System Verification and Testing

Workbook

Open University of the Netherlands
Faculty of Management, Science & Technology

Course team
dr. ir. J.J.A. Keiren, *course director and author*
dr. F. Verbeek, *author*
dr. N. van Vugt-Hage, *author*

Program director
prof. dr. M.C.J.D. van Eekelen

WORKBOOK

System Verification and Testing

Open Universiteit
www.ou.nl

Production
Open Universiteit

Cover
Team Visuele communicatie, Open Universiteit

Printing and binding
Canon Business Service

First edition 2015
Second edition: 2017
Third edition: 2018

IM0402_50195_27062018

ISBN 978 94 92739 19 3 (workbook)

Course code IM0402

Structure of the course System Verification and Testing

Part	Block	Learning unit	Material Ch*, Sec**	Workload (hours)	Page
Workbook		1 Introduction to the course			7
		2 Introduction to System Verification and Testing	Ch1, sec1	2	11
	1 Formal modelling	3 Kripke Structures	Ch2	4	21
		4 Timed automata	Ch17	6	33
	2 Formal specification	5 Temporal logics	Ch3	6	55
		6 Explicit model-checking	Ch4.1	3	71
		7 Symbolic model-checking	Ch5, Ch6	6	83
	3 Formal testing	8 Model-based testing	Sec2	1	105
		9 Labelled transition systems	Sec4	4	113
		10 The ioco implementation relation	Sec4	6	123
		11 Test generation and execution	Sec5	6	137
Reader	http://youlearn.ou.nl				
Textbook	*Model Checking,* E.M. Clarke, O. Grumberg and D.A. Peled The MIT press, 1999				
Course site	http://youlearn.ou.nl				

* ChX refers to a chapter of the textbook *Model Checking* by Clarke, Grumberg, Peled

** SecX refers to a Section of "Model-based testing with labelled transition systems" by Tretmans

Introduction to the course

Introduction to the course

Before you start with studying this course, this introduction informs you about the objectives and learning goals of the course. A detailed overview of the content is presented in the next learning unit.

1 Positioning of this course

System Verification and Testing (SVT) is part of the Master Software Engineering and the Master Computer Science. The theme of the course is *software quality assurance*. The main topic of SVT is the use of *formal methods* to support the development of high-quality software. *Formal models* play a central role in SVT. SVT covers theoretical and mathematical aspects of ensuring software quality with formal methods. The practical application of the techniques presented in this course is exercised in a project assignment.

1.1 PREREQUISITES

To be able to successfully study this course you need to have a good understanding of propositional and predicate logics, as well as basic notions of set theory. Before starting this course you should be able to:
– understand formulas in propositional- and predicate logics
– write formulas in propositional- and predicate logics
– understand basic concepts of set theory
– write proofs in set theory

These notions are studied in the course *Logica en informatica*. We advise you to take this course before starting with Software Verification and Validation. Additionally, basic notions in algorithmics – like breadth-first search and depth-first search – will be helpful.

1.2 LEARNING GOALS

Learning goals describe which knowledge, insight, and capabilities you should acquire with this course. In every learning unit, you find a list of specific learning goals. The global learning goals of this course are:

LEARNING GOALS
After having studied this course you should be able to:
– write formal specifications using temporal logics
– create formal models of software systems
– recognize in which situations model-checking is applicable
– use a model-checker to prove that models satisfy their formal specifications
– understand the basic principles of model-based testing
– use a model-based testing tool to test whether an implementation conforms to its specification.

2 Course material

The course material consists of the following:
- The textbook *Edmund M. Clarke, Orna Grumberg, Doron A. Peled: "Model checking". MIT Press 1999, ISBN 978-0-262-03270-4, pp. I-XIV, 1-314* .
- The course website in yOUlearn.
- A reader available digitally on the course website containing several articles.
- The article "Model-based testing with labelled transition systems" by Tretmans in the reader.
- This workbook.

The textbook and Tretmans' article are the main texts used in the course. The workbook tells you how to study these texts and points to additional material found in the reader. On the website you will find pointers to tools used in the course, the project assignment, and organisational information.

3 How to read the workbook

This workbook is structured using *learning units*. Each learning unit consists of an introduction, a kernel, and general questions and exercises. The introduction gives an overview of the content of a learning unit. It then proceeds with the following parts:
1. Learning goals: these goals are the knowledge, insight, and capabilities that you should possess after studying the learning unit.
2. Reading assignments: this points to the part(s) of the material that is covered in the learning unit. It refers to chapters of the textbook, sections or complete articles from the reader.
3. Exercises: this part tells you how to proceed with the exercises at the end of the learning unit, if any.
4. Time estimate: finally, an estimate of the time needed to study the learning unit and complete the exercises is given.

The kernel of a learning unit guides you through the course material. It has two components:
1. Reading assignments: A reading assignment looks as follows:
   Reading — Read Section 5.1 of Chapter 5 of the book "Model Checking".

   Studying the workbook will be a lot of reading and solving exercises. You can expect a lot of scary looking theory from the textbook. When the workbook asks you to read a part of the textbook, first read this part entirely to get a global impression of the content. The workbook will clarify the essential points and point to the important formulas of which a deeper understanding is necessary. You should master the content of the textbook just enough to solve the exercises of this workbook. A superficial understanding will not suffice but an in-depth understanding is not necessary. The explanations in the workbook and the exercises are meant to make you focus on the important topics.

2 Questions: while reading the kernel you will often encounter questions. These questions should help you to focus on important points or to check your understanding. You should try to answer the questions by yourself before looking at the solution given at the end of the learning unit. These questions are meant to be short and easy. If you do not see a solution after a few minutes, you should look at the solution. Sometimes, the questions are all given at the end of the learning unit. In that case, you should read the learning unit first and then work on the questions.

All the non-introductory learning units end with one or more exercises. These exercises are meant to illustrate the important notions introduced in the learning unit. They constitute a good way to check whether you have reached the learning goals of the learning unit. A complete solution to every exercise is given at the end of the learning unit.

4 Grading

The final grade is based on the project assignment (1/3) and a written exam (2/3). Sample exams can be found on YouLearn.

5 YouLearn

The course website in yOUlearn contains important information about face-to-face and on-line meetings, the project assignment, grading, and the tools used in the course.

Introduction to System Verification and Testing

Learning unit 1

Introduction to System Verification and Testing

INTRODUCTION

This introductory learning unit presents the content and organisation of the course. It explains the relation between the different formalisms and techniques that you will study in this course. It also relates the content of the course to the design process.

LEARNING GOALS
After having studied this learning unit you should be able to:
- understand the notions of validation and verification as defined in this course
- understand the global structure of the course
- understand the purpose of the different formalisms and techniques for formal validation and verification.

Reading

Chapter 1 of the textbook "Model Checking" and Section 1 of the paper “Model-based testing with labelled transition systems” belong to this learning unit.

Exercises

This chapter ends with two short on-line exercises. The last section explains how to proceed with them.

Time

The expected time needed to study this learning unit is about 2 hours.

1 Validation, verification, and testing

Figure 1.1 shows an overview of concepts of this course. For any system, there is an intention of what that system should do, i.e., how it should behave. This intention consists of *informal* ideas. In this course, we use logic to formalize such ideas into a *formal specification*. Such a specification is written in a language whose semantics are mathematically defined, and is therefore unambiguous. Assessing whether the formal specification correctly captures the intended behavior is *validation*.

Subsequently, we build a *formal model* of the system. We again use mathematical constructs, such as state machines (also state automata or finite state machines), instead of more informal approaches such as UML. As a result, the semantics of our model are again unambiguously defined. It is therefore possible to assess whether the formal model behaves according to the formal specification. That assessement is called *formal verification*. We will learn a specific technique of formal verification called *model checking*, which is an automated technique for assessing whether a formal model adheres to a formal specification.

The formal specification and model live in a "mathematical world". It is often said that formal verification is as good as the model, by which it is meant that if one makes an error while modelling, the whole assessment becomes futile. It must therefore also be assessed whether the model is good, i.e., whether the model is an accurate description of the actual realization of the system. That assessment is called *testing*. In this course, you will learn *model-based testing* where test suites are generated automatically from the model.

Ultimately, you can have a realization of a system that is *tested* to be conform to a formal model, a formal model that is *verified* to behave according to a formal specification, and a formal specification that is *validated* to accurately reflect the informal ideas of how the system should behave.

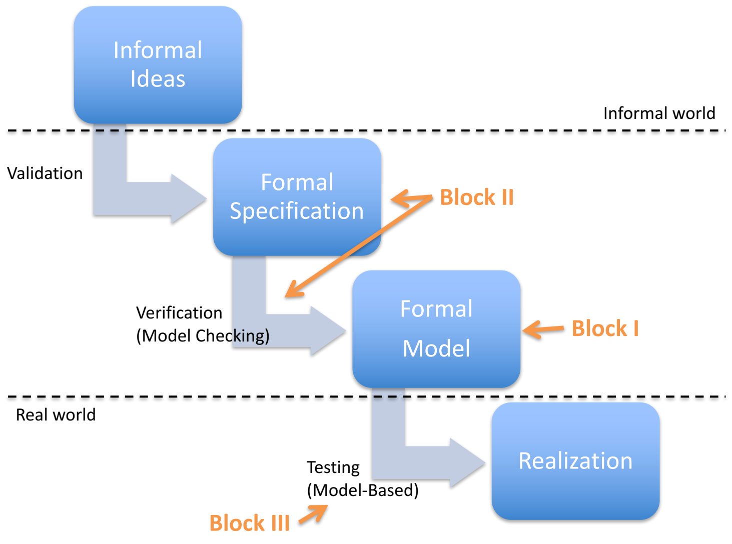


FIGURE 1.1 SVT overview

2 The structure of this course

This course is structured in three blocks:
– Block 1: Formal modelling
– Block 2: Model-checking
– Block 3: Model-based testing

The course considers various formalisms to express formal models. The commonality between these formalisms is that they all constitute some variation of automata. An automaton is a graph where vertices denote states of the system and edges denote transitions between states. Sometimes edges are labelled with an action that is performed to move from one state to another.

Block 1

The first block has two learning units. The first discusses Kripke structures, which are the most fundamental way of modelling a system. It will be shown that any system, whether it concerns software or hardware, can be modelled as a Kripke structure. The second learning unit discusses timed automata, which allow efficient modelling of concepts such as time or parallelism. The first part of the project assignment concerns modelling a system using timed automata.
- Kripke stuctures (Learning Unit 2)
- Timed automata (Learning Unit 3)

Block 2

The second block has two parts. It first introduces temporal logics, that can be used to write formal specifications. It then presents two kinds of model-checking algorithms. For the second part of the project assignment, you will formalize an informal specification to temporal logic and then verify that your model is correct.
- Temporal logics (Learning Unit 4)
- CTL explicit-state model-checking (Learning Unit 5)
- CTL symbolic model-checking (Learning Unit 6)

Block 3

The last block is about using model-based testing techniques. It introduces a theory for model-based testing based on the **ioco** conformance relation and how to automatically derive and execute test cases in this theory. The **ioco** relation is based on a different kind of formalism called labelled transition systems, which are introduced first.
- Model-based testing principles (Learning Unit 7)
- Labelled transition systems (Learning Unit 8)
- The **ioco** conformance relation (Learning Unit 9)
- Generating test cases for **ioco** (Learning Unit 10)

3 Verification and testing of a small example

We now illustrate the topics covered in this course using a small example (see Figure 1.2). We consider a simple microwave oven with a door, a start button, and a lamp. The door can be closed or open. Users push the start button to cook for 10 seconds. The lamp turns red when the oven is cooking. We focus on the control software running on the micro-controller inside the oven. This micro-controller is also pictured in Figure 1.2. It is connected to five wires. It receives two signals from sensors giving information about the status of the door and the oven. A door sensor signal set to 1 means that the door is open. This signal is 0 when the door is closed. The oven sensor signal is 1 when the oven is cooking and 0 otherwise. The micro-controller also receives a pulse when the button is pushed. When the button is pushed, the button signal is set to 1 for 10 ms before coming back to 0. The micro-controller has two command signals. A 1 is sent to the lamp or the oven to turn these devices on. When the command is put back to 0, these devices are turned off. The oven must satisfy the safety property that it is never cooking while the door is open. This completes an informal description of the oven and its expected behaviour.

