-way transmission delay (i.e., $\mathsf{T1} > 2 \times \mathsf{TD}$) plus the processing delay of the receiver R (which due to the presence of location invariants equals 0). It remains to be checked under which conditions timer T_2 of receiver R does not generate premature timeouts. This amounts to checking that R times out whenever the sender has indeed aborted the transmission of the file. Observe that a premature timeout appears in R if it moves from location `idle` to state `new_file` although there is still some frame of the previous file to come. We therefore check that in location `first_safe_frame` receiver R can only receive first chunks of a file (i.e., $\texttt{rb1} = 1$) and not remaining ones of previous files. This is formulated in UPPAAL notation as:

$$\mathsf{AG}\,(\texttt{R.first\_safe\_frame} \;\Rightarrow\; \texttt{rb1} = 1) \tag{4.4}$$

Using several verifications with the verification engine `verifyta` of UPPAAL we can establish that this property holds whenever $\mathsf{TR} \geqslant 2 \times \mathsf{MAX} \times \mathsf{T1} + 3 \times \mathsf{TD}$.

In order to conclude this, the simulator of UPPAAL together with the diagnostic traces have been of big help. We were able to try different values for TD, $\mathsf{T1}$, and MAX, and thus, to study the behavior of the protocol. Figure 4.18 depicts the longest trace that makes the protocol loose the connection when $\mathsf{MAX} = 2$ and $n \geqslant 2$. The $\times$ symbol indicated after sending a frame represents that it is lost in some of the channels K or L. Notice that frames are received within TD time units but they always take some time to travel through the channel. In particular, in the transmission of the first frame, the difference in time between synchronization on F and synchronization on G cannot be 0.



$= 2 \cdot \mathsf{MAX} \cdot \mathsf{T1} + 3 \cdot \mathsf{TD}$

Figure 4.18: Loosing the connection

From the figure, it is clear that the receiver should not timeout (strictly) before $2 \times \mathsf{MAX} \times \mathsf{T1} + 3 \times \mathsf{TD}$ units of time, since this is the last time a frame can arrive. Premature timeouts would induce the receiver to abort the connection when there is still the possibility of some frame to arrive. As a result, property (4.4) would be violated.

Premature abortions

Assumption **(A2)** states that sender S starts the transmission of a new file only after R has properly reacted to the failure. For our model this means that if S is in location `error`, eventually, within SYNC time units, R resets and is able to receive a `new_file`. This property is expressed in TCTL as

$$\mathsf{A}\,[\mathtt{S.error}\,\mathsf{U}_{\leqslant \mathsf{SYNC}}\,\mathtt{R.new\_file}] \tag{4.5}$$

Unfortunately, the property language of UPPAAL does not support this type of formula. Therefore, we check the following property:

$$\mathsf{AG}\,[(\mathtt{S.error}\ \wedge\ x = \mathsf{SYNC})\ \Rightarrow\ \mathtt{R.new\_file}] \tag{4.6}$$

The difference between properties (4.5) and (4.6) is that (4.5) requires that `S.error` is true until `R.new_file` becomes true, while (4.6) does not take into account what happens when time passes, but considers only the instant for which $x = \mathsf{SYNC}$. Provided that S is in location `error` while clock x evolves from 0 to SYNC — which is obviously the case — (4.6) implies (4.5). Using several verifications with the verification engine `verifyta` of UPPAAL we can establish that property (4.6) is satisfied under the condition that $\mathsf{SYNC} \geqslant \mathsf{TR}$. This means that **(A2)** is fulfilled if this condition on the values SYNC and TR is respected.

Verification results

Summarizing, we were able to check with UPPAAL that assumptions **(A1)** and **(A2)** are fulfilled by the BRP if the following additional constraints hold

$$\mathsf{TD} > 2 \times \mathsf{TD}\ \wedge\ \mathsf{SYNC} \geqslant \mathsf{TR} \geqslant 2 \times \mathsf{MAX} \times \mathsf{T1} + 3 \times \mathsf{TD}$$

Remark that SYNC and $\mathsf{T1}$ are constants in the sender S, while TR is a constant used in receiver R. These results show the importance of real-time aspects for the correctness of the BRP.

Exercises

EXERCISE 26. The intuitive meaning of the formula $\mathsf{E}\{\, I \,\}\phi$ is that there exists a path starting from the current state such that the *earliest* possible time at which ϕ is true lies in the interval I, for I an interval with integer boundaries. Define this operator either semantically (using $\models$), or try to formulate it in terms of existing temporal operators.

EXERCISE 27. Consider the region $r = [(2 < x < 3), (1 < y < 2), (x{-}y < 1)]$ and let clock valuation v such that $r = [v]$.

1. Characterize the region to which $[x]v$ belongs.
2. Characterize the direct time successor of r.
3. Suppose assignments of the form $x := y$ are allowed. Characterize the region r' that results from r under $x := y$.

EXERCISE 28. Let v, v' be two clock valuations over the set of clocks C such that $v \approx v'$. Prove that for any time constraint $\alpha \in \Psi(C)$ we have: $v \models \alpha$ if and only if $v' \models \alpha$.

EXERCISE 29. Consider the timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$ depicted in Figure 4.19. Construct the region automata $\mathcal{R}(\mathcal{A}_1)$ and $\mathcal{R}(\mathcal{A}_2)$ for $c_z = 1$. Explain the differences and indicate for both region automata the unbounded regions.

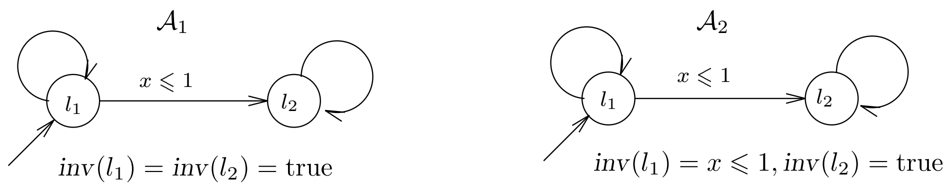
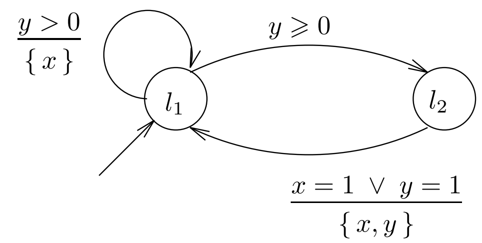


Figure 4.19: Two similar timed automata

EXERCISE 30. Consider the modified light switch of Figure 4.10 and suppose *on* is labelled with atomic proposition p and *off* is labelled with q.

1. Give a TCTL-formula that expresses the property "once the light is off, it will be switched off eventually after it has been switched on".
2. Check the property with the help of the region automaton of Figure 4.11 by indicating a path that invalidates this property or by arguing that all paths do satisfy it.
3. Give a TCTL-formula that expresses the property "there is a possibility that when the light is off, it is switched on, and switched off within 5 time-units?"
4. Check the property with the help of the region automaton of Figure 4.11 by indicating a path that satisfies this property or by arguing that there is no path that satisfies it.

EXERCISE 31. Consider the following timed automaton:



$inv(l_1) = x \leqslant 1 \;\wedge\; y \leqslant 1$

$inv(l_2) = x \leqslant 1 \;\wedge\; y \leqslant 1$

1. Construct the region automaton for this timed automaton for a formula with maximal constant 1; indicate explicitly the delay transitions and the unbounded regions.
2. Check the validity of the following formula

   $$\mathsf{E}\,[(y = 0)\,\mathsf{U}\,\mathsf{E}\,[(x > 0)\,\mathsf{U}\,(y = 0)]]$$

   and justify your answer.
3. Extend the timed automaton with a counter n that is initialized as 0 and is increased by 1 on every visit to location l_1. Check the validity of the formula

   $$z \text{ in } \mathsf{EF}\,(z \leqslant 10 \;\wedge\; n > 10)$$

   and justify your answer.

4. Consider the formula $\mathsf{AF}\, p$ where p is an atomic proposition that is true if and only if the timed automaton is in location l_1.

   (a) Check the validity of this formula.

   (b) What is the validity of this formula if a notion of weak fairness (see Chapter 3) is adopted?

EXERCISE 32. Model and analyses the following soldier problem in UPPAAL. Four soldiers are heavily injured, try to flee to their home land. The enemy is chasing them and in the middle of the night they arrive at a bridge that has been damaged and can only carry two soldiers at a time. Furthermore, several land mines have been placed on the bridge and a torch is needed to move the mines out of the way. The enemy is approaching, so the soldiers know that they only have 60 minutes to cross the bridge. The soldiers only have a single torch and they are not equally injured; thus the crossing time (one-way) for the soldiers is 5 min, 10 min, 20 min and 25 minutes.

(a) Model the soldier problem in UPPAAL as a network of interacting timed automata

(b) Verify whether there exists a strategy that results in saving all soldiers.

(c) Is there such strategy if the enemy is arriving a bit earlier, say, within 55 minutes?

(This exercise is due to Ruys and Brinksma.)

4.10 Selected references

Introduction and overview of real-time temporal logics:

- R. KOYMANS. *Specifying Message Passing and Time-Critical Systems with Temporal Logic.* LNCS 651. 1992.
- J.S. OSTROFF. Formal methods for the specification and design of real-time safety critical systems. *Journal of Systems and Software*, **18**, 1992.

- T.A. Henzinger. It's about time: real-time logics reviewed. In *Concurrency Theory*, LNCS 1466, pages 439–454, 1998.

Past operators in real-time temporal logics:

- R. Alur and T.A. Henzinger. Back to the future: towards a theory of timed regular languages. In *IEEE Symp. on Foundations of Computer Science*, pages 177–186, 1992.

Overview of issues related to the incorporation of real-time in formal methods:

- R. Koymans. (Real) time: a philosophical perspective. In *Real-Time: Theory in Practice*, LNCS 600, pages 353–370, 1992.