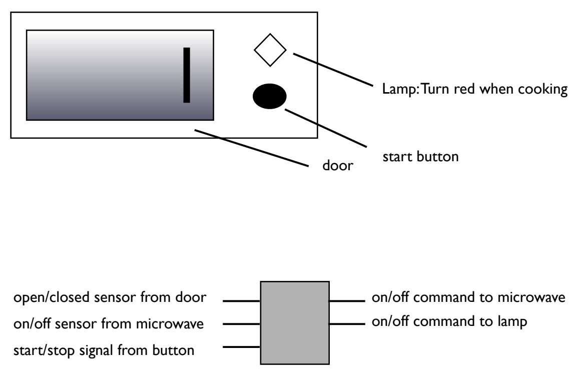

Lamp:Turn red when cooking
start button
door
open/closed sensor from door
on/off sensor from microwave
start/stop signal from button
on/off command to microwave
on/off command to lamp

FIGURE 1.2 A simple microwave-oven and its controller

3.1 FORMAL SPECIFICATION

Formal specification

A *formal specification* consists of a set of properties capturing the intended behaviour of the system. Properties may be expressed in temporal logics. These logics are a combination of propositional and predicate logics with modal operators. These logics enable the expression of properties about the occurrence of events over time. Temporal properties are used to reason about *execution paths* of systems. An execution path is an infinite sequence of states representing the behaviour of the system. For instance, the property that the oven is never cooking while the door is open is stated as “for any execution path, at any time, if the oven is cooking then the door is closed”. The fact that something is true for all executions is specified using the quantifier “for all”, denoted **A**. The fact that something is true at any time during an execution is expressed by the temporal operator “globally”, denoted **G**. The formula specifying this requirement will have the form "**AG** *if the oven is cooking then the door is closed*". To formally express this fact, we need two state variables keeping track of the status of the oven and the status of the door. We name these variables `IsCooking` and `IsOpen`. Variable `IsCooking` is true when the oven sensor signal is 1. Variable `IsOpen` is true when the door sensor signal is 1. Now, we can complete our formula as follows:

$$\mathbf{AG}\ (\texttt{IsCooking} \implies \neg\texttt{IsOpen}) \tag{1.1}$$

3.2 FORMAL MODEL

We are interested in modelling the expected behaviour of state variables `IsCooking` and `IsOpen`. Such a model is given in Figure 1.3. This model is a kind of state machine specifying the possible transitions between the different values of these two variables. Such a kind of model is called a *Kripke structure* and it is discussed in detail in Learning Unit 2. Initially, the door is opened and the oven is not cooking. Once the door has been closed, either the oven starts cooking or the door is open again.

When the oven is cooking, the oven may stop and keep its door closed or stop with an open door. It is clear that the state where the oven is cooking with an open door is not reachable as no transitions lead to it. Hence Property 1.1 holds.

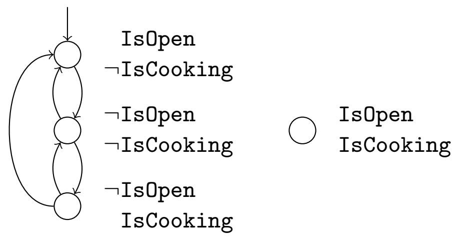


FIGURE 1.3 Modelling the microwave-oven as a Kripke structure for model-checking

3.3 FORMAL VERIFICATION

Kripke structures focus on the state variables of a design and abstract away from the inputs. The objective is to show that the design never reaches a state where the door is open and the oven is cooking. This has been formally expressed in temporal logic by Property 1.1. The technique used to prove that a model satisfies this kind of property is called model-checking and is presented in details in Learning Units 5 and 6. The idea behind model-checking is to exhaustively explore the set of reachable states and check that a property holds for all these states. In the model discussed above, the set of reachable states does not include the state where `IsOpen` and `IsCooking` are true. Therefore, Property 1.1 holds for all states *and for all input sequences*.

Reading

Read Chapter 1 of the textbook "Model Checking".

3.4 MODEL-BASED TESTING

So far, we have discussed a formal model and a formal specification. As defined earlier, the goal of testing is to check that the final realization is conforming to the formal model. In this course, the final realization is a black-box. It is impossible to gain access to its internal state. The only possible interaction happens at the interface, that is, on the pins of the micro-controller. The notions of inputs and outputs are now the point of focus.

Models for testing.
Models for testing are based on these notions of inputs and outputs. Figure 1.4 shows such a model for our oven example. As a convention, inputs are prefixed with a '?' and outputs with a '!'. The oven controller has three inputs corresponding to the actions of opening or closing the door and starting the oven. The controller has two outputs modelling the actions of cooking or stopping. We can relate these actions to the state variables as follows. An input action "?*open*" will set the door sensor and variable `IsOpen` to 1. An input action "?*close*" will do the

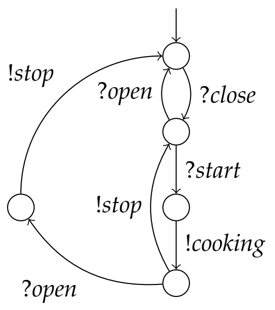


FIGURE 1.4 Modelling the microwave-oven as a labelled transition system for testing

opposite and set the sensor and the variable to 0. Similarly, output action "!*cooking*" corresponds to observing that the oven is cooking, that is, "observing" that variable `IsCooking` is 1. An output action "!*stop*" corresponds to observing a transition of `IsCooking` from 1 to 0. The kind of model used in this context is a *labelled transition system* (LTS) and is discussed in detail in Learning Unit 8. These transition systems have abstract states containing no information. The labels on the transition are the input and output actions. The model in Figure 1.4 shows that after closing the door the oven reaches a state where the start button can be pushed. After pushing that button, the oven reaches a state where cooking can start. If someone opens the door while the oven is cooking, the oven stops and get back to its initial state.

From this model, a test suite can automatically be generated and tests can be executed. As a result, it can be shown that the realization is conform to the model.

Reading

Read Section 1 of "Model-based testing with labelled transition systems" by Tretmans.

3.5 REAL-TIME SYSTEMS

All models discussed so far lack a concrete notion of time. Timed automata extend automata with a *continuous* notion of time. Time is represented by clock variables. These variables are real-valued and they all grow at the same rate. Timed automata are used to represent real-time systems. Figure 1.5 shows a timed automaton modelling the simple oven example. Variable x is a clock variable. It is used in *clock constraints*, for instance $x \leq 10$, $x == 10$ or $x \leq 1$. Clock variables can be reset to 0, for instance $x := 0$. In a timed automaton, time only progresses in states, called locations. A transition happens in zero time. The constraints are used to restrict the amount of time that can elapse in a state. In the example, once the start button has been pushed the oven should start cooking after at most 1 time unit. When it starts cooking it will stop after exactly 10 time units. The constraint on the !*stop* transition is called a (clock) guard. This transition is only possible when clock x is exactly 10. If the door is opened while cooking, the oven should stop at the latest 1 time unit after opening the door. Note that a

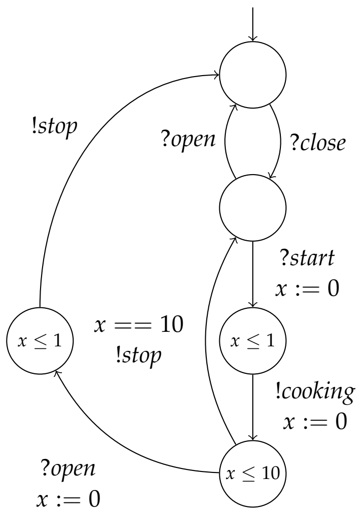


FIGURE 1.5 A timed automaton for the oven example

time unit is arbitrary here. It typically is a second or millisecond. Timed automata are introduced in Learning Unit 3 together with their associated validation techniques. Timed automata are also used in the project assignment.

Block 1

Formal modelling

Kripke Structures

Learning unit 2

Kripke Structures

INTRODUCTION

This learning unit introduces Kripke structures and their use to model computing systems. Kripke structures constitute a basic formalism used in model-checking. Model-checking algorithms for Kripke structures are introduced in the second block of this course.

LEARNING GOALS
After having studied this learning unit you should be able to:
– understand the basic intuition behind Kripke structures
– understand the formal definition of Kripke structures
– create models using Kripke structures

Studeeraanwijzingen

Reading

Chapter 2 of the book "Model Checking" belongs to this learning unit. The paper from the reader "What is a good model?" by F. Vaandrager also belongs to this learning unit.

Exercises

This chapter finishes with two exercises. The complete solution of these two exercises is also given.

Time

The expected time needed to study this learning unit is about 4 hours.

KERNEL

1 Introduction

Reading

Read the introduction to Chapter 2 of the textbook "Model Checking" on page 13.

All models considered in this course are variations of transition systems. In the textbook "Model Checking", the authors consider Kripke structures. We already sketched a Kripke model of the oven example in Learning Unit 1. An important assumption in model-checking, and hence also in this course ,is that systems are reactive. As briefly described in the textbook, reactive systems never stop interacting with their environment. Reactive systems keep taking transitions from one state to another. In that sense, they never terminate and execute indefinitely. Indeed, execution paths of Kripke structures formally are infinite sequences of states.

2 Modelling concurrent systems

Reading

Read the introduction of section 2.1 on page 14 of the textbook "Model Checking".

This paragraph gives a formal definition of Kripke structures. This definition has four elements:

1. a finite set of states;
2. a set of initial states, that is, the states in which the system can be when it is turned on;
3. a transition relation that gives for every state the possible successor states. There may be more than one successor state. The transition relation is total, which means that all states have a successor;
4. a labelling function that tells which propositions are true in a state. This is the central aspect of Kripke structures: information is stored in states.

To create a Kripke model for a system means to specify these four elements.

We now return to the oven example discussed in Learning Unit 1.

QUESTION 2.1
Recall the Kripke structure representing the oven example than was pictured in Figure 1.3 in Learning Unit 1.
a Give a formal definition of this Kripke structure.
b Give three examples of paths of this Kripke structure.

2.1 FIRST ORDER REPRESENTATION

Defining all components of the definition of a Kripke structure is a tedious task. First order logic provides a more compact way to represent a Kripke structure. In particular, it introduces the "prime" notation to encode transitions. For instance, the transition of Boolean variable a from true to false is encoded as $a \wedge \neg a'$. Here, a denotes the value before the transition and a' denotes the value after the transition.

Note that the notation for the first order presentation of a Kripke differs from the notation used to represent the Kripke structure itself. In general, caligraphic letters are used for the first order representation. For instance, $\mathcal{S}$ represents the first order representation of the set of states S and $\mathcal{R}$ represents the first order representation of the transition relation R.

Let us take a look at a simple example. Consider the program statements below:

```
0: x = x + y
1: y = x - y
2: x = x - y
```

These three lines are executed over and over again. The initial values of x and y are 3 and 5. This is represented by the following formula defining the set of initial states:

$$\mathcal{S}_0 \equiv x = 3 \land y = 5$$

Each line of the program is represented by a transition. Using the prime notation the three transitions can be represented as follows:

$$\begin{aligned}\mathcal{R}(x,y,x',y') \equiv\ & (x' = x + y \land y' = y) \lor (x' = x \land y' = x - y) \\ & \lor (x' = x - y \land y' = y)\end{aligned}$$

We assume a synchronous semantics, that is, the sequential execution of the three lines of our program example. This synchronous execution needs to be expressed in the Kripke model. This is achieved by introducing an extra variable pc as a "program counter". Variable pc takes values in $\{0, 1, 2\}$ and it points to the line that should be executed. In every transition, variable pc is incremented cyclically, that is, $pc' = (pc + 1) \bmod 3$. At the end, the complete transitionrelation is:

$$\begin{aligned}\mathcal{R}(x,y,pc,x',y',pc') \equiv\ & (x' = x + y \land y' = y \land pc = 0 \land pc' = (pc+1) \bmod 3) \\ & \lor (x' = x \land y' = x - y \land pc = 1 \land pc' = (pc+1) \bmod 3) \\ & \lor (x' = x - y \land y' = y \land pc = 2 \land pc' = (pc+1) \bmod 3)\end{aligned}$$

The state graph of the Kripke structure is given in Figure 2.1. States are triples representing the values of x, y, and pc. Note that we only show the states that are reachable from the initial state.

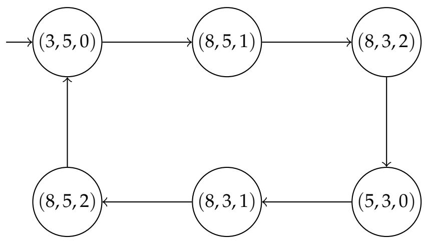


FIGURE 2.1 State graph for the Kripke structure of our example

As expected the program simply exchanges the values of x and y.

From the first order encoding, one can extract the Kripke structure defined as follows:
- $D = \{3, 5, 8\}$ and $S = D \times D \times \{0, 1, 2\}$
- $S_0 = \{(3, 5, 0)\}$
- $R = \{((3,5,0),(8,5,1)), ((8,5,1),(8,3,2)), ((8,3,2),(5,3,0)),$
  $((5,3,0),(8,3,1)), ((8,3,1),(8,5,2)), ((8,5,2),(3,5,0))\}$
- $L((i, j, k)) = \{x = i, y = j, pc = k\}$

Reading

Read section 2.1.1 of Chapter 2 on pages 14 to 16 of the textbook "Model Checking".

A valuation simply is a function assigning concrete values to variables. The states of a system are all possible valuations for all the variables. For instance, a system with a 64-bit register has 2^{64} possible states. Transitions also are encoded using first order logic using the prime notation.

As we sketched earlier, this encoding uses a convention where v denotes the current value of variable – that is, before a transition – and v' denotes the next value of a variable – that is, the value after a transition. A function incrementing a 64-bit register would be defined as $reg' = (reg + 1) \bmod 2^{64}$.

QUESTION 2.2
Give the first order representation of the Kripke structure of the oven example given as answer to the previous question.

2.2 GRANULARITY OF TRANSITIONS

Reading

Read section 2.1.2 of the textbook "Model Checking" on pages 16 to 17.

This section raises a very important issue about formal verification. Model checking is as good as the model! A model is always an abstraction of the real system. A model is a sound abstraction if any property that holds on the model always holds on the real system. A model is complete if any property satisfied by the real system is also satisfied by the model.

2.3 WHAT IS A GOOD MODEL?

Reading

Read the article from the reader titled "What is a good model?" by Frits Vaandrager. Note that the reader can be found in yOUlearn.

3 Concurrent systems

Reading

Read sections 2.2 and 2.3 of the textbook "Model Checking" on pages 17 to 26.

This section describes different relations between actual systems and their representations as Kripke structures. This section focuses only on communications via shared variables. In the project assignment, more communication patterns will be studied. The two modes of execution presented in this section are important. The interleaving or asynchronous mode represents the parallel – also named concurrent – execution of processes. It considers all possible ways of combining sequential executions. The synchronous mode represents the execution on a single machine. All processes execute at the same time.

Remark

It is important to observe that the translation of sequential programs to Kripke structures in section 2.2.2 of the book results in a transition relation which is not total. At the end of the translation, when $pc = m'$, there are no outgoing transitions. This can be repaired by adding a disjunction $pc = m' \wedge pc' = m' \wedge same(V)$ to the translation of the whole program. This change essentially closes the final states of the program with a self-loop.

QUESTION 2.3
Consider the following program fragment, where div denotes integer division. In the initial state, $x = 3452, z = 0$.

```
while x != 0 do
  x := x div 10;
  z := z + 1
end while
```

a Consider the program fragment as a statement P. Give the labelled statement $P^{\mathcal{L}}$ that is obtained by applying the labelling in section 2.2.2 of the textbook.

b Provide the Kripke structure that is the result of applying the translation to the given program code.

Observe that the state graph of the Kripke structure already becomes large, even for such a simple program! In order to obtain a (slightly) more concise representation of the state graph, we sometimes take the liberty to combine a number of states that appear in sequence. For instance, in the solution to the previous question, we could combine states that have the x and z values, but program counter values m and l_1. For example, states $x = 3452 \wedge z = 0 \wedge pc = m$ and $x = 3452 \wedge z = 0 \wedge pc = l_1$ could be combined.

QUESTION 2.4

Argue that the model of the program in section 2.3 of the textbook is a good model.

4 Conclusion

This learning unit gave a formal definition to Kripke structures and illustrated their use to model hardware and software systems. In the second block of this course, the formal specification of temporal properties of Kripke structures and the algorithmic validation of these properties are presented.

EXERCISES

1 Peterson's algorithm is a concurrent program allowing two processes to access a single-use resource. The two processes communicate via two shared variables. The first variable is an array containing two flags. The second variable is either 0 or 1 and is not initialised. Initially, we have `flag[0] = flag[1] = 0`. Each process executes the following code:

Considering an interleaving semantics – that is process P_0 can execute before or after process P_1 and process P_0 can execute many times in a row before process P_1 executes – build a Kripke model representing the evolution of the state variables `flag` and `turn`.

Argue why it is a good model.

```
P_0::  l_0  flag[0] = 1;
       l_1  turn = 1;
       l_2  while (flag[1] == 1 && turn == 1) {
            ;; process P_0 waits
            }
            ;; process P_0 enters critical section
       l_3  ...
            ;; process P_0 leaves critical section
       l_4  flag[0] = 0;

P_1::  l_0  flag[1] = 1;
       l_1  turn = 0;
       l_2  while (flag[0] == 1 && turn == 0) {
            ;; process P_1 waits
            }
            ;; process P_1 enters critical section
       l_3  ...
            ;; process P_1 leaves critical section
       l_4  flag[1] = 0;
```

2 A goat, a cabbage, and a wolf are on the left side of a river. A sailor must use a small boat to transport them to the right side of the river. The boat can only transport one passenger at a time (in addition to the sailor of course). The following rules apply. If the goat and the wolf are left alone on one side of the river, the wolf eats the goat. If the goat and the cabbage are left alone on one side of the river, the goat eats the cabbage. The sailor can load the boat, unload the boat, and cross the river.
Build a Kripke model of the puzzle. Your model should help you answer the question: how can the sailor safely transport every passenger? Argue that your model is a good one.
Hint: you can consider states where the cabbage or the goat is eaten as an "error" state. An error state is a *sink* state that has no outgoing transitions and only a self-loop to itself.

FEEDBACK

Answers to questions

2.1 a Formally defining the Kripke structure means giving definitions for the different components of the definition. There are two variables, therefore the set of atomic propositions is $AP = \{\texttt{IsCooking}, \texttt{IsOpen}\}$.

- Each proposition is a Boolean. It can be true (1) or false (0). The set of states is the set of all possible pairs of Booleans, that is, $S = \mathbb{B}^2$. The first element of the pair represents variable `IsOpen` and the second element of the pair variable `IsCooking`.
- The initial state is when the door is open and the oven is not cooking, so $S_0 = \{(1,0)\}$.
- The transition relation contains all pairs of states between which there is a transition in the Kripke structure, so

$$R = \{((1,0),(0,0)),((0,0),(1,0)),((0,0),(0,1)),((0,1),(0,0)),\\ ((0,1),(1,0))\}.$$

- The labelling function must be defined for each state. This function returns the set of atomic propositions that are true in a state. For the oven example, we have the following:
  - $L((0,0)) = \emptyset$
  - $L((0,1)) = \{\texttt{IsCooking}\}$
  - $L((1,0)) = \{\texttt{IsOpen}\}$
  - $L((1,1)) = \{\texttt{IsOpen}, \texttt{IsCooking}\}$

b Examples of paths in this Kripke structure are:

- $(1,0)(0,0)(0,1)(1,0)(0,0)(0,1)(1,0)\ldots$
- $(1,0)(0,0)(1,0)(0,0)(1,0)\ldots$
- $(1,0)(0,0)(0,1)(0,0)(0,1)(1,0)(0,0)(0,1)(0,0)(0,1)(1,0)\ldots$

Here, with $\ldots$ we indicate that the preceding sequence is repeated. Note that there are infinitely many correct paths that can be given here. All such paths start in state $(1,0)$ (the initial state), and only follow transitions in the Kripke structure. Also, all paths are infinite (by definition of path).

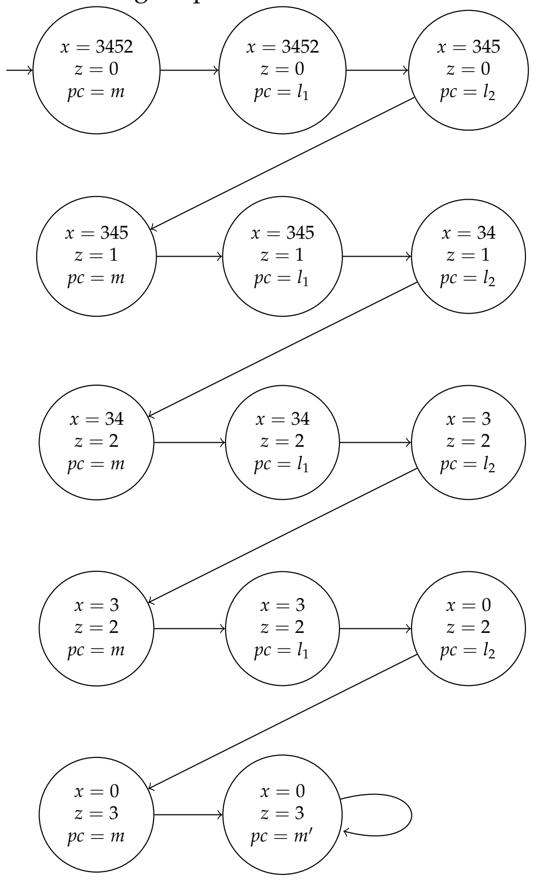
2.2 The set of atomic propositions is unchanged, $AP = \{\texttt{IsCooking}, \texttt{IsOpen}\}$. There is one initial state encoded as $\mathcal{S}_0 = \texttt{IsOpen} \wedge \neg\texttt{IsCooking}$. The transition relation, noted $\mathcal{R}(\texttt{IsOpen}, \texttt{IsCooking}, \texttt{IsOpen}', \texttt{IsCooking}')$, is encoded as the following disjunction:

$$\begin{aligned} & \texttt{IsOpen} \wedge \neg\texttt{IsCooking} \wedge \neg\texttt{IsOpen}' \wedge \neg\texttt{IsCooking}' \\ \vee\ & \neg\texttt{IsOpen} \wedge \neg\texttt{IsCooking} \wedge \texttt{IsOpen}' \wedge \neg\texttt{IsCooking}' \\ \vee\ & \neg\texttt{IsOpen} \wedge \neg\texttt{IsCooking} \wedge \neg\texttt{IsOpen}' \wedge \texttt{IsCooking}' \\ \vee\ & \neg\texttt{IsOpen} \wedge \texttt{IsCooking} \wedge \neg\texttt{IsOpen}' \wedge \neg\texttt{IsCooking}' \\ \vee\ & \neg\texttt{IsOpen} \wedge \texttt{IsCooking} \wedge \texttt{IsOpen}' \wedge \neg\texttt{IsCooking}' \end{aligned}$$

2.3 (a) We obtain the following labelling

```
m:  while x != 0 do
l1:   x := x div 10;
l2:   z := z + 1
    end while
m':
```

(b) The resulting Kripke structure is as follows.