Model checking of real-time systems:

- R. Alur, C. Courcoubetis and D. Dill. Model-checking in dense real-time. *Information and Computation*, **104**: 2–34, 1993.
- R. Alur and D. Dill. Automata-theoretic verification of real-time systems. In *Formal Methods for Real-Time Computing*, pages 55–82, 1996.
- K.G. Larsen, B. Steffen and C. Weise. Continuous modelling of real-time and hybrid systems: from concepts to tools. *Software Tools for Technology Transfer*, **1**: 64–86, 1997.
- S. Yovine. Model checking timed automata. In *Embedded Systems*, LNCS 1494, 1998.

Expressiveness and decidability of real-time logics:

- R. Alur and T. Henzinger. Real-time logics: Complexity and expressiveness. *Information and Computation*, **104**: 35–77, 1993.

Tools for real-time model checking:

- K.G. Larsen, P. Pettersson and W. Yi. Uppaal in a nutshell. *Software Tools for Technology Transfer*, **1**: 134–153, 1997.
- S. Yovine. Kronos: a verification tool for real-time systems. *Software Tools for Technology Transfer*, **1**: 123–134, 1997.
- T.A. Henzinger, P.-H. Ho and H. Wong-Toi. HyTech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, **1**: 110–123, 1997.
- S. Tripakis and C. Courcoubetis. Extending Promela and Spin for real time. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1055, pages 329–348, 1996.

Bounded retransmission protocol case study:

- P.R. D'Argenio, J.-P. Katoen, T. Ruys, and G.J. Tretmans. The bounded retransmission protocol must be on time! In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1217, pages 416–432, 1997.

Chapter 5

A Survey of State-Space Reduction Techniques

As we have seen in the previous chapters, the worst case time complexity of model checking is polynomial in the size of the state space of the system model. Experiments have demonstrated that time complexity is *not* the limiting factor of model checking, but the usage of memory is:

The real practical limiting factor of model checking is the often excessive size of the state space.

More precisely, the amount of main memory is the major limiting factor of most conventional model checking algorithms. For model checking, being it PLTL, CTL or TCTL, the basic ingredient of checking the validity of a given property is a systematic traversal of the state space. For linear-time temporal logic, the emptiness of a Büchi automaton needs to be checked which boils down to a depth-first search of the reachable state space, for CTL (and its real-time variant) model checking boils down to labelling the reachable state space. This exploration of the state space is efficient only, when the reachable states are stored in main memory, since usage of secondary memory would introduce a major bottleneck. It is, however, typical for many applications that with traditional state space exploration techniques — based on explicit state enumeration — this is a major obstacle: the available amount of main memory is highly insufficient.

A prominent example of systems where the state space is typically too large are systems that consist of various components that cooperatively carry out a certain task. For such applications, like descriptions of asynchronous hardware systems, communication protocols or parallel (distributed) systems, the system description usually consists of a number of concurrent processes. Typically the state space of the parallel composition of a process with n states and a process with k states results in a state space of $n \times k$ states. Accordingly, the parallel composition of N processes with each a state space of k states leads to a state space with k^N states. Even for small systems this may easily run out of control, as illustrated in the next example.