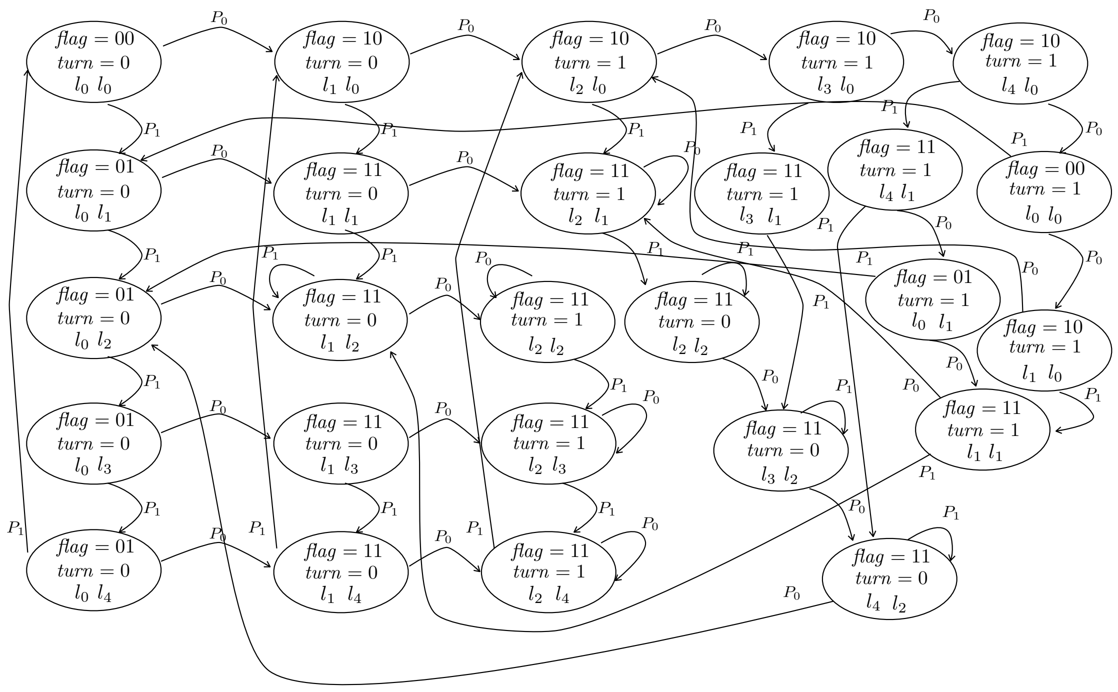

$x = 3452$ $z = 0$ $pc = m$
$x = 3452$ $z = 0$ $pc = l_1$
$x = 345$ $z = 0$ $pc = l_2$
$x = 345$ $z = 1$ $pc = m$
$x = 345$ $z = 1$ $pc = l_1$
$x = 34$ $z = 1$ $pc = l_2$
$x = 34$ $z = 2$ $pc = m$
$x = 34$ $z = 2$ $pc = l_1$
$x = 3$ $z = 2$ $pc = l_2$
$x = 3$ $z = 2$ $pc = m$
$x = 3$ $z = 2$ $pc = l_1$
$x = 0$ $z = 2$ $pc = l_2$
$x = 0$ $z = 3$ $pc = m$
$x = 0$ $z = 3$ $pc = m'$

2.4 We look at the seven criteria for a good model and comment about how the model on page 26 satisfies them:

a Object of modelling: the object is the program given on page 25.
b Purpose: the purpose of the model is mentioned at the end of page 26 and is to prove the safety property of mutual exclusion.
c Traceability: the description below the program precisely relates the different variables of the model to the code of the program.
d Truthfulness: this is a really important property of a model. A model that would satisfies a property unsatisfied by the object would be useless. The model of the program satisfies the mutual exclusion property. We must now argue that the program also satisfies this property. Traceability of the model is really good, as it directly relates to the program. Therefore we can conclude with a high confidence that the program also satisfies mutual exclusion.
e Simplicity: The object is already simple. The model is also simple.
f Extensibility and re-usability: the model is very specific and cannot easily be extended. For instance, a better model would have as parameter the number of processes.
g Interoperability and sharing: this criterion is very specific to the context of embedded systems where several models of the same object exist. It is not really applicable to our example, where one model is enough.

Answers to exercises

1 The solution is similar to the example on pages 25 and 26 of the textbook. We consider the pair of the program counters and the turn and flag variables. A possible solution is given in Figure 2.2.



$flag = 00$ $turn = 0$ l_0 l_0
$flag = 10$ $turn = 0$ l_1 l_0
$flag = 10$ $turn = 1$ l_2 l_0
$flag = 10$ $turn = 1$ l_3 l_0
$flag = 10$ $turn = 1$ l_4 l_0
$flag = 01$ $turn = 0$ l_0 l_1
$flag = 11$ $turn = 0$ l_1 l_1
$flag = 11$ $turn = 1$ l_2 l_1
$flag = 11$ $turn = 1$ l_3 l_1
$flag = 11$ $turn = 1$ l_4 l_1
$flag = 00$ $turn = 1$ l_0 l_0
$flag = 01$ $turn = 0$ l_0 l_2
$flag = 11$ $turn = 0$ l_1 l_2
$flag = 11$ $turn = 1$ l_2 l_2
$flag = 11$ $turn = 0$ l_2 l_2
$flag = 01$ $turn = 1$ l_0 l_1
$flag = 10$ $turn = 1$ l_1 l_0
$flag = 01$ $turn = 0$ l_0 l_3
$flag = 11$ $turn = 0$ l_1 l_3
$flag = 11$ $turn = 1$ l_2 l_3
$flag = 11$ $turn = 0$ l_3 l_2
$flag = 11$ $turn = 1$ l_1 l_1
$flag = 01$ $turn = 0$ l_0 l_4
$flag = 11$ $turn = 0$ l_1 l_4
$flag = 11$ $turn = 1$ l_2 l_4
$flag = 11$ $turn = 0$ l_4 l_2
P_0
P_1

FIGURE 2.2 A Kripke structure representing Peterson's algorithm

We look at the seven criteria for a good model and comment about how this satisfies them:

a Object of modelling: the object is given in the question and is Peterson's mutual exclusion algorithm

b Purpose: the purpose of the model is to prove the safety property of mutual exclusion.

c Traceability: the model is a direct encoding of the program counter and variables. It is easy to relate the model to the program.

d Truthfulness: Because of the strong traceability and because no abstractions were made in the model, it is very likely that the program satisfies mutual exclusion.

e Simplicity: The model involves a small number of variables but the state graph, even for this small program, is quite large. This state-explosion-problem is a major obstacle to the application of model-checking to large systems.

f Extensibility and re-usability: the model is very specific and cannot easily be extended. For instance, a better model would have as parameter the number of processes and the number of flags or turn values.

g Interoperability and sharing: this criterion is very specific to the context of embedded systems where several models of the same object exist. It is not really applicable in this exercise, where one model is enough.

2 We define three main variables: wolf, goat, and cabbage. The value of each variable represents the state of the goat, the wolf, or the cabbage. A state is either left, boat, or right indicating the position or dead meaning that the goat or the cabbage have been eaten. In the Kripke structure we simply write *left*, *left*, *left* to mean that the wolf, the goat, and the cabbage are "alive" on the left side of the river. The graph is given in Figure 2.3. Note that the Figure omits the self-loop transitions on the error states.

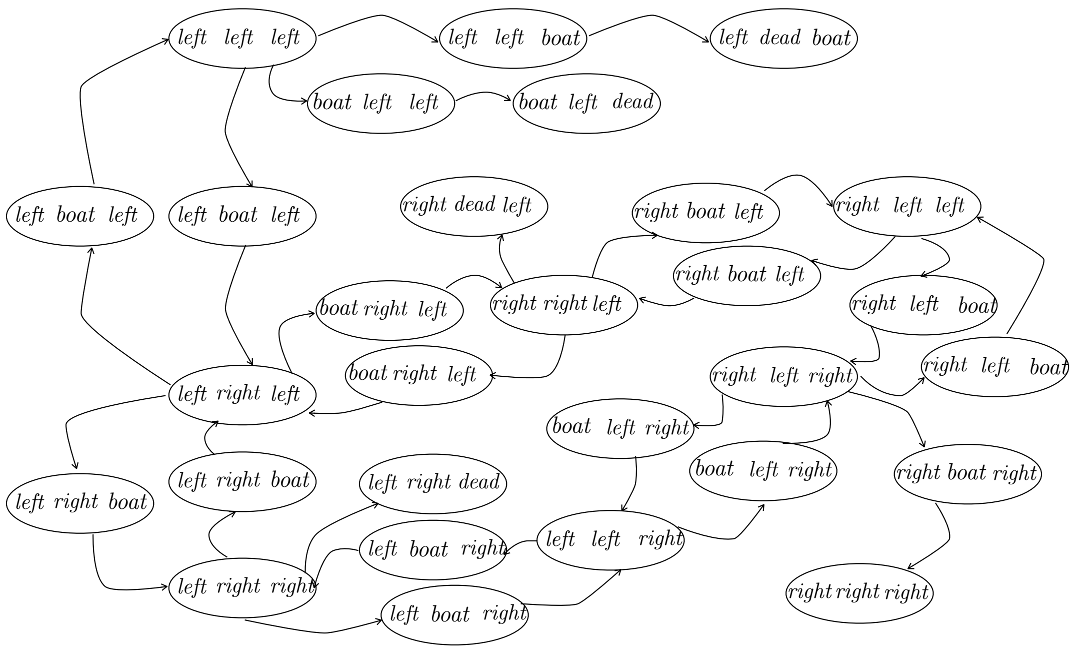


FIGURE 2.3 A Kripke structure representing a solution to the Wolf Goat Cabbage puzzle.

We look at the seven criteria for a good model and comment about how this satisfies them:

a Object of modelling: the object of modelling is the Wolf, Goat, Cabbage puzzle.

b Purpose: the purpose of the model is to find a solution to the puzzle. This is also given in the question.

c Traceability: Participants are only represented by their position. This is the only relevant information. The goat and the cabbage can have an additional state representing the fact that they have been eaten. A state with a dead participant is considered a final state with no outgoing transitions.

d Truthfulness: The goal is to find a solution and the solution found using the model can always be checked on the real problem.

e Simplicity: The model assumes that once a participant is on the boat it will cross the river. For instance, the state "left boat right" is duplicated to separate the event of traversing from left to right or right to left. Merging these two states mean that it is possible to change direction while crossing. This solution would reduce the state space and in that sense make the model smaller. On the other hand, it introduces infinite behaviours where the sailor could keep changing direction in the middle of the river. Both models give the same solution to the puzzle. The model presented here as a solution is cleaner than its smaller alternative.

f Extensibility and re-usability: the model is very specific and cannot easily be extended. It would be interesting to general the models with n participants and rules to determine when participants die.

g Interoperability and sharing: Not Applicable.