Example 43. Let us consider Dijkstra's solution to the mutual exclusion problem. In the original mutual exclusion algorithm by Dijkstra in 1965, a 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
```

It is not difficult to convert this algorithm into SMV*-code, the model checker for* CTL *that we have encountered in Chapter 3. The precise code is not of importance here, and is omitted. By a small experiment we can show the impact of the number of processes in Dijkstra's mutual exclusion program to the size of the state space. To that purpose we let* $2 \leqslant n \leqslant 8$ *and show the size of the state space in Figure 5.1. (Note that the scale of the vertical axis is logarithmic.) (End*

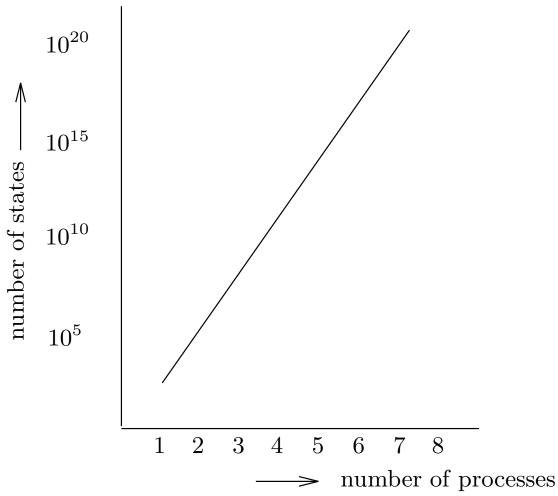


Figure 5.1: Number of states versus the number of processes

of example.)

This exponential growth of the state space is referred to as the *state-space explosion* problem. The state-space explosion problem is a problem that not only appears for model checking, but also for various other problems like performance analysis that are strongly based on state space exploration. In order to tackle this problem in the setting of model checking various approaches have been considered. In this chapter we give a (brief) overview of some of the major techniques to overcome state-space explosion. In particular we consider the following techniques:

- *Symbolic representation of state spaces* using binary decision diagrams: a technique to encode transition systems in terms of a compact model that is suited to represent boolean functions efficiently.
- *Efficient memory management strategies*: techniques that try to optimize

the use of (main) memory for model checking by using hashing, caching techniques, and so forth.

- *Partial-order reduction*: a technique that exploits the fact that for checking many properties it is not necessary to consider all possible ways in which a state space can be traversed.

- *Equivalence and pre-order relations*: a technique to transform models (like CTL-models and PLTL-models) into equivalent, but smaller models that satisfy the same properties.

- *Compositional verification*: a technique to decompose properties in sub-properties such that each sub-property can be checked for a part of the state space, and the combination of the sub-properties implies the required global property.

- *Abstract interpretation* is a technique to reduce the increase of the state space due to date by providing a mapping between the concrete data values in the system and a small set of abstract data values that suffices to check the properties of interest.

(In the current version of the lecture notes, the later two techniques are not treated.)

5.1 Symbolic state-space representation

The different methods for model checking, like model checking linear and branching temporal logic, are all based on a system description in terms of (a kind of) finite state-transition diagram. The most obvious way to store this automaton in memory is to use a representation that *explicitly* stores the states and the transitions between them. Typically data structures such as linked lists and adjacency matrices are used.

> *The idea of the BDD (Binary Decision Diagram)-technique is to replace the explicit state-space representation by a compact* symbolic *representation.*

The state space, being it a Büchi automaton, CTL-model, or a region automaton, is represented using binary decision diagrams, models that are used for the compact representation of boolean functions. BDDs can not guarantee the avoidance of the state space explosion in all cases, but provide a compact representation of several systems, allowing these systems to be verified — systems that would be impossible to handle using explicit state enumeration methods.

A BDD (due to Akers, 1959) is similar to a binary decision tree, except that its structure is a *directed acyclic graph*, and there is a total order on the occurrences of variables that determines the order of vertices when traversing the graph from root to one of its leaves (i.e. end-point). Once the state space is encoded using BDDs, the model checking algorithm at hand is carried out using these symbolic representations. This means that all manipulations performed by these algorithms must be carried out, as efficiently as possible, on BDDs. It turns out that in particular the operations needed for CTL model checking can be carried out efficiently on BDDs. In SMV, the model checker we have discussed in Chapter 3, these techniques are used.

Encoding relations as boolean functions

To introduce the use of BDDs, we describe in this section how CTL-models can be symbolically represented using BDDs. Recall that such models consist of a finite set of states S, a successor relation $R \subseteq S \times S$, and a labelling of states with atomic propositions. This labelling function is not of great importance here, so we omit this component in order not to blur the presentation. We start by discussing how R can be considered as a boolean function after which we describe how boolean functions can be represented as BDDs. Without loss of generality, assume that S has a cardinality 2^n for $n \geqslant 1$. (If S does not satisfy this constraint, then dummy states can be added.)

Each state in S is represented by a bit-vector of length n using a bijective encoding $[\![\ ]\!] : S \longrightarrow \{0,1\}^n$. For instance, a possible encoding for $S = \{s, s'\}$ is $[\![s]\!] = 0$ and $[\![s']\!] = 1$. The successor relation R is a binary relation over $\{0,1\}^n$ such that

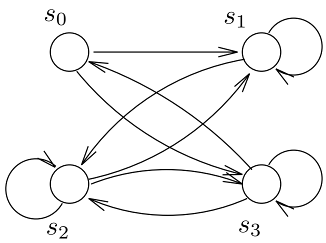
$$([\![s]\!], [\![s']\!]) \in \mathsf{R} \text{ if and only if } (s, s') \in R$$

For instance, the pair $(0,1)$ is in R if and only if state s' is a successor of s. Relation R is defined by its characteristic function:

$$f_{\mathsf{R}}([\![s]\!], [\![s]\!]') = 1 \text{ if and only if } ([\![s]\!], [\![s]\!]') \in \mathsf{R}$$

where $f_{\mathsf{R}} : \{0,1\}^{2n} \longrightarrow \{0,1\}$. In this way the relation R is represented as a boolean function.

Example 44. *Consider the* CTL*-model depicted below. Let the encoding function be defined as* $[\![s_0]\!] = 00$, $[\![s_1]\!] = 01$, $[\![s_2]\!] = 10$ *and* $[\![s_3]\!] = 11$. *The characteristic function of the successor relation* R *is listed by its truth table next to the* CTL*-model. For example, the fact that state* s_1 *is a successor of* s_2 *follows from* $f_{\mathsf{R}}([\![s_2]\!], [\![s_1]\!]) = 1$, *that is,* $f_{\mathsf{R}}(1,0,0,1) = 1$.



An example CTL*-model*

f_{R}	*00*	*01*	*10*	*11*
00	*0*	*1*	*0*	*1*
01	*0*	*1*	*1*	*0*
10	*0*	*1*	*1*	*1*
11	*1*	*0*	*1*	*1*

(End of example.)

Binary decision trees

Boolean function $f : \{0,1\}^k \longrightarrow \{0,1\}$ (like the transition relation before) can be represented by a truth table, listing all possible inputs and corresponding output, or alternatively, as a rooted *binary decision tree* (BDT). The idea of this representation is that each leaf n is labelled with a boolean *value*(n) which is either 0 or 1. Other vertices are labelled with a boolean variable $var(n)$ and have two children: $z(n)$ and $o(n)$. The first child corresponds to the case that the value of $var(n)$ equals zero; the latter to the case that $var(n)$ equals one. Leafs are often called terminals and other vertices are referred to as non-terminals. In the representation of a BDT edges to zero-children are drawn as dashed lines, while edges to one-children are represented by solid lines. For example, the binary decision tree of the characteristic function f_{R} of the previous example is given in Figure 5.2.

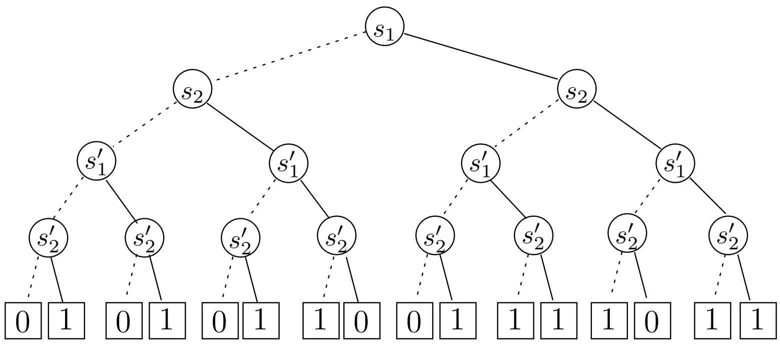


Figure 5.2: A binary decision tree representing f_{R}

The function value for a given assignment to the arguments can be determined by traversing the tree starting from the root, branching at each vertex based on the assigned value of the variable that labels the vertex, until a leaf is reached. The value of the leaf is now the function value. For instance, to determine $f_{\mathsf{R}}(1, 0, 0, 1)$, we instantiate the variables $s_1 := 1$, $s_2 := 0$, $s'_1 := 0$ and $s'_2 := 1$ and traverse the tree accordingly. BDTs are not very compact; in fact the size of such tree is identical to the size of the truth table of f. In addition, 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, we could change the current variable ordering s_1, s_2, s'_1, s'_2 into s_1, s'_1, s_2, s'_2 without affecting the size of the BDT depicted in Figure 5.2.

The boolean function represented by means of a BDT is obtained as follows:

Definition 57. (**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 a vertex v is defined as follows: for v a terminal vertex $f_{\mathsf{B}}(v) = \mathit{value}(v)$ and for v a non-terminal vertex:

$$f_{\mathsf{B}}(v) = (\mathit{var}(v) \ \wedge \ f_{\mathsf{B}}(o(v))) \ \vee \ (\neg \, \mathit{var}(v) \ \wedge \ f_{\mathsf{B}}(z(v))).$$

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. The value of f_{B} for assignment $x_1 := b_1, \ldots, x_n := b_n$, for $b_i \in \{ 0, 1 \}$, equals $f_{\mathsf{B}}[x_1 := b_1, \ldots, x_n := b_n]$. The above definition is inspired by the so-called *Shannon expansion* for boolean functions. Function $f : \{ 0, 1 \}^n \longrightarrow \{ 0, 1 \}$ can be rewritten using this expansion into:

$$f(x_1, \ldots, x_n) = (x_1 \ \wedge \ f(1, x_2, \ldots, x_n)) \ \vee \ (\neg \, x_1 \ \wedge \ f(0, x_2, \ldots, x_n)).$$

Reduced ordered binary decision diagrams

In a binary decision *diagram* the redundancy that is present in a decision tree is reduced. A binary decision diagram is a binary decision tree in which isomorphic subtrees are collapsed and redundant vertices are omitted. As an auxiliary notion, let us recall the concept of directed acyclic graph.

Definition 58. (**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 (i.e. a finite path $v_1 \ldots v_n$ with $v_n = v_1$ and (v_i, v_{i+1}) for $i < n$ in E). 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 $(i \neq j)$.

Definition 59. (**Ordered binary decision diagram**)
An *ordered binary decision diagram* (OBDD, for short) over $\langle X, < \rangle$ is a rooted dag with vertex-set V containing two disjoint types of vertices:

- each *non-terminal* vertex v is labelled by a boolean variable $var(v) \in X$ and has two children $z(v), o(v) \in V$
- each *terminal* vertex v is labelled by a boolean value $value(v)$,

such that for all non-terminal vertices $v, w \in V$:

$$w \in \{\, o(v), z(v)\,\} \;\Rightarrow\; (var(v) < var(w) \;\vee\; w \text{ is a terminal vertex}).$$

The (ordering) constraint on the labelling of the non-terminals requires that on any path from the root to a terminal vertex, the variables respect the given ordering $<$. This constraint also guarantees that the OBDD is an acyclic graph. Ordered BDDs are due to (Bryant, 1986). For non-terminal v, the edge from v to $z(v)$ represents the case that $var(v) =$ false (=0); the edge from v to $o(v)$ the case $var(v) =$ true (=1).

Definition 60. (**Reduced OBDD**)
OBDD B over $\langle X, < \rangle$ is called *reduced* if the following conditions hold:

1. for each non-terminal v: $o(v) \neq z(v)$
2. for each non-terminal v, w:

$$(var(v) = var(w) \;\wedge\; o(v) = o(w) \;\wedge\; z(v) = z(w)) \;\Rightarrow\; v = w$$

3. for each terminal v, w: $(value(v) = value(w)) \;\Rightarrow\; v = w$.

The first constraint states that no non-terminal vertex has identical one- and zero-successors. The second constraint forbids vertices to denote isomorphic sub-dags; the last constraint does not allow identical terminal vertices (such that at most two terminal vertices are allowed).

Example 45. *The BDT of Figure 5.2 is an OBDD, but not a ROBDD. For instance, the two subtrees rooted at the vertex labelled with s'_1 in the leftmost*

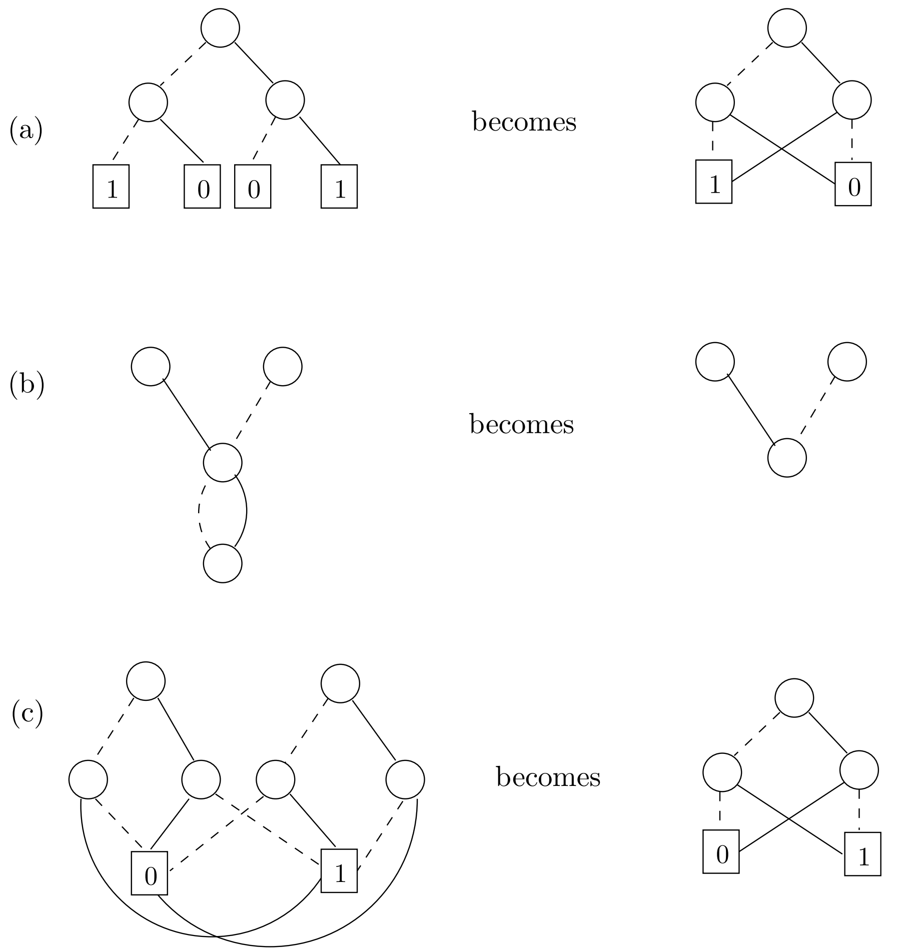


Figure 5.3: Steps to transform an OBDD into reduced form

subtree are isomorphic, and the top-vertices of these subtrees violate the second constraint of being a ROBDD. *(End of example.)*

The definition of reduced OBDD (ROBDD, for short) 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 labelled 0 (or 1), redirect all edges that point to such vertex to one of them, and remove the obsolete terminal vertices, cf. Figure 5.3(a).
- Removal of redundant tests: if both outgoing edges of a vertex v point to the same vertex w, then eliminate vertex v and redirect all its incoming edges to w, cf. Figure 5.3(b).

- Removal of duplicate non-terminals: if two distinct vertices v and w are the roots of isomorphic ROBDDs, then eliminate v (or w), and redirect all incoming edges to the other one, cf. Figure 5.3(c).

The transformation of an OBDD into a 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.

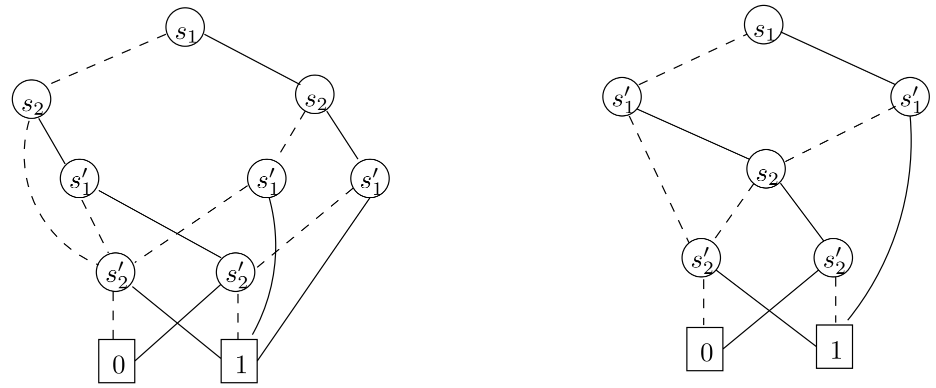
The function represented by a ROBDD is obtained in the same way as for BDTs (cf. Definition 57). Given a total ordering on the boolean variables and some boolean function, there exists a unique ROBDD (up to isomorphism) that represents this function. This important result is due to (Bryant, 1986).

Theorem 61.
Let $X = \{ x_1, \ldots, x_n \}$ be a set of boolean variables and $<$ a total ordering on X. For ROBDDs B_1 and B_2 over $\langle X, < \rangle$ we have:

$$f_{\mathsf{B}_1} = f_{\mathsf{B}_2} \;\Rightarrow\; \mathsf{B}_1 \textit{ and } \mathsf{B}_2 \textit{ are isomorphic.}$$

As a result several computations on boolean functions can be easily decided, such as equivalence of functions — it suffices to check the equality of the ROBDDs. For instance, in order to check whether a ROBDD is always true (or false) for any variable assignment of boolean values to its variables, amounts to simply check equality with a single terminal vertex labelled 1 (or 0), the ROBDD that represents true (or false). For boolean expressions e.g. this problem is NP-complete.

Example 46. *Consider again our running example. The BDT that represents the transition relation R is depicted in Figure 5.2 and consists of $2^5-1 = 31$ vertices. The corresponding ROBDD, that is a ROBDD that represents the same function f_{R}, for variable ordering $s_1 < s_2 < s'_1 < s'_2$ is depicted in Figure 5.4(a). Since the BDT contains a substantial degree of redundancy, 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. $f_{\mathsf{R}}(0, 0, 0, 1)$ is determined* *without using the fact that* $s'_1 = 0$. *Figure 5.4(b) depicts the ROBDD representing the same function, but using a different variable ordering:* $s_1 < s'_1 < s_2 < s'_2$. *This ROBDD consists of 8 vertices and thus exploits the redundancy in the BDT in a more optimal way. This is, for instance, also illustrated by considering the path to traverse in order to determine the value of* $(1, 1, 1, 1)$*: in the left ROBDD three vertices are visited, whereas in the right ROBDD only two.* *(End of*



(a) ordering $s_1 < s_2 < s'_1 < s'_2$ (b) ordering $s_1 < s'_1 < s_2 < s'_2$

Figure 5.4: Two ROBDDs (for different orderings) for our sample CTL-model

example.)

The size of a ROBDD strongly depends on the ordering of the boolean variables under $<$. This is illustrated in Figure 5.4 for a small example. For larger examples, these effects can be substantially larger. For representing the transition relation of finite-state transition systems like CTL-models, 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, if $[\![\, s \,]\!] = (s_1, \ldots, s_n)$ and $[\![\, s' \,]\!] = (s'_1, \ldots, s'_n)$ then the ordering in the ROBDD is

$$s_1 < s'_1 < s_2 < s'_2 < \ldots < s_n < s'_n$$

or its symmetric counterpart $s'_1 < s_1 < \ldots < s'_n < s_n$. This schema has been applied in Figure 5.4(b). Examples exist for which under a given ordering an exponential number of states are needed, whereas under a different ordering of the variables only a linear number of states is needed. Moreover, there do exist boolean functions that have exponential-size ROBDDs for any variable ordering.

Example 47. *Consider the function*

$$f(x_1, \ldots, x_{2n}) = (x_1 \ \vee \ x_2) \ \wedge \ \ldots \ \wedge \ (x_{2n-1} \ \vee \ x_{2n}).$$

Using the variable ordering $x_i < x_{i+1}$*, for* $0 < i < 2n$*, gives rise to a ROBDD with* $2n{+}2$ *vertices. The variable ordering* $x_1 < x_3 < \ldots < x_{2n-1} < x_2 < x_4 < \ldots < x_{2n}$ *gives rise to a ROBDD with* 2^{n+1} *vertices. (The reader is invited to check this for* $n = 3$ *or* $n = 2$*.)* *(End of example.)*

The problem of finding an optimal variable ordering is NP-complete. Most model checkers based on a symbolic representation of the state space using ROBDDs apply heuristics (or dynamic re-ordering of variables) to find a variable ordering that results in a compact representation. In practice, these algorithms work quite well.

The use of ROBDDs does not improve the worst-case time complexity of the model checking algorithms: the worst-case time complexity is identical to that of model checking using explicit state space representations. It is very hard to give exact average-time or best-time complexity figures for ROBDD-model checking. Empirical results show that for many practical cases state spaces can be represented in a compact way by means of ROBDDs (Burch et al, 1992). Experiments with state spaces of 10^{120} states and beyond have been reported in the literature.

Model checking using ROBDDs

The symbolic model checking algorithm for CTL works briefly as follows. Let $\mathcal{M} = (S, R, \pi)$ be a CTL-model that is obtained as system description. $\mathcal{M}$ is represented by a ROBDD R over $2n$ boolean variables, where $2n$ is the cardinality of the state space S. R is constructed as explained in the previous section; e.g. for $\mathcal{M}$ of our running example, the corresponding ROBDD R is given in Figure 5.4(b).

In practice it is often not feasible to obtain a ROBDD-representation starting from an explicit state space enumeration, since the state space is to large to handle. Instead, the ROBDD-representation is directly obtained from a high-level description of the model $\mathcal{M}$.

For each CTL-formula ϕ a ROBDD $[\![\, Sat(\phi) \,]\!]$ is constructed over the boolean variables $(s_1, \ldots, s_n)$ that represents state s. $[\![\, Sat(\phi) \,]\!]$ represents the characteristic function of the set $Sat(\phi) = \{\, s \in S \mid \mathcal{M}, s \models \phi \,\}$. Thus,

$$f_{[\![\, Sat(\phi) \,]\!]}(s_1, \ldots, s_n) = 1 \text{ if and only if } s \in Sat(\phi) \text{ for } [\![\, s \,]\!] = (s_1, \ldots, s_n).$$

Model checking CTL now boils down to constructing the ROBDD $[\![\, Sat(\phi) \,]\!]$ in an iterative way starting with the sub-formulas of ϕ of length 1. In each iteration the results of previous iterations can be exploited. For instance,

$$[\![\, Sat(\phi_1 \ \wedge \ \phi_2) \,]\!] = \textsc{Apply}([\![\, Sat(\phi_1) \,]\!], [\![\, Sat(\phi_2) \,]\!], \wedge\,)$$

where $\textsc{Apply}$ is an operator on ROBDDs, such that for ROBDDs B_1 and B_2, $f_{\mathsf{B}} = f_{\mathsf{B}_1} \oplus f_{\mathsf{B}_2}$ for $\mathsf{B} = \textsc{Apply}(\mathsf{B}_1, \mathsf{B}_2, \oplus)$ with $\oplus$ a binary operator on booleans. Like for model checking CTL without ROBDDs (cf. Chapter 3), the ROBDD $[\![\, Sat(\mathsf{E}\,[\phi \,\mathsf{U}\, \psi]) \,]\!]$ is computed by a least fixed point computation. In a nutshell, one might say that all model checking algorithms of Chapter 3 for computing $Sat(\phi)$ carry over in a direct way to the symbolic case to compute the ROBDD $[\![\, Sat(\phi) \,]\!]$. For a more thorough treatment of model checking using ROBDDs we refer to (McMillan 1993, and Clarke, Grumberg & Long, 1993). Since this type of model checking uses a symbolic representation of the state space, this is referred to as *symbolic model checking* algorithms.

5.2 Memory management strategies

In this section we consider some memory management strategies that intend to reduce the often excessive size of the state space for model checking purposes. These techniques have originally been developed for state space exploration, i.e.

procedure *DepthFirstSearch* (s_0 : *Vertex*);
(* precondition: true *)
begin var S : **sequence of** *Vertex*, (* path from s_0 to current vertex *)
    Z : **set of** *Vertex*; (* visited vertices *)
  $S, Z := \langle s_0 \rangle, \langle\,\rangle, \varnothing$;
  do $S \neq \langle\,\rangle \longrightarrow$ (* let $S = \langle s \rangle ^\frown S'$ *)
    if $\rho(s) \subseteq Z \longrightarrow S := S'$;
    [] $\rho(s) \not\subseteq Z \longrightarrow$ **let** s' **in** $\rho(s) \setminus Z$;
        $S, Z := \langle s' \rangle ^\frown S, Z \cup \{ s' \}$
    fi;
  od;
end

Table 5.1: Depth-first search traversal of state space

reachability analysis. Since model checking PLTL is strongly based on similar reachability analysis methods, like the checking of emptiness of Büchi automata, these memory management strategies will be discussed in the context of PLTL model checking.

Recall from Chapter 2 that the algorithm for checking emptiness of Büchi automata is based on a (nested) depth-first search algorithm. In the first (outermost) depth-first search an accepting state that is reachable from the initial state is computed. In the second (innermost) depth-first search, it is checked whether such accepting state is reachable from itself, i.e. is member of a cycle. If such accepting state is found, the Büchi automaton at hand is non-empty, and the property is invalidated.

The basic idea of the memory management strategies in this section is to reduce the amount of memory used in this depth-first search strategy as much as possible. These techniques are not only applicable to depth-first search strategies, but can also be applied to other search strategies like breadth-first search. In order to discuss these techniques let us have a more detailed look at a typical (unnested) depth-first search algorithm, see Table 5.1. The two basic data structures in this algorithm are a sequence of states S (usually implemented as a stack) and a set of states Z. Recall that $\rho(s)$ denotes the set of direct successors of s. The last state in sequence S denotes the state that is currently being explored while Z records the states that have been considered (visited). During the search, once states have been explored they are stored in Z. Storing states avoids redundant explorations of parts of the state space: if a state in Z is encountered again later in the search, it is not necessary to re-explore all its successors. (If we would know in advance that each state is visited at most once on exploring the state space, Z could be omitted, but this knowledge is absent in practice.) To check whether an encountered state is in Z is referred to as *state matching*.

Measurements of traditional reachability analysis methods have shown that the most time-consuming operation is not the generation of states nor the analysis, but the storage of states and state matching (Holzmann, 1988). Since accesses to sequence S are predictable — only the "top" element of S is of importance — this data structure can be stored in secondary memory without slowing down the computations significantly. Accesses to Z, however, are random, and thus Z is stored in main memory. The next two techniques are focussed on storing Z, or part of it, in a suitable manner, such that the memory usage is reduced (since main memory is typically a scarce resource), and that state matching can be done efficiently.

Bit-state hashing

In order to discuss the bit-state hashing technique (which is due to Holzmann, 1988) let us briefly describe the traditional storage of states. To do so, consider a parallel (or distributed) system with n processes, k channels that connect these processes such that they can exchange information by means of message passing, and m global variables. A state of such system, usually referred to as *state descriptor*, consists of several components: the control location ("program counter" plus values of local variables) for each process, the content of each channel, and the value of each global variable. For N states per process, channels of capacity K and range-size of each global variable M, this leads to maximally $N^n \times K^k \times M^m$ states. Typically this leads to a huge number of states and long state descriptors. State matching thus amounts to comparing two (rather long) state descriptors. The main problem though is the poor coverage of this explicit state enumeration method: in case of e.g. a main memory of capacity 100 Mbyte, state descriptors of 1000 bytes, and 10^6 reachable states, a coverage of 0.1 is obtained; i.e. only 10% of the reachable states is consulted!

> *By bit-state hashing a state descriptor is mapped onto a memory address, that points to a single bit indicating whether the state has been visited before.*

After having provided the mapping for state descriptors onto addresses in main memory, state descriptors are ignored, and thus never need to be stored: instead addresses will represent them. The bit array represents the set Z of before: it equals 1 for a given address (= state) if that state has been visited already and equals 0 otherwise. Thus for state s, function h that maps states onto memory addresses and bit-array *bitar*, $\mathit{bitar}[h(s)] = 1$ if and only if state s has been visited before. Initially all values in the bit-array equal 0, and if state s is visited (that is, added to Z in the code of Table 5.1) $\mathit{bitar}[h(s)]$ is assigned value 1. State matching simply amounts to checking whether $\mathit{bitar}[h(s)] = 1$, a simple table look-up. For an internal memory size of, for instance, 8 Mbyte (= 67 Mbit), this strategy can handle state descriptors of up to 26 bits.

For practical systems, however, state descriptors may be much longer, and the number of necessary addresses exceeds the addressing capacities of the main memory of the machine at our disposal by a large factor. For instance, a state descriptor of 520 bits — which is not exceptional — exceeds the addressing capabilities of an 8 Mbyte machine by a factor $520/26 = 20$ times. Thus, although the above scheme seems interesting, its usage is still limited. Empirical results have shown, however, that for communication protocols with a total state space of e.g. 10^9 states, the number of reachable states is a factor 29 lower (Holzmann, 1988). This is due to the fact that state spaces are typically very sparse: the ratio of the total number of states over the number of reachable states is high! Based on this observation, the idea is to store only a subset of all possible state descriptors. In this way, a *probabilistic* technique is obtained: there is a positive probability that a reachable state is not covered, since it does not belong to the selected subset of state descriptors to be stored. Due to the above observation, though, the probability of such event is low.

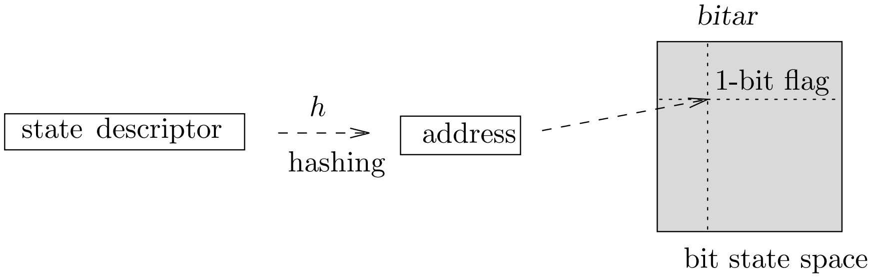


Figure 5.5: Basic principle of bit-state hashing

Technically, this scheme is achieved by letting h a (non-perfect) *hash-function*. This scheme is illustrated in Figure 5.5. (In fact, in the aforementioned scheme where the number of addresses is assumed to be equal to the number of reachable state-descriptors, h is a perfect hash-function, i.e. h is injective). As before, the function h maps a state descriptor (called hash-key) onto a position (address) in the bit-array. Since the number of addresses is significantly smaller than the number of reachable states, so-called hash-collisions may appear. Mathematically speaking, h may be non-injective. That is, for states s, s' with $s \neq s'$ it is possible that $h(s) = h(s')$. Consequently, one might wrongly decide that a state has been visited while it has not: if state s has been visited (thus $\mathit{bitar}[h(s)] = 1$) and state $s' \neq s$ is encountered with $h(s) = h(s')$, state s' is not considered. Consequently, also the successors of s' might not be considered. Since state descriptors are not stored, but addresses are used instead, there is no way to resolve the hash-collision. Notice that although a hash-collision may cause the truncation of a search path at a too early stage, it will not yield wrong results.

Thus, this scheme is actually providing a *partial* state space search. The probability p that no collisions appear is approximated as (Wolper and Leroy, 1993):

$$p \approx e^{-\frac{N^2}{2^k}}$$

where N is the number of states and k the number of stored bits. Several proposals have been made to reduce this probability, for instance by only storing states that have been visited before (Wolper and Leroy, 1993), the use of two hash functions with a sophisticated collision resolution scheme (Stern and Dill, 1996), and the use of dynamic hashing using two hash functions (Knottenbelt, Mestern, Harrison and Kritzinger, 1998).

The effect of bit-state hashing on the coverage of the problem at hand is illustrated in Figure 5.6. The curves show that with a considerable smaller amount of available memory, the coverage of the state space with bit-state hashing outperforms that with conventional (i.e. explicit state space storge) significantly. Here, the bit-state hashing algorithm has been applied to a data transfer protocol that requires the generation of about 427,000 states each taking 1.3 Kbits of memory for the state descriptor in case of an exhaustive state search. The total memory requirements are close to 73 Mbytes of memory (about 2^{29} bits). The bit-state hashing techniques have been applied with good results in several large-scale industrial case studies (Holzmann, 1998).

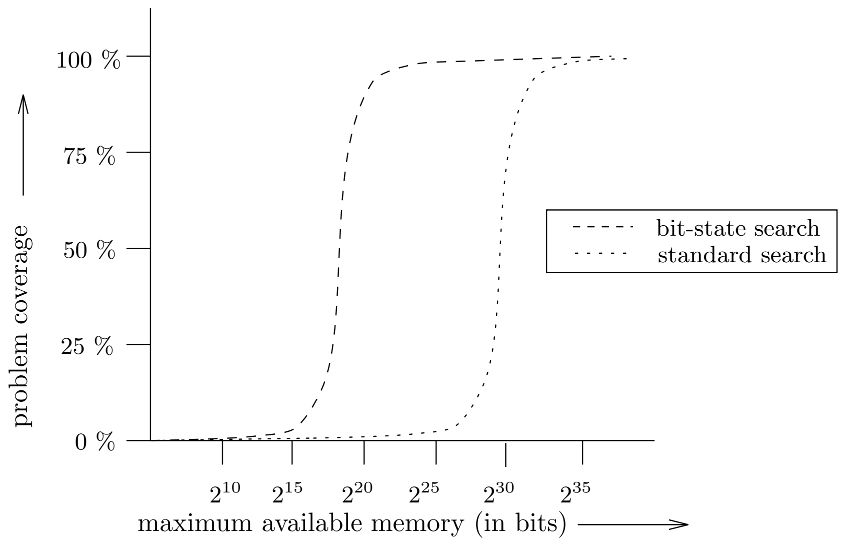


Figure 5.6: Effect of bit-state hashing on problem coverage

State-space caching

Another technique for coping with the state space explosion problem in depth-first search is called *state space caching* (Holzmann, 1985).

State-space caching is based on the idea to maintain a cache (a fast memory of restricted size) of states that have been visited before.