Timed automata

Learning unit 3

Timed automata

INTRODUCTION

Timed automata provide models with an explicit and continuous notion of time. This corresponds to our intuition about time and is useful to model and reason about real-time systems and asynchronous systems. This learning unit introduces a formal definition of timed automata and their essential properties. Timed automata constitute the basic models of the tool UPPAAL used in the project assignment.

LEARNING GOALS

After having studied this learning unit you should be able to:

- understand the formal definition of timed automata
- create models using timed automata
- understand the basic algorithms for reachability
- understand the notions of clock regions and clock zones
- give the region automaton corresponding to a timed automaton
- understand the encoding of zones using DBMs

Reading Chapter 17 of the textbook "Model Checking" belongs to this learning unit.

Exercises You should work on the exercises at the end of the learning unit.

Time The expected time needed to study this learning unit is about 6 hours.

KERNEL

1 Continuous real-time

Reading Read the introduction of Chapter 17 on page 265 of the textbook "Model Checking".

The fundamental element introduced in this chapter is to consider time as a continuous entity. In this context, time is often represented by the non-negative real numbers. Several years after the publication of the textbook, timed automata are still the standard models to perform model-checking of real-time systems. In this learning unit, we introduce their basic theoretical foundations.

2 Timed automata

Let us first have a look at the example in Figure 17.1 on page 266 of the textbook. This example has two states (s_0 and s_1) and two transitions. States are called locations. One transition is an a-action and the other one is a b-action. Variables x and y are clock variables. A clock variable is a real-valued variable representing time. Clock variables are read-only except for a possible reset to 0 on transitions ($x := 0$ and $y := 0$). The value of a clock variable simply denotes the amount of time that has elapsed since the last reset of this clock. Clocks can be used in formulas, called clock constraints, on transitions and states. On a transition, a clock constraint is called a *guard*. A transition is only possible when the guard is true. For instance, the transition with an a action is only possible if the value of clock y is at least 3 time units. In a location, a clock constraint is called a location *invariant*. Transitions are instantaneous, no time elapses by making a transition. Time can only make progress in a location as long as the invariant is true. For instance, when clock y is greater than 5, location s_0 has to be left, that is, the system must perform action a. Note that to make a transition from a location to another, the invariant of the target location must be true after the transition. For instance, to make a transition from s_1 to s_0 the value of y cannot be greater than 5. If y is greater than 5, as y is not reset on the transition, the invariant of location s_0 does not hold. In such a situation the system would be in a deadlock as it becomes impossible to stay in location s_1 and it is impossible to enter location s_0. In general, we say that a timed automaton is in a deadlock if, in the current location, and given the current value of the clocks, no transition to another state can be taken, and time cannot elapse.

QUESTION 3.1
The example in Figure 17.1 suffers from deadlocks. Can you describe an execution, starting from s_0 with x and y initially equal to 0 that leads to a deadlock situation?

Reading

Read section 17.1 on pages 265 to 268 of Chapter 17 of the textbook "Model Checking".

2.1 SYNTAX

The bottom of page 266 and the top of page 267 introduce the syntax of timed automata. It first defines the restricted set of constraints that may be used in guards and invariants. It then defines the different components of a timed automaton.

The set of possible clock constraints is defined at the bottom of page 266. A clock constraint can either bound a clock from below, for instance $10 \leq x$, or from above, for instance $x \leq 10$. In the definition of the textbook clocks can be bounded by non-negative rational numbers. In practice, non-negative integers are sufficient. From now on, we assume that clock constraints use non-negative integers only. The second point states that only a conjunction of clock constraints is possible. Disjunction of constraints or negation of a constraint are not allowed in clock constraints.

The top of page 267 defines the syntax of a timed automaton. A timed automaton has six elements:

1 A finite set of *actions*, denoted by Σ, and called the "alphabet".
2 A finite set of *locations*, denoted by S.
3 A set S_0 of *initial locations*. Very often there is a single starting location. The set of initial locations must be a subset of the set of locations, that is, $S_0 \subseteq S$.
4 A finite set of *clocks*, denoted by X.
5 A function I that gives the *invariant* of each location. For a location l, then $I(l)$ denotes the invariant of location l.
6 A finite set of *transitions*, denoted by T. In the previous chapter, we saw that in Kripke structures a transition was defined by its start state and its end state. Similarly, for timed automata, a transition is defined by a start location s and a target location s'. However, in timed automata, a transition is accompanied by additional information. First, an action a which is a *synchronization*. Constraint φ is a *guard*. Finally, set λ are the *resets*. Each of these terms is explained in the next sections of this chapter.

2.2 SEMANTICS

The textbook describes the semantics of timed automata, by describing which kind of transitions can occur. Basically, a timed automaton is translated into a kind of Kripke structure where each state s has a clock assignment ν. In the textbook, they refer to this Kripke structure as a TLTS. In the TLTS, two kinds of transitions are possible: delay and action transitions. The first type essentially means "wait a while and do nothing". The second type essentially means "perform an action".

2.2.1 *Time and Invariants*

A delay transition with delay $d \in \mathbb{R}^+$, denoted $(s, \nu) \xrightarrow{d} (s, \nu + d)$, exists provided $I(s)$ holds for any time between ν and $\nu + d$. This means that at any time, it is possible to stay in the current state s and wait a while. If we wait d time units, then time progresses to $\nu + d$ (this increments all clocks with d). Crucial, however, is that *the invariant must always be true*. It is only possible for time to progress in state s, if at all times the invariant $I(s)$ holds. Otherwise, time cannot progress, meaning that the state must be left!

Consider, e.g., state s_0 in Figure 17.1 of the textbook. Initially, clock $x = y = 0$. We can execute a delay transition with, e.g., $d = 5$. This means: stay in s_0 for 5 time units. In the corresponding TLTS, the following transition is possible:

$$(s_0, x == 0 \wedge y == 0) \xrightarrow{5} (s_0, x == 5 \wedge y == 5)$$

At all times, the invariant has been true. However, now time can no longer progress, since otherwise the invariant is violated. State s_0 must thus be left immediately. If for some reason that would not have been possible, then the automaton would be in a deadlock. In this example, however, we can move to s_1, by performing action a.

2.2.2 *Guards and Resets*

An action transition between (s, v) and (s', v') exists only if the current values of the clocks satisfy the guard on the transition between locations s and s' and the *updated* clocks satisfy the invariant of the target location s'. The definition at the top of page 268 misses the latter. We reformulate the rule:

– In the TLTS, an action transition, noted $(s, v) \xrightarrow{a} (s', v')$, exists provided the timed automaton has a transition $(s, a, \varphi, \lambda, s')$ such that v satisfies φ, $v' = v[\lambda := 0]$ and v' satisfies $I(s')$.

This rule states a couple of things. First, a transition can only be taken if the guard is true. Secondly, the resets are executed: all clocks in λ are set to 0. All other clocks remain the same, meaning that taking a transition does not take any time. Thirdly, once arrived in the next state s' the invariant of s' must hold, since, once again, *the invariant must always be true*.

Consider again Figure 17.1 of the textbook. First, execute a delay transition representing waiting for 5 time units. Then, we can execute an action transition:

$$(s0, x == 5 \wedge y == 5) \xrightarrow{a} (s1, x == 5 \wedge y == 0)$$

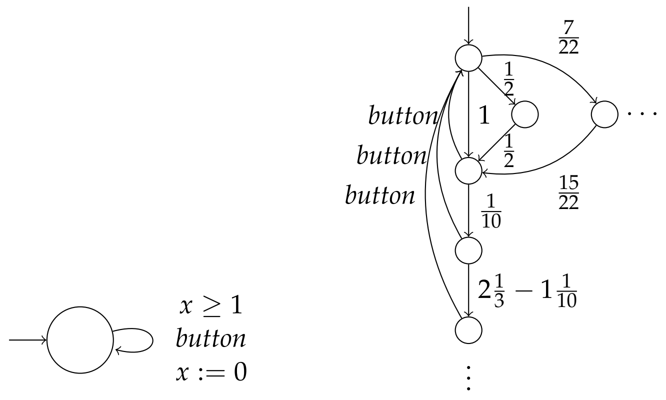
The guards holds, since $y \geq 3$. In the next state, we have reset clock y. Clock x does not change value, since taking a transition does not take time. In the next state, the invariant holds.

At the end of Section 17.1, it is stated that the semantics of a timed automaton can be provided by a TLTS that combines delay and action transitions. At any time and in any state, one can either choose to delay (if possible) or perform an action (if possible). Consider, again, Figure 17.1. Starting in the initial state, one can only delay. Once time has progressed with 3 time units, there are two possibilities: delay further, or perform an action and go to $s1$. Once time has progressed with 5 time units, only one possibility remains: the transition to $s1$ must be taken. Note that it is not the case that once a guard is true, a transition *must* be taken: if the guard is true, the transition *can* be taken.

The peculiar aspect of a TLTS is that it is an infinite structure and therefore difficult to grasp or draw. This comes from the fact that there are infinitely many ways to let time pass.

Consider the example in Figure 3.1a. There is one location and one clock x. The location invariant is simply "true", hence, it is possible to stay in the location forever. There is one self-loop with a "button" action. The timed automaton models a button that can be pressed at any time, but maximally once per time unit.

The TLTS in Figure 3.1b expresses the fact that a button can be pushed after one time unit. In this particular TLTS there is one delay transition of one time unit. But it is also possible to first let 0,5 time unit pass and then another 0,5 pass. It is also possible to let 7/22 time unit pass and after that to let 1 - 7/22 time unit pass, etc. The semantics of our example is an infinite TLTS with all the possible ways to let time pass. Figure 3.1b shows only a minor part of the semantics of the button example.



(A) A button (B) Semantics

FIGURE 3.1 A timed automaton and a part of its semantics

2.3 PARALLEL COMPOSITION

From two or more automata, we can *compose* a new automaton. Intuitively, the composition of two automata models a system in which both automata run independently of each other. Each automaton has its own clocks, and both run asynchronously. There is, however, a way to have them communicate with each other. If the two automata have a transition with the same action a, then both automata will wait until they can perform action a *at the exact same time*. In other words, they *synchronize*.

Reading

Read Section 17.2 on pages 268-269 of Chapter 17 of the textbook "Model Checking".

Note that, according to the definition on page 268 of the textbook, two timed automata synchronize on *all actions that are in the set of actions of both automata*.

Consider, for example, the timed automata in Figure 3.2. The timed automata can synchronize on the shared a-action. The parallel composition of the timed automata is given in Figure 3.3.

2.4 ZENO AND NON-ZENO

The textbook quickly mentions the notion of a Zeno behaviour. Such a behaviour roughly corresponds to an execution where infinitely many actions occur in finite time. Consider Figure 3.4. In this example, it is always possible to perform a *button* action. It is even possible to push the button infinitely many times without letting time pass. This is an example of a Zeno behaviour.

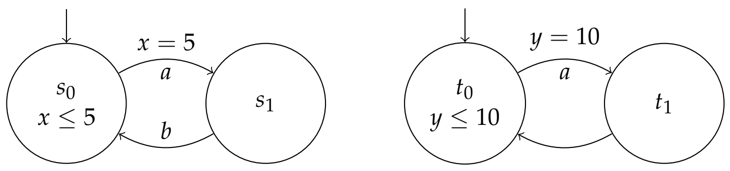


FIGURE 3.2 Two timed automata that can synchronize on *a*

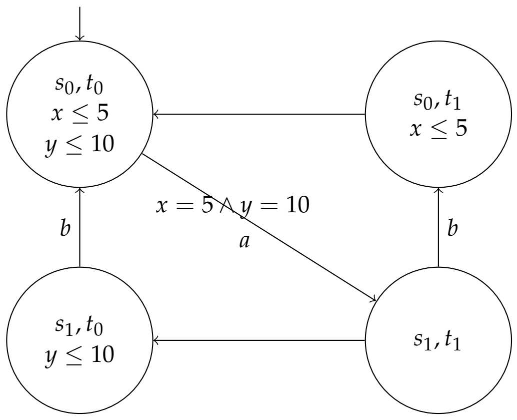


FIGURE 3.3 Timed automaton for the parallel composition of the automata in Figure 3.2

The example in Figure 3.1a has no Zeno behaviour. The guard on the *button* transition together with the clock reset ensures that at least one time unit has elapsed between two button actions. Pushing the button infinitely many times in zero time is no longer possible.

3 Modelling with timed automata

Reading

Read Section 17.3 on pages 269 to 273 of Chapter 17 of the textbook "Model Checking".

This section shows an example of a network of timed automata according to the definitions of the textbook.

4 Modelling timed automata in UPPAAL

UPPAAL is a popular tool for modelling and verifying timed automata. A download link can be obtained through the course website.

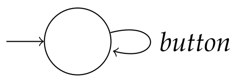


FIGURE 3.4 A button with Zeno behaviour.

UPPAAL automata are essentially the same as the timed automata introduced so far in this chapter, however, their notation differs a bit. In Figure 3.5 we show the timed automaton from Figure 3.3 as it is presented in UPPAAL. Note the initial state is denoted using a double circle. Henceforth we sometimes use the UPPAAL notation for timed automata.

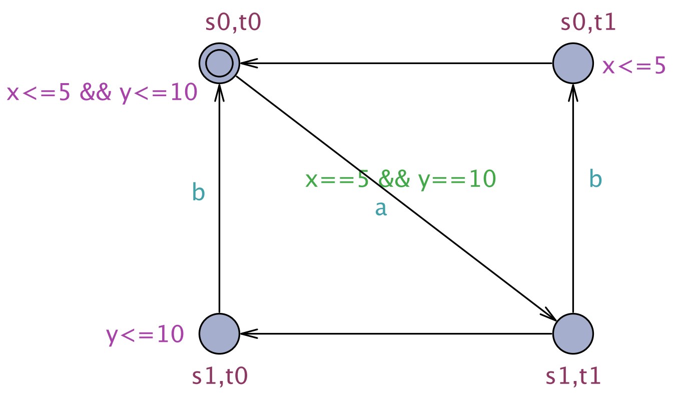


FIGURE 3.5 Timed automaton from Figure 3.3 in UPPAAL notation

An important difference between the timed automata as described in the textbook and the timed automata in UPPAAL is the way synchronization is treated. Whereas in the textbook automata synchronize on transitions with the same action label, transitions in UPPAAL synchronize whenever one transition is labelled with an input, marked by a label starting with ?, for example ?a, and the other transition is labelled with an output, marked by a label starting with !, for example !a.

5 Clock regions

Reading

Read Section 17.4 on pages 274 to 280 of Chapter 17 on page of the textbook "Model Checking". The section contains several lemmas and their proofs. The lemmas are important but you can skip the proofs.

5.1 CLOCK REGIONS

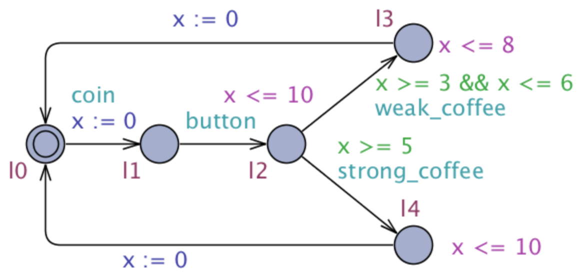


FIGURE 3.6 A timed automaton representing a simple coffee machine.

The definition of a clock region is given on page 275. You do not need to understand all the details of this definition. You need to understand the intuition behind regions. Clock regions are meant to show you a first finite representation of the infinite state-space of timed automata. In this course, we will focus on a refinement of clock regions, called clock zones, which are presented later in this learning unit. It is important to notice that a clock region represents a set of clock values.

Consider the example in Figure 3.6. It shows a coffee machine that first receives a coin and then waits for someone to push a button. Because of the invariant in location l_2, if the button is not pushed within 10 time units, the automaton reaches a deadlock. Progress is no longer possible. If the button is pushed within 3 to 5 time units after insertion of the coin, the machine will produce a weak coffee. If the button is pressed after 6 time units, the machine will produce a strong coffee. If the button is pushed between 5 and 6 time units, the behaviour is non-deterministic. The machine will produce either strong or weak coffee. This automaton has only one clock x. The regions for clock x are determined using points 1 and 2 at the bottom of page 275. Let us determine the clock regions for x. We illustrate point 1 only. You do not need to understand the details of point 2 to continue with this course.

To specify the different regions, we need the largest constant used in any clock constraint. For this example, this constant is 10. As we have only one clock, only point 1 on page 275 applies. The first set of constraints states that there is a region for each point from 0 to 10. So, there is a region for $x = 0$, $x = 1$, $x = 2$, …, until $x = 10$. As clock values are real-numbers, there is a region for all the values in between these integers. That is, there is a region for $0 < x < 1$, $1 < x < 2$, …, until $9 < x < 10$. Finally, there is a region for $x > 10$. In total, we have 22 clock regions. This illustrates the fact that the number of regions is finite but still a very large number. Lemma 43 on page 275 gives an upper bound on the number of clock regions.

Note that there is a small error in Lemma 43. The correct upper bound is:

$$|X|! \cdot 2^{|X|-1} \cdot \prod_{x \in X} (2c_x + 2)$$

Applying this formula to our coffee machine gives 22 regions.

5.2 REGION AUTOMATA

The goal of clock regions is to provide a finite representation of the infinite semantics of a timed automaton. The construction of the region automaton is explained at the bottom of page 279 and the top of page 280.

A region automaton has two types of transitions. Transitions representing the passage of time (these are always without a label), and transitions that represent the transitions of the timed automaton. The definition of the region automaton in the textbook is not completely clear about this. In fact, region automaton $\mathcal{R}(A)$ has:

- an a transition $(s, [\nu]) \xrightarrow{a} (s', [\nu'])$ if and only if $(s, \omega) \overset{a}{\Longrightarrow} (s', \omega')$ for some $\omega \in [\nu]$ and $\omega' \in \nu'$, and
- a delay transition $(s, [\nu]) \to (s, [\nu'])$ if and only if for every $\omega \in [\nu]$ there exists a delay d such that $(s, \omega) \xrightarrow{d} (s, \omega + d)$ and $\omega + d \in \omega'$, and for all $t < d$, $\omega + t \in [\nu] \cup [\nu']$.

In this definition, the first part corresponds to the definition on page 280 of the textbook. The second part states that there is a transition between two states in the region automaton if we can move from one state to the other by delaying without passing through other clock regions. Note that when delaying the location component of the state in the region automaton never changes.

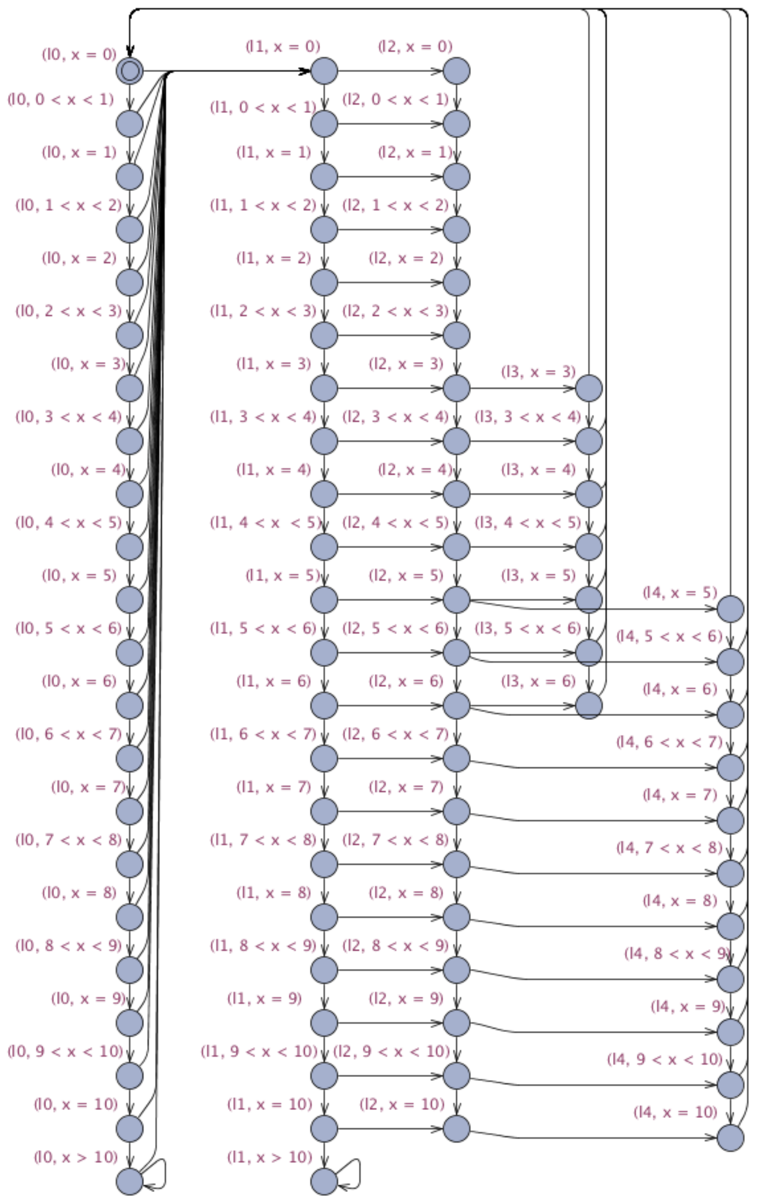


FIGURE 3.7 The region automaton for the timed automaton in Figure 3.6.

Let us construct part of the region automaton for our coffee machine example. The result is pictured in Figure 3.7. We omit the labels on the transitions for the sake of readability. The initial state is $(l_0, x = 0)$. Here are a few examples of other states of this automaton: $(l_0, x = 1)$, $(l_1, x = 5)$, $(l_2, x = 9)$, $(l_2, 9 < x < 10)$, $(l_2, x = 10)$, $(l_3, 8 < x < 9)$, $(l_4, 7 < x < 8)$, etc. The number of states is bounded by 22 regions times 5 locations, so 110 states. The number of reachable states is lower as a state like $(l_2, x > 10)$ is not reachable because of the invariant in l_2. Still, there are many reachable states and as we can see in the Figure 3.7, the region automaton is still a much larger structure than the original timed automaton. Let us have a closer look at this region automaton. All transitions going out from states with location l_0 lead to state $(l_1, x = 0)$. This is due to the clock reset on the transition between location l_0 and l_1 in the original timed automaton. The same holds for all transitions going out of states with locations l_3 and l_4. State $(l_1, x > 10)$ has only one self transition. The only possibility is to let time pass forever. Indeed, if the system waits more than 10 time units for the button, the transition in the timed automaton from l_1 to l_2 is no longer possible. The system will stay in location l_1 until infinity. The behaviour of the transitions between states with locations l_2, l_3, and l_4 is also clear in the region automata. If the button is pushed between at least 3 and before 5 time units, then the system produces weak coffee. If the button is pushed after 5 to 6 time units, the system has a non-deterministic behaviour. It can either produce strong or weak coffee. After 6 time units, only transitions to l_4 are possible and the system is deterministic.

QUESTION 3.2
Assume that instead of using a single clock, a model would use a clock x for weak coffee and a clock y for strong coffee. How would the number of regions change? How would the region automaton evolve? You do not have to rewrite a new region automaton but just give a qualitative argument.