Initially, the cache is empty and all visited states are stored in the cache until the cache is fully occupied. If a new state is to be added to the full cache a previously inserted state in the cache is selected and removed from the cache, and the new state is added. Different policies to select the state to be removed from the cache can be adopted: least recently used, less frequently visited state, and so on. Studies have shown that state-space caching is only useful for exploring state spaces that are only a few times larger than the size of the cache. The critical point of this approach is that a previously visited state that has been removed from the cache is encountered again in the search, and explored again. By experiments it has turned out that the number of transitions is about 3 times the number of states in a model, and thus each state is encountered about 3 times on average during a search. The efficiency of this technique can be increased significantly if the number of visits to a state can be reduced. A possible technique for this purpose is partial-order reduction, described in the next section. The combination of state-space caching and partial-order reduction is implemented in the model checker SPIN and experiments indicate a significant reduction of the memory requirements (Godefroid, Holzmann & Pirottin, 1995).

5.3 Partial-order reduction

Systems that consist of a set of components that cooperatively solve a certain task are quite common in practice, e.g. hardware systems, communications protocols, distributed systems, and so on. Typically, such systems are specified as the parallel composition of n processes P_i, for $0 < i \leqslant n$. The state space of this specification equals in worst case $\prod_{i=1}^{n} | P_i |$, the product of the state spaces of the components. A major cause of this state space explosion is the representation of parallelism by means of *interleaving*. Interleaving is based on the principle that a system run is a totally ordered sequence of actions. In order to represent all possible runs of the system, all possible interleavings of actions of components need to be represented and, consequently, parallel composition of state spaces amounts to building the Cartesian product of the state spaces of the components. For n components, thus $n!$ different orderings (with 2^n states) are explicitly represented. In a traditional state space exploration all these orderings are considered. For checking a large class of properties, however, it is sufficient to check only some (in most of the cases even just one) representative of all these interleavings. For example, if two processes both increment an integer variable in successive steps, the end result is the same regardless of the order in which these assignments occur. The underlying idea of this approach is to reduce the interleaving representation into a *partial-order* representation. System runs are now no longer totally ordered sequences, but partially ordered sequences. In the best case, the state space of the parallel composition of n processes P_i now reduces to $\sum_{i=1}^{n} | P_i |$ and only a single ordering of $n+1$ states needs to be examined.

For checking a property ϕ, partial-order reduction methods explore only a reduced part of the state space that is (provably) sufficient to check the validity of ϕ.

The difference between the reduced and the global state space is that not all interleavings of concurrent activities are represented in the reduced one. Which interleavings need to be preserved may depend on the property to be checked. The basic requirement for the reduced state space is that it contains sufficient interleavings to enable checking of the desired property. For performing model checking it is then sufficient to consider the reduced state space.

There are basically three different techniques for partial-order approaches towards model checking.

1. In *dynamic* partial-order reduction a subset of the states that need to be explored is determined *during* the search procedure. This embodies that the possible interesting successors of a given state are computed while exploring the current state. Dynamic partial-order reduction is the most popular and most developed technique; it requires, however, a modification of the search procedure[1].

[1]For instance, for SPIN the size of the code of the model checker was doubled due to the incorporation of dynamic partial-order reduction.

2. *Static* partial-order reduction techniques avoid the modification of the search engine of the model checker by performing the reduction at the level of the system specification, that is, syntactically. The subset of the states that need to be explored is thus determined *before* the actual searching takes place.

3. In the *purely* partial-order approach, the construction of the interleaved expansion of parallel components is avoided by directly constructing a partial-order representation of the state space from the system specification. An interleaving representation is thus never generated, and needs never to be stored. This approach is particularly suitable for Petri nets.

In this chapter we will treat the basic principles of dynamic partial-order reduction; for static partial-order reduction we refer to (Kurshan et al, 1998), while for the entirely partial-order approach we refer to (McMillan, 1995 and Esparza, 1994).

In the linear temporal case, dynamic partial-order reduction amounts to traversing the reduced state space to check for emptiness (of the product Büchi automaton), that is, reachable cycles that contain accepting states. Such traversal starts in the non-reduced case from the initial state and successively considers new states by exploring all successors of the current state s (cf. Table 5.1). In partial-order reduction only a subset of these successors are considered. Such a partial exploration of the entire state space is called a *selective search*.

There are different techniques for performing a selective search and they basically differ in the way in which they select the representative interleavings. Amongst others, notions like ample sets, persistent sets, sleep sets and stubborn sets have been defined. The basic idea of these concepts is to consider only those successor states that are reachable via dependent transitions, that is, to *ignore* independent transitions. Intuitively, independent transitions are transitions whose effects are the same, irrespective of their order. For instance, the execution of two independent transitions in either order both must lead to the same resulting state. In the sequel of this section we will deal with the ample sets approach (Peled, 1996). This approach can be used in the context of on-the-fly model checking for PLTL as explained in Chapter 2.

Selective search using ample sets

For reducing the state space, transitions play a prominent role. In particular one has to decide which transitions are dependent and which are not. In order to distinguish between the different transitions in a system, we replace R, the successor relation, by a set of transitions T. Each element $a \in T$ is referred to as a transition (rather than a transition relation).

Definition 62. (**State transition system**)
$\mathcal{M} = (S, T, \mathit{Label})$ is a *transition system* with S a non-empty, finite set of states, $\mathit{Label} : S \rightarrow 2^{AP}$, the labelling of states with sets of atomic propositions, and T a set of transitions such that for each $a \in T$, $a \subseteq S \times S$.

A CTL-model can be defined as a triple (S, R, Label) where $R(s, s')$ if and only if there exists a transition $a \in T$ such that $a(s, s')$. We consider deterministic transitions, i.e. transitions a such that for each state s there is at most one state s' such that $a(s, s')$. Notice that this is not a real restriction, since non-deterministic transitions can be modeled by a set of transitions: e.g. if $R(s, s')$ and $R(s, s'')$, then let $a, b \in T$ with $a \neq b$ such that $a(s, s')$ and $b(s, s'')$. For deterministic transition a we often denote $s' = a(s)$ as shorthand for $a(s, s')$.

Definition 63. (**Enabled and disabled transitions**)
Transition a is *enabled* in s if $\exists\, s' \in S.\, a(s, s')$, otherwise a is *disabled* in s. The set of transitions that is enabled in s is denoted $\mathit{enabled}(s)$.

As we have stated above, the idea is to perform a selective search, such that only a subset of the enabled transitions is explored. Let for state s, this subset be denoted $\mathit{ample}(s)$. Thus, $\mathit{ample}(s) \subseteq \mathit{enabled}(s)$. The basic skeleton of a selective depth-first search is given in Table 5.2. In a similar way the algorithm in Chapter 2 for checking the emptiness of a Büchi automaton can be modified such that only a selective search is performed. The results of this partial search coincide with the results of the exhaustive search with a small difference: since transitions are removed from the reduced graph, the counter-example that is generated in case a failure is detected might differ from the counter-example obtained with the exhaustive search algorithm.

procedure *SelectiveSearch* (s_0 : *State*);
(* precondition: true *)
begin var S : **sequence of** *State*, (* path from s_0 to current state *)
  Z : **set of** *State*; (* visited states *)
  $S, Z := \langle s_0 \rangle, \varnothing$;
  do $S \neq \langle\,\rangle \longrightarrow$ (* let $S = \langle s \rangle ^\frown S'$ *)
    if $\{\, a(s) \mid a \in \mathit{ample}(s) \,\} \subseteq Z \longrightarrow S := S'$;
    $[]\ \{\, a(s) \mid a \in \mathit{ample}(s) \,\} \nsubseteq Z \longrightarrow$
      let s' **in** $\{\, a(s) \mid a \in \mathit{ample}(s) \,\} - Z$;
      $S, Z := \langle s' \rangle ^\frown S, Z \cup \{\, s' \,\}$
    fi
  od
end

Table 5.2: Selective depth-first search using ample sets

The correctness of the selective search algorithm critically depends on the calculation of $\mathit{ample}(s)$; that is, sufficiently many transitions must be included in this set so that the obtained results coincide with the results when $\mathit{enabled}(s)$ would be used instead. This requirement can of course, be simply fulfilled by letting $\mathit{ample}(s) = \mathit{enabled}(s)$, but this is not the intention: the number of states in $\mathit{ample}(s)$ should be preferably significantly smaller than the cardinality of $\mathit{enabled}(s)$. Moreover, the ample sets need to computed with small overhead.

Independence and invisibility

As stated before, the idea is to ignore consider from a pair of independent transition only one. The notion of independence is defined as follows.

Definition 64. (**Independence relation on transitions**)