Theorem 31 on page 280 justifies the fact that the region graph is equivalent to the infinite graph of the semantics. This justifies the use of the region graph for verification. As we have seen in the examples of this section, the region graph is extremely large as the number of regions grows exponentially with the number of clocks and the value of the constants in clock constraints. Therefore, it is not completely satisfactory. The clock zones abstraction presented in the next section provides a more concise abstraction. It follows a similar construction. Zones are used instead of regions. A zone automaton is built instead of a region automaton.

6 Clock zones

Reading

We now summarize the content of Section 17.5 on pages 280 to 287 of Chapter 17 of the textbook "Model Checking". The presentation in the texbook is a bit confusing. You do not have to read this section of the textbook. Instead, you must read the explanations below.

6.1 DEFINITION

A clock zone basically is a constraint on a clock or a pair of clocks. For instance, $x = 0$, $x < 10$ or $x - y < 5$. A clock zone, as a clock region, represents a set of clock values. These values correspond to the solution set of the clock zone. Formally, a clock zone is a constraint of the form $x \prec c$, $c \prec x$, or $x - y \prec c$ where x, y are clocks, $\prec \in \{<, \leq\}$, and c is an integer. Note that c can be negative.

Using this definition, there are many ways to write the same constraint. To obtain a more uniform notation, a special clock is introduced. This clock is named x_0 and is always equal to 0. Assume a time automaton with a set of clocks $\mathcal{C}$. Let $\mathcal{C}_0 = \mathcal{C} \cup \{x_0\}$. Any clock zone is represented as $x - y \prec c$, where $x, y \in \mathcal{C}_0$, $c \in \mathbb{Z}$, and $\prec \in \{<, \leq\}$. Now, constraint $x_1 \leq 1$ is written as $x_1 - x_0 \leq 1$.

There are three operations on clock zones used to explore the state-space of a timed automaton. We briefly describe these operations. Section 7 shows algorithms to compute these operations efficiently. Figure 3.8 shows two zones (φ and ψ) with clocks x_1, x_2 and the results of the three operations below.

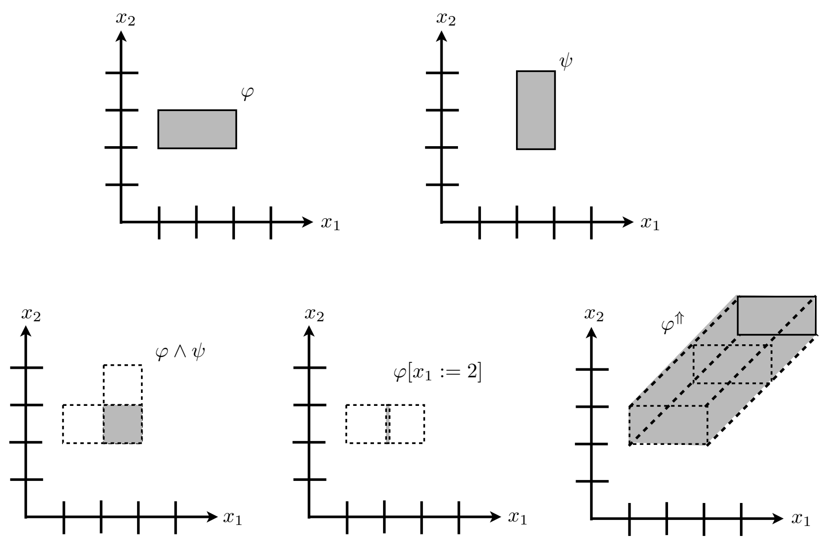


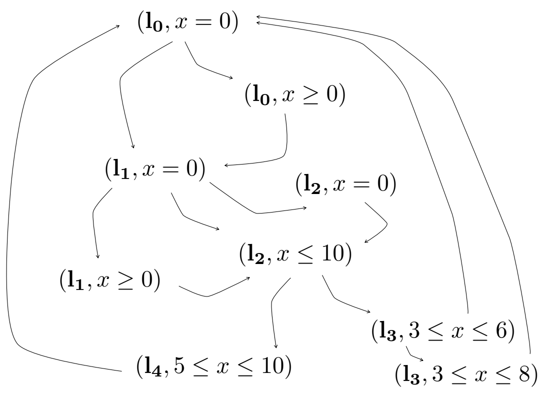
FIGURE 3.8 Operations on zones φ and ψ.

Intersection.
The intersection of two clock zones ψ and φ is their conjunction $\psi \wedge \varphi$. A zone is a conjunction of constraints. A conjunction of clock zones is still a clock zone.

Clock reset.
The reset operation is the projection of the clock onto one dimensional subspace. Projecting a zone on a axis results in a clock zone. Figure 3.8 shows the zone obtained by resetting clock x_1 to 2 in the bottom center.

Elapsing of time.
This operation is best explained in the bottom right drawing in Figure 3.8. This drawing shows the zone obtained by letting time pass from zone φ. This zone is noted $\varphi^{\Uparrow}$. Note that this zone seems to stop in the Figure but it does not. Time never stops. The zone continues expanding in the same direction infinitely. The zone expands with a 45° angle. This corresponds to the fact that all clocks grow at the same rate.

6.2 ZONE AUTOMATON



$(\mathbf{l_0}, x = 0)$
$(\mathbf{l_0}, x \geq 0)$
$(\mathbf{l_1}, x = 0)$
$(\mathbf{l_2}, x = 0)$
$(\mathbf{l_2}, x \leq 10)$
$(\mathbf{l_1}, x \geq 0)$
$(\mathbf{l_3}, 3 \leq x \leq 6)$
$(\mathbf{l_4}, 5 \leq x \leq 10)$
$(\mathbf{l_3}, 3 \leq x \leq 8)$

FIGURE 3.9 The zone automaton for the timed automaton in Figure 3.6.

The construction of the zone automaton is similar to the region automaton. States are pairs of a location and a clock zone. Figure 3.9 gives the zone automaton for our simple coffee machine.

The initial location is l_0 and clock x is initially 0, represented by zone $x = 0$. In the timed automaton, it is either possible to let time pass or to move to l_1. As the clock is reset on the transition to l_1, the zone reached in l_1 is $x = 0$. The zone $x \geq 0$ represents states where time can elapse up to infinity. Location l_2 can be reached in zero time but then the clock is at most 10. This is represented by the zone $x \leq 10$. Finally, the zones in locations l_3 and l_4 represent the fact that weak and strong coffee are produced in the time intervals [3:6] and [5:10]. It is clear from this example that the zone automaton is a much more concise abstraction than the region automaton. The zone representation is a symbolic representation of the infinite state space of a timed automaton in a similar sense as BDDs are a symbolic representation of Kripke structures (see Learning Unit 6 in the next block). Another advantage of zones is that they can be manipulated efficiently. The next subsection discusses the use of Difference Bound Matrices to efficiently manipulate zones.

As the three operations on zones still produce a zone, it is easy and safe to manipulate a zone. In particular, the computation of the reachable zone from a given state is done as follows. Assume a timed automaton A and a transition $e = (l, a, \psi, \lambda, l')$. Let $Z(A)$ be the zone automaton of A. Assume that the current state in $Z(A)$ is (l, φ). So, the timed automaton is in location l ready to execute action a. The successor zone of φ after transition e is noted $succ(\varphi, e)$. It represents the zone that can be reached by letting time pass and executing the action on the transition. In the zone automaton, the successor state of (l, φ) will be $(l', succ(\varphi, e))$, which is computed by the following steps:

1 Compute the possible delays in location s. This is obtained by taking the intersection of the current zone with the invariant in location l, that is, one computes $\varphi \wedge I(l)$.
2 We let time pass using operator $\Uparrow$, that is, one computes $(\varphi \wedge I(l))^{\Uparrow}$.
3 The previous operation might allow delays prevented by the invariant in the target location l', so, we take the intersection with the target invariant. One computes $(\varphi \wedge I(l))^{\Uparrow} \wedge I(l')$.
4 We then take the intersection with the guard ψ of the transition. One computes $(\varphi \wedge I(l))^{\Uparrow} \wedge I(l') \wedge \psi$.
5 Finally, all the clocks in λ must be reset.

This results in the successor formula

$$succ(\varphi, e) = ((\varphi \wedge I(s))^{\Uparrow} \wedge I(s') \wedge \psi)[\lambda := 0]$$

A version of this formula can be found on page 283 of the textbook. The version in the book only works for self-transition, that is, in case $l = l'$.

7 Difference bound matrices

Reading

Read Section 17.6 on pages 287 to 291 of Chapter 17 of the textbook "Model Checking".

7.1 ZONES AND DBMS

Let $\mathcal{C}$ be a set of clocks for Timed Automaton A. Let $\mathcal{C}_0 = \mathcal{C} \cup \{x_0\}$. A Difference Bound Matrix (DBM) is a $|\mathcal{C}_0| \times |\mathcal{C}_0|$ matrix. Every element in the DBM is a bound on the difference between two clocks. Every clock is numbered from 0 to $|\mathcal{C}_0| - 1$. Each row of the matrix denotes one clock. The row is used to represent the lower bound of the difference between the clock and all other clocks. The column is used to store the upper bounds between the clock and all other clocks. Consider the example DBM at the bottom of page 287. Let M be that matrix. Let $M_{i,j}$ be the element at the intersection of row i and column j.

- $M_{1,2}$ means that $x_1 - x_2 < 2$
- $M_{2,1}$ means that $x_2 - x_1$ is not bounded
- $M_{2,0}$ means that $x_2 - x_0 \leq 2$
- $M_{0,2}$ means that $x_0 - x_2 < 2$, that is, $x_2 \geq 0$
- $M_{1,1}$ means that $x_1 - x_1 \leq 0$

Each element in the matrix, say M, is computed according to the following four steps

1 For each constraint $x_i - x_j \prec n$, insert $(n, \prec)$ for $M_{i,j}$.

2 For each clock difference $x_i - x_j$ that is unbounded add $(\infty, <)$ at $M_{i,j}$. A clock difference is unbounded if it is not mentioned in the set of constraints of the zone.
3 Add the implicit constraints that all clocks are positive, that is, add the constraints $x_0 - x_i \leq 0$. Note that this might replace some of the $(\infty, <)$ elements introduced in the previous step.
4 Add the implicit constraints that the difference between a clock and itself is always 0, that is, add the constraints $x_i - x_i \leq 0$.

Let us consider the zone described at the bottom of page 287:

$$x_1 - x_2 < 2 \wedge 0 < x_2 \leq 2 \wedge 1 \leq x_1$$

We first rewrite these constraints according to the uniform notation. We obtain:

$$x_1 - x_2 < 2 \wedge x_0 - x_2 < 0 \wedge x_2 - x_0 \leq 2 \wedge x_0 - x_1 \leq -1$$

Applying the first step, we obtain the following elements:

$$M_{1,2} = (2, \leq) \wedge M_{0,2} = (0, <) \wedge M_{2,0} = (2, \leq) \wedge M_{0,1} = (-1, \leq)$$

The following differences are not bounded: $x_1 - x_0$ and $x_2 - x_1$. This gives the following elements:

$$M_{1,0} = (\infty, <) \wedge M_{2,1} = (\infty, <)$$

There are already constraints that ensure that clocks x_1 and x_2 are positive. The last elements simply state that the difference between a clock and itself is always 0. This creates the following elements $M_{i,i} = (0, \leq), \forall i \in \{0, 1, 2\}$.

7.2 CANONICAL DBMS

At the top of page 288 of the textbook, one can find a matrix which has the same solution set as the one described on the previous page. Each matrix represents a different zone, but the sets of clock values satisfying the constraints of each zone are equal. In general, there is an infinite number of zones sharing the same solution set. This is not practical for developing algorithms for DBMs. Fortunately, for each zone there exists a canonical DBM for it, that is, there exists a unique representation of the solution set of a family of zones.