Relation $I \subseteq T \times T$ is called an *independence relation* if it is symmetric (i.e. $(a, b) \in I \;\Rightarrow\; (b, a) \in I$) and irreflexive (i.e. $(a, a) \notin I$) such that for each $s \in S$ and for each $(a, b) \in I \cap \mathit{enabled}(s)$:

1. $a \in \textit{enabled}(b(s))$, and
2. $a(b(s)) = b(a(s))$.

The complement of I (that is, $(T \times T) \setminus I$) is referred to as the dependency relation. The first condition states that a pair of independent transitions must not disable each other (but they may enable each other): if a and b are enabled in s, then after taking a, b is still enabled in the resulting state after a. The second condition states that executing independent transitions in either order results in the same state.

The latter condition suggests a possible reduction of the state space: if the orderings ab and ba are possible and transitions a, b are independent, then simply ignore one of the possibilities, since the resulting state is the same. This naive recipe is, however, not guaranteed to obtain correct results. For instance, the property to be checked might be sensitive on the intermediate state reached after first a has been executed, while this state is not consulted if the alternative ab is omitted. To determine the intermediate states that are irrelevant for a property, the notion of invisibility of a transition is introduced.

Definition 65. (**Invisible transition**)
Let AP be a set of atomic propositions and $AP' \subseteq AP$. Transition $a \in T$ is *invisible* with respect to AP' if for all $s, s' \in S$ such that $s' = a(s)$ we have: $\textit{Label}(s) \cap AP' = \textit{Label}(s') \cap AP'$. Otherwise, a is called *visible*.

Stated in words: a transition is invisible with respect to AP' if it does not change the validity of the atomic propositions in AP'.

Definition 66. (**Stuttering equivalence**)
Infinite paths $\sigma = s_0\, s_1 \ldots$ and $\sigma' = t_0\, t_1 \ldots$ are *stuttering equivalent*, denoted $\sigma \sim_{st} \sigma'$, if there are two infinite sequences of integers $i_0 < i_1 < \ldots$ and $j_0 < j_1 < \ldots$ such that $i_0 = j_0 = 0$ and for all $k \geqslant 0$:

$$
\begin{array}{rl}
& \textit{Label}(\sigma[i_k]) = \textit{Label}(\sigma[i_k{+}1]) = \ldots = \textit{Label}(\sigma[i_{k+1}{-}1]) \\
= & \textit{Label}(\sigma'[j_k]) = \textit{Label}(\sigma'[j_k{+}1]) = \ldots = \textit{Label}(\sigma'[j_{k+1}{-}1]).
\end{array}
$$

Intuitively, sequences σ and σ' are stuttering equivalent if they can be partitioned into infinitely many blocks, finite sequences of equally labelled states, such that the k-th block of σ is equally labelled to the k-th block of σ'.

Example 48. Consider the (part of the) transition system of Figure 5.7 and let $a = \{ (s_0, s_1), (s_2, s_3) \}$ *and* $b = \{ (s_0, s_2), (s_1, s_3) \}$. *Then we have* $s_0\, s_1\, s_3 \sim_{st} s_0\, s_2\, s_3$*: block* $s_0\, s_1$ *is matched with* s_0 *and block* s_3 *is matched with* $s_2\, s_3$. *(End*

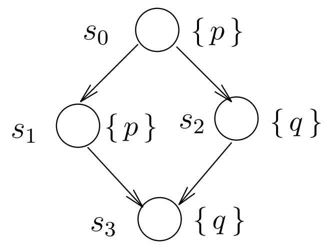


Figure 5.7: Example of stuttering equivalent paths

of example.)

The basic idea of partial-order reduction is to explore of each equivalence class of stuttering-equivalent paths only one representative.

This is guaranteed to be correct for the set of PLTL-formulas whose validity is identical for each stuttering equivalent sequence. Such formulas are said to be invariant under stuttering.

Definition 67. (**Invariance under stuttering**)
PLTL-formula ϕ is *invariant under stuttering* if and only if for all paths $\sigma, \widehat{\sigma}$:

$$\sigma \sim_{st} \widehat{\sigma} \;\Rightarrow\; (\mathcal{M}, \sigma[0] \models \phi \text{ if and only if } \mathcal{M}, \widehat{\sigma}[0] \models \phi).$$

Is it possible to characterize the set of formulas that are invalid under stuttering? It is not difficult to see that a formula like $\mathsf{X}\, p$, for atomic proposition p, is not invariant under stuttering: suppose σ is the path $s_1\, s_1\, s_2\, s_1\, s_1 \ldots$ and $\widehat{\sigma}$ equals $s_1\, s_2\, s_1 \ldots$. These two paths are stuttering equivalent: each subsequence

$s_1\, s_1$ in σ is matched with the subsequence s_1 in $\widehat{\sigma}$, whereas s_2 in σ is matched with s_2 in $\widehat{\sigma}$. Assume p is valid in s_2, and not in s_1. Then $\mathsf{X}\, p$ is valid in the initial state of $\widehat{\sigma}$, but not in the initial state of σ, although $\sigma \sim_{st} \widehat{\sigma}$. In fact, X is the only temporal operator in PLTL that disturbs validity under stuttering. The following results are due to (Lamport, 1983) and (Peled and Wilke, 1997), respectively.

Theorem 68.

1. *Any* PLTL*-formula that does not contain any next operator is invariant under stuttering.*
2. *Any* PLTL*-formula that is invariant under stuttering can be expressed without next operators.*

Characterization of ample sets

When the property at hand is invariant under stuttering, the independence relation and the notion of invisibility allow us to avoid the exploration of certain paths in the state space. The idea of the ample sets is to guarantee that for each path that is not considered by the selective search algorithm, there is a stuttering equivalent path that is considered. The reduction will depend on the set of atomic propositions that appears in the PLTL-formula to be checked.

The characterization of ample sets is given by defining successive constraints; subsequently we discuss how these sets can be computed such that the constraints are fulfilled. The constraints on ample sets are (Peled, 1996):

1. $\mathit{ample}(s) = \varnothing$ if and only if $\mathit{enabled}(s) = \varnothing$.
2. For each path starting at state s in the original state space, the following condition holds: a transition dependent on a transition in $\mathit{ample}(s)$ cannot be taken without some transition in $\mathit{ample}(s)$ taken first.
3. $\mathit{ample}(s) \neq \mathit{enabled}(s) \;\Rightarrow\;$ each transition in $\mathit{ample}(s)$ is invisible.

4. A cycle is not allowed if it contains a state in which a is enabled, but is never included in $ample(s)$ for any state s in the cycle.

The first constraint says that if a state has at least a single outgoing transition, then it should have so in the reduced state space. The second constraint implies that only transitions that are independent of $ample(s)$ can be taken before some transition in $ample(s)$ is taken. (It is even the case that by taking such independent transition first, this does not disable any of the transitions in $ample(s)$.) The last two constraints are necessary to guarantee that all paths in the original state space are represented by some (stuttering-equivalent) path in the reduced state space. The justification of these constraints is beyond the scope of these lecture notes.

Given this characterization the question is how to compute these ample sets in an effective and (if possible) efficient manner. The first and third condition can easily be checked; checking the last constraint is hard, whereas checking the second condition is at least as hard as exploring the entire state space. To resolve this, the last constraint is usually replaced by a stronger condition that facilitates an efficient computation of ample sets. Different strengthenings are possible. (Peled, 1996) proposes to strengthen the last constraint by: $ample(s) \neq enabled(s) \Rightarrow$ no transition in $ample(s)$ may close a cycle. (This can simply be checked: if a state is explored that is already in the sequence S in the depth-first search, it closes a cycle.) Since checking the second constraint is as hard as exploring the full state space — which we would like to avoid! — the usual idea is to construct ample sets that guarantee this constraint by construction, thus avoiding checking it. The precise way in which this is done strongly depends on the model of computation and is not treated here.

Experimental results

Partial-order reduction is for instance realized in the linear temporal logic model checker Spin, in combination with techniques like bit-state hashing (as an option) and state-space caching, as explained in the previous section. More precisely, Spin computes the non-emptiness of the product Büchi automaton on-the-fly;

that is, it checks whether there exists an accepting run in this automaton whereby a new state in the automaton is generated only if no accepting run has been found yet. Some care needs to be taken by combining the partial-order reduction with the nested depth-first search such that in each depth-first search the same ample sets are chosen, for instance, to ensure that cycles are closed (second constraint above) at the same place.

As an example the leader election protocol of Dolev, Klawe and Rodeh is taken. For an extensive explanation of this protocol we refer to Chapter 2. For three processes the unreduced state space consists of about 16,000 states (1.8 Mbyte), whereas the reduced state space consists of about 1,450 states (1.5 Mbyte). This difference increases significantly when the number of processes is increased to 4: 522,000 states (15.7 Mbyte) versus 8,475 states (1.7 Mbyte) in the reduced case. For 6 processes, the unreduced state space could not be generated, whereas for the reduced case 434,000 states (15.6 Mbyte) were needed. This shows how partial-order reduction enables to alleviate the state space explosion, in particular for systems with many parallel composed components. The effect of partial order reduction is illustrated in Figure 5.8

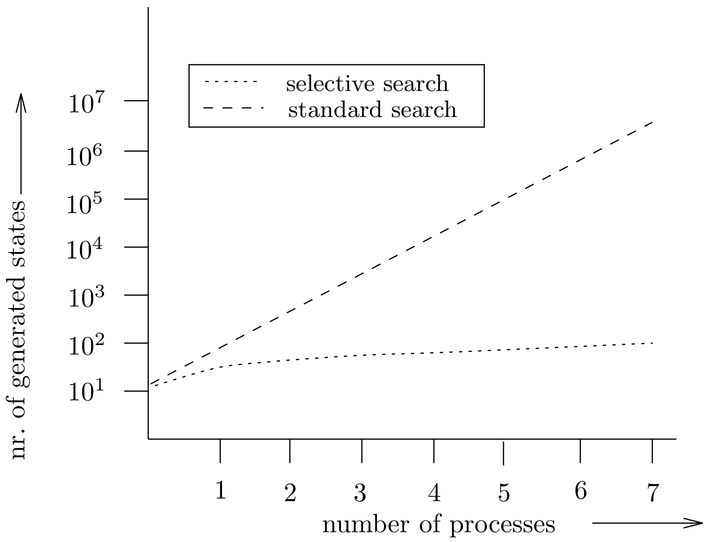


Figure 5.8: Effect of partial order reduction on leader election protocol

This example provides a best-case performance for partial-order reduction, where exponential growth of the state space in the number of processes participating in the protocol is reduced to a linear growth. In more typical case, the reduction in the state space size and in the memory requirements is linear in the size of the model, yielding savings in memory and run-time from 10 to 90% (Holzmann and Peled, 1994).

5.4 Reduction through equivalences

The basic idea of reduction through equivalences is to replace a state space by an equivalent state space that contains fewer states and that is invariant under the validity of all formulas. (Actually, partial-order reduction is a specific variant of these techniques where a model is reduced into stuttering-equivalent model.) More precisely, the idea is to find an equivalence relation $\sim$ between models such that for any formula ϕ:

$$\mathcal{M} \sim \mathcal{M}' \;\Rightarrow\; (\mathcal{M}, s \models \phi \text{ if and only if } \mathcal{M}', s \models \phi).$$

That is, if two models are equivalent, then the validity of each formula in the logic at hand is identical. Now suppose we want to check whether $\mathcal{M}, s \models \phi$. In case $\mathcal{M}'$ is smaller than $\mathcal{M}$ and $\mathcal{M} \sim \mathcal{M}'$, model checking formula ϕ is performed on $\mathcal{M}'$ and state s' where s and s' are equivalent states.

For CTL such equivalence notion is bisimulation equivalence. Informally speaking two models are bisimilar if each transition in the one model can be simulated by the other and vice versa.

Definition 69. (**Bisimulation equivalence**)
Let $\mathcal{M} = (S, R, \mathit{Label})$ and $\mathcal{M}' = (S', R', \mathit{Label}')$ be two CTL-models over the same set of atomic propositions AP. Relation $\mathcal{B}$ is a *bisimulation* if and only if for all $(s, s') \in \mathcal{B}$ we have:

1. when $R(s, s_1)$ then $\exists\, s'_1 \in S'.\, R'(s', s'_1)$ such that $(s_1, s'_1) \in \mathcal{B}$, and
2. when $R'(s', s'_1)$ then $\exists\, s_1 \in S.\, R(s, s_1)$ such that $(s'_1, s_1) \in \mathcal{B}$, and
3. $\mathit{Label}(s) = \mathit{Label}'(s')$.

$\mathcal{M}$ and $\mathcal{M}'$ are said to be *bisimulation equivalent*, denoted $\mathcal{M} \sim \mathcal{M}'$, if there exists a bisimulation $\mathcal{B}$ for any initial state s_0 of $\mathcal{M}$ and initial state s'_0 of $\mathcal{M}'$ such that $(s_0, s'_0) \in \mathcal{B}$.

Stated in words, states s and s' are bisimilar if they are labelled with the same atomic propositions, and if all possible outgoing transitions of s can be simulated by transitions emanating from s', and vice versa. It follows by the above definition that bisimulation is an equivalence relation. The notion of bisimulation can be extended to fair CTL-models (cf. Chapter 3) by requiring that each fair path starting at s in $\mathcal{M}$ can be simulated by a bisimilar fair path starting st s' in $\mathcal{M}'$.

Example 49. Consider the following two models $\mathcal{M}$ and $\mathcal{M}'$. These models are bisimilar; in particular, states that have an identical shading are bisimilar. Since for each state $s \in \mathcal{M}$ a bisimilar state $s' \in \mathcal{M}'$ exists, $\mathcal{M} \sim \mathcal{M}'$. (End of

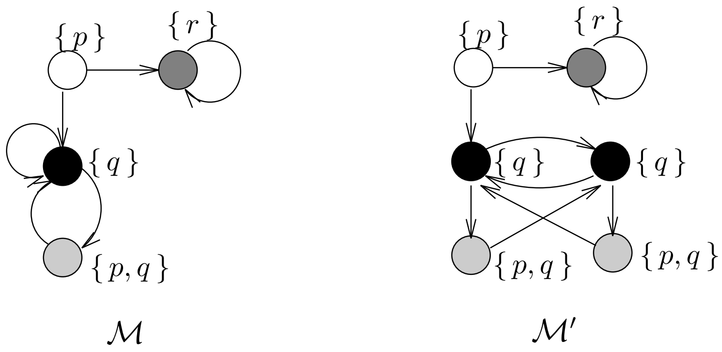


Figure 5.9: Two bisimilar CTL-models

example.)

Theorem 70.
For any CTL-formula ϕ and CTL-models $\mathcal{M}$ and $\mathcal{M}'$:*

$$\mathcal{M} \sim \mathcal{M}' \Leftrightarrow (\mathcal{M}', s' \models \phi \text{ if and only if } \mathcal{M}, s \models \phi).$$

This is in fact a strong preservation result, since both negative and positive results carry over: model checking of $\mathcal{M}'$ is complete with respect to $\mathcal{M}$, since if $\mathcal{M}' \models \phi$ then $\mathcal{M} \models \phi$ (positive result) and if $\mathcal{M}' \not\models \phi$ then $\mathcal{M} \not\models \phi$ (negative result). In case the model $\mathcal{M}'$ is much smaller than $\mathcal{M}$, it is therefore beneficial to take $\mathcal{M}'$ instead for model checking, knowing that all results of this (faster) validation carry over to the original case.

Notice that the implication in the reverse direction also holds: if the validity of all CTL^*-formulas is the same for two given models, then these models are bisimulation equivalent. To decide whether two given models are bisimulation equivalent, an algorithm based on *partition refinement* can be used (Paige and Tarjan, 1987). For a model with $\mid S \mid$ states and $\mid R \mid$ transitions, the worst-case time complexity of this algorithm is $\mathcal{O}(\mid R \mid \times \log \mid S \mid)$.

In case no significant reduction can be obtained using bisimulation equivalence, a more substantial reduction of the state space can be obtained by (1) restricting the logic, and (2) relaxing the requirement that the validity of all formulas is invariant for all equivalent models. An example of such approach is to restrict the logic by disallowing existential path quantifiers, and by using a pre-order relation (that is, a reflexive and transitive relation) instead of an equivalence relation between models. For an appropriate pre-order relation $\trianglelefteq$ — usually a simulation relation (a "half" bisimulation) — the idea is to obtain:

$$\mathcal{M} \trianglelefteq \mathcal{M}' \;\Rightarrow\; (\mathcal{M}' \models \phi \;\Rightarrow\; \mathcal{M} \models \phi)$$

for arbitrary ϕ that does not contain any existential path-quantifier. Note that this is a weak preservation of validity: if we check ϕ on $\mathcal{M}'$ then if $\mathcal{M}' \models \phi$ we also have $\mathcal{M} \models \phi$, but if $\mathcal{M}' \not\models \phi$, this does not imply that the original model $\mathcal{M}$ does not satisfy ϕ. Thus, in this case positive results carry over, but negative results do not!

Exercises

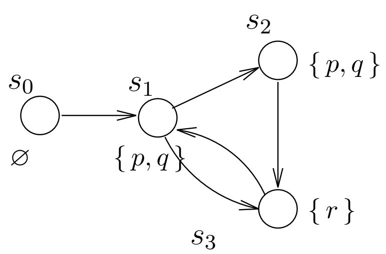
Exercise 33. Let F be a function on tuples of 4 boolean variables which is defined by $F(x_1, x_2, y_1, y_2) = (x_1 \Leftrightarrow y_1) \;\wedge\; (x_2 \Leftrightarrow y_2)$.

1. Construct a ROBDD B with ordering $x_1 < x_2 < y_1 < y_2$ that represents F.
2. Construct a ROBDD B' with ordering $x_1 < y_1 < x_2 < y_2$ that represents F.
3. Argue that both ROBDDs indeed represent F, i.e. prove that $f_{\mathsf{B}_1} = f_{\mathsf{B}_2} = F$.

EXERCISE 34. Let F be a function on tuples of n ($n \geqslant 1$) boolean variables which is defined by $F(x_1, \ldots, x_n) = 1$ if and only if the number of variables x_i with value 1 is even. Construct the ROBDD for this function for $n = 4$ using the ordering $x_1 < \ldots < x_n$.

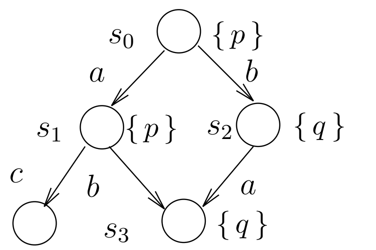
EXERCISE 35. Let $s \in S$. Prove that all transitions in $\mathit{enabled}(s) - \mathit{ample}(s)$ are independent of those in $\mathit{ample}(s)$. That is, for all $a \in \mathit{enabled}(s) - \mathit{ample}(s)$ and $b \in \mathit{ample}(s)$ prove that $(a, b) \in I$. (Hint: by contradiction.)

EXERCISE 36. Consider the Kripke structure $\mathcal{M}$ depicted below:



Let $\sigma = s_0\, s_1\, s_1\, s_1\, s_3\, s_1\, s_3\, s_1\, s_1\, s_1\, s_1\, s_2\, s_3$ a finite path through $\mathcal{M}$ and $\widehat{\sigma} = s_0\, s_1\, s_2\, s_3\, s_1\, s_2\, s_3\, s_1\, s_2\, s_3$. Are these two paths stuttering equivalent or not? If so, indicate the blocks that correspond.

EXERCISE 37. Consider the following part of an original (i.e. non-reduced) state space:



Let $(a, b) \in I$ and assume $ample(s_0) = \{ b \}$ (thus, state s_1 is not visited during the search). Show using the conditions on ample sets that:

(a) $c \in enabled(s_3)$

(b) $a\,c \sim_{st} b\,a\,c$.

5.5 Selected references

Binary decision diagrams:

- S.B. Akers. On a theory of boolean functions. *J. SIAM*, 1959.
- R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* **C-35**(8): 677-691, 1986.
- H.R. Andersen. An introduction to binary decision diagrams. Technical Report, Dept. of Computer Science, Technical University of Denmark, 1994.

Model checking CTL using ROBDDs:

- E.M. Clarke, O. Grumberg, D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency–Reflections and Perspectives*, LNCS 803, pages 124–175, 1993.
- K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

Hashing for state space exploration:

- G.J. Holzmann. An improved protocol reachability analysis technique. *Software–Practice and Experience* **18**(2): 137–161, 1988.

- P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Computer-Aided Verification*, LNCS 697, pages 59–70, 1993.
- U. Stern and D. Dill. A new scheme for memory-efficient probabilistic verification. In *Formal Description Techniques IX–Theory, Applications and Tools*, pages 333–348, 1996.
- W. Knottenbelt, M. Mestern, P.G. Harrison and P. Kritzinger. Probability, parallelism and the state space exploration problem. In *Computer Performance Evaluation*, LNCS 1469, 1998.

State-space caching:

- G.J. Holzmann. Tracing protocols. *AT&T Technical Journal* **64**(12): 2413–2434, 1985.
- P. Godefroid, G.J. Holzmann and D. Pirottin. State-space caching revisited. *Formal Methods in System Design* **7**: 227–241, 1995.

Partial-order reduction:

- D. Peled. Combining partial order reductions with on-the-fly model checking. *Formal Methods in System Design* **8**: 39–64, 1996.
- P. Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-space explosion problem.* LNCS 1032, 1996.
- R. Kurshan, V. Levin, M. Minea, D. Peled and H. Yenigün. Static partial order reduction. In *Tools for the Construction and Analysis of Systems*, LNCS 1394, pages 345–357, 1998.
- G.J. Holzmann and D. Peled. An improvement in formal verification. In *Formal Description Techniques*, pages 177–194, 1994.
- R. Gerth, R. Kuiper, D. Peled and W. Penczek. A partial order approach to branching time temporal logic model checking. In *Proc. 3rd Israel Symposium on Theory of Computing and Systems*, pages 130–139, 1995.

Model checking based on pure partial orders:

- J. Esparza. Model checking using net unfoldings. *Science of Computer Programming* **23**: 151–195, 1994.
- K.L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design* **6**: 45–65, 1995.

Stuttering equivalence:

- L. Lamport. What good is temporal logic? In *Proc. IFIP 9th World Computer Congress Information Processing '83*, pages 657–668, 1983.
- D. Peled and T. Wilke. Stutter-invariant temporal properties are expressible without the next-time operator. *Information Processing Letters* **63**: 243–246, 1997.

Partition refinement:

- R. Paige and R. Tarjan. Three efficient algorithms based on partition refinement. *SIAM Journal on Computing* **16**, 1987.

Model checking using abstract interpretation:

- D. Dams, R. Gerth and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems* **19**(2): 253–291, 1997.