Page 288 explains how to obtain this canonical form. The basic idea is to notice that if we know $x_i - x_j \prec n$ and $x_j - x_k \prec n$, we can deduce that $x_i - x_k \prec n + m$. The idea is then to use this fact to tighten the bounds in a matrix.

7.3 OPERATIONS ON DBMS

Three basic operations on DBMs are described on page 289. The description is not optimized as operations may results in non-canonical DBMs. Nevertheless, the textbook gives the intuition behind these operations.

Intersection means the conjunction of constraints. For upper bounds on differences, this means keeping the lowest bound. Say if two zones have $x_1 - x_2 \leq 5$ and $x_1 - x_2 \leq 10$, the conjunction of these zones will keep $x_1 - x_2 \leq 5$.

Clock reset is rather straightforward. If the two clocks of a difference are reset, the difference is also 0. If only one clock is reset, the difference of this clock with clock x_0 is set to 0.

7.4 THE COFFEE MACHINE REVISITED

Let us come back to the coffee machine example presented in Figure 3.6 and Figure 3.9. We will now compute the zones of the zone automaton using DBMs.

Elapsing of time simply removes the upper bound on all individual clocks. This means that all individual clocks are allowed to be increased by any amount of time.

For the initial zone Z_0, clock x is 0. We have the following constraints:

$$Z_0 \equiv x - x_0 \leq 0 \wedge x_0 - x \leq 0$$

Note that we omitted $x - x \leq 0$ and $x_0 - x_0 \leq 0$.

The DBM $M(Z_0)$ is the following:

$$M(Z_0) = \begin{pmatrix} (0, \leq) & (0, \leq) \\ (0, \leq) & (0, \leq) \end{pmatrix}$$

As the invariant in location l_0 is "True", it is possible to delay for any amount of time in location l_0. The successor zone Z_1 of zone Z_0 is obtained by letting time pass. The operation lifts the upper bound on x. This results in the following DBM:

$$M(Z_1) = \begin{pmatrix} (0, \leq) & (0, \leq) \\ (\infty, \leq) & (0, \leq) \end{pmatrix}$$

The only possible transition from l_0 is a "coin" transition to l_1. There is no guard and the invariant in location l_1 is also "True". Clock x is reset on the transition and after applying this operation to $M(Z_1)$ we are therefore back to $M(Z_0)$.

It is possible to delay for any amount of time in l_1. The zone obtained after this delaying operation is $M(Z_1)$.

From l_1, we can now make a "button" transition to l_2. The invariant in l_2 states that $x \leq 10$. This results in the following DBM:

$$M(Z_2) = \begin{pmatrix} (0, \leq) & (0, \leq) \\ (10, \leq) & (0, \leq) \end{pmatrix}$$

From l_2, two transitions are possible. After the "weak coffee" transition, the conjunction of the guard and the invariant ensures that x is between 3 and 6. The resulting DBM is as follows:

$$M(Z_3) = \begin{pmatrix} (0, \leq) & (-3, \leq) \\ (6, \leq) & (0, \leq) \end{pmatrix}$$

If we take the "strong coffee" transition, the DBM becomes:

$$M(Z_4) = \begin{pmatrix} (0, \leq) & (-5, \leq) \\ (10, \leq) & (0, \leq) \end{pmatrix}$$

From location l_4, the transition back to the initial state results in $M(Z_0)$. From location l_3, the same transition is possible. When entering l_3, the clock is at most 6. This means that a delay of 2 time units is still possible. This delay is represented by the following DBM:

$$M(Z_5) = \begin{pmatrix} (0, \leq) & (-3, \leq) \\ (8, \leq) & (0, \leq) \end{pmatrix}$$

8 Conclusion

Reading

Read Section 17.7 on pages 291 and 292 of Chapter 17 on page of the textbook "Model Checking".

The remark about the fact that DBMs constitute an explicit state-space exploration is not completely correct. It really is a symbolic representation of the infinite state-space of a timed automaton. On the other hand, it is an explicit representation of the state-space of the zone automaton.

Fore more detail about the semantics of timed automata and associated algorithms, we suggest you to read the paper "Timed Automata: Semantics, Algorithms, and Tools" by Bengtsson and Yi. You will find this article in the reader. Reading this article is optional.

EXERCISES

1 (Zones and DBMs)

Consider the following zone with clocks x_1 and x_2.

$$x_1 \leq 20 \wedge x_2 \leq 4 \wedge 6 \leq x_1 - x_2 \leq 9 \wedge x_1 \geq 2$$

Give the canonical DBM for this zone.

2 (Successors of DBMs)

Compute the DBM for the reachable zones from the zone of exercise 1 by delaying and according to the following invariant/guard:

$$5 \leq x_1 \leq 8 \wedge 1 \leq x_2 \leq 3$$

Of course, your result must be a canonical DBM.

FEEDBACK

Answers to questions

3.1 The system waits for 3 time units in location s_0 and then makes a transition to s_1. When entering s_1, clock y is equal to 0 and clock x is equal to 3. It stays in s_1 for 5 time units and make a transition to s_0. Clock y is now 5 and clock x is 0. Then, the system makes a transition to s_1 without letting time pass. Now, clock y is reset to 0 and clock x has not been increased. Clock x equals 0 as well. Nothing prevents the system for waiting 8 time units in s_1. At that time, clocks x and y are both equal to 8. The invariant in s_1 does not allow to let more time pass. The invariant in s_0 prevents a "b" transition. The system can no longer make a transition or let time pass. The system can no longer make progress. It has reached a deadlock.

3.2 The number of regions grows exponentially with the number of clocks. This is clear in Lemma 43 on page 275 of the textbook (see also the corrected version in the workbook). The number of clocks appears in a factorial and in a power of 2. Clearly, a good practice to reduce the state space and ease verification is to minimize the number of clocks used in a model.

Answers to exercises

1 We first introduce x_0 and write the zone in the uniform notation:

$$x_1 - x_0 \leq 20 \wedge x_2 - x_0 \leq 4 \wedge x_2 - x_1 \leq -6 \wedge x_1 - x_2 \leq 9 \wedge x_0 - x_1 \leq -2$$

We omitted the differences between each clock and itself and the implicit constraint that x_2 is positive. This comes from the fact that $x_2 \geq 0$ is the same as $-x_2 \leq 0$ that in turn is the same as $x_0 - x_2 \leq 0$, and indeed, this is the missing difference (apart from $x_i - x_i$ for $i \in \{0, 1, 2\}$). This gives us the following matrix:

$$\begin{pmatrix} (0, \leq) & (-2, \leq) & (0, \leq) \\ (20, \leq) & (0, \leq) & (9, \leq) \\ (4, \leq) & (-6, \leq) & (0, \leq) \end{pmatrix}$$

Now we should try to tighten the bounds. Let us first try to reduce the bound $x_1 - x_0 \leq 20$.
We know that $x_1 - x_2 \leq 9$ and $x_2 - x_0 \leq 4$. Thus, we can deduce that $x_1 - x_0$ must not be greater than 13. The exact computation is that $x_1 - x_2 + x_2 - x_0 \leq 9 + 4$ gives $x_1 - x_0 \leq 13$. This reduces the bound and we have to update the matrix as follows:

$$\begin{pmatrix} (0, \leq) & (-2, \leq) & (0, \leq) \\ (13, \leq) & (0, \leq) & (9, \leq) \\ (4, \leq) & (-6, \leq) & (0, \leq) \end{pmatrix}$$

One can now try to reduce any bounds but one will see that this is no longer possible. The DBM is in its canonical form.

2 We first let time pass. This gives the following DBM:

$$\begin{pmatrix} (0,\leq) & (-2,\leq) & (0,\leq) \\ (\infty,\leq) & (0,\leq) & (9,\leq) \\ (\infty,\leq) & (-6,\leq) & (0,\leq) \end{pmatrix}$$

Then, we compute the DBM for the invariant/guard. This gives the following matrix:

$$\begin{pmatrix} (0,\leq) & (-5,\leq) & (-1,\leq) \\ (8,\leq) & (0,\leq) & (\infty,\leq) \\ (3,\leq) & (\infty,\leq) & (0,\leq) \end{pmatrix}$$

We now take the conjunction of these two matrices. This gives the following:

$$\begin{pmatrix} (0,\leq) & (-5,\leq) & (-1,\leq) \\ (8,\leq) & (0,\leq) & (9,\leq) \\ (3,\leq) & (-6,\leq) & (0,\leq) \end{pmatrix}$$

We now have to reduce it to its canonical form.
Let us look at $x_2 - x_0 \leq 3$. We have $x_2 - x_1 \leq -6$ and $x_1 - x_0 \leq 8$. We deduce $x_2 - x_0 \leq 2$.
This gives the following matrix:

$$\begin{pmatrix} (0,\leq) & (-5,\leq) & (-1,\leq) \\ (8,\leq) & (0,\leq) & (9,\leq) \\ (2,\leq) & (-6,\leq) & (0,\leq) \end{pmatrix}$$

Let us now look at $x_1 - x_2 \leq 9$. We know that $x_1 - x_0 \leq 8$ and $x_0 - x_2 \leq -1$. So, we have $x_1 - x_2 \leq 7$.
This gives the following matrix:

$$\begin{pmatrix} (0,\leq) & (-5,\leq) & (-1,\leq) \\ (8,\leq) & (0,\leq) & (7,\leq) \\ (2,\leq) & (-6,\leq) & (0,\leq) \end{pmatrix}$$

Finally, we look at $x_0 - x_1 \leq -5$. We know that $x_0 - x_2 \leq -1$ and $x_2 - x_1 \leq -6$. So, we have $x_0 - x_1 \leq -7$.

$$\begin{pmatrix} (0,\leq) & (-7,\leq) & (-1,\leq) \\ (8,\leq) & (0,\leq) & (7,\leq) \\ (2,\leq) & (-6,\leq) & (0,\leq) \end{pmatrix}$$

One can check that it is no longer possible to tighten any bound. This matrix is in canonical form. Note that the order in which you reduce the constraints has no influence on the final result. You can reduce $x_1 - x_2 \leq 9$ before $x_2 - x_0 \leq 3$, you will obtain the same final result.

Block 2

Formal specification

Temporal logics

Learning unit 4

Temporal logics

INTRODUCTION

Learning unit 4 starts the second block, which is about CTL, computation tree logic.

LEARNING GOALS
After having studied this learning unit you should be able to:
– understand the syntax and semantics of CTL
– write temporal properties to specify the correct behavior of systems.

Reading In this learning unit, the textbook is not used. Chapter 3 of the textbook provides background information for the interested reader.

Exercises You should do the exercises at the end of this learning unit.

Time The expected time needed to study this learning unit is about 6 hours.

KERNEL

1 Temporal Logics

In propositional logic, there are propositions such as "It is Monday" or "It is raining" that can be either true or false. In predicate logic, there are predicates such as "x is prime" or "x is a brother of y" that can be either true or false. In such logics, formulae are evaluated within a single state. In *temporal* logics, a notion of time is added. The proposition "It is Monday" may be true now, but not tomorrow. In other words, a temporal logic allows to reason about the current state of a system, about the next state of the system, and even about all possible next states of a system.

2 CTL: Syntax

There are various kinds of temporal logics, but for this course, we only consider *Computation Tree Logic* (CTL). A statement in CTL is of the following form:

$$M, s \models \phi$$

This can be pronounced as "formula ϕ holds in state s of Kripke structure M". Consider the Kripke structure called K in Figure 4.1. We might state:

$$K, s_1 \models q \wedge \neg p$$

This states that atomic propositions q and $\neg p$ hold in state s_1. CTL allows us to express properties not only over a state, but over *executions* starting in that state as well, using *temporal operators*.

Central to CTL is the notion of an *execution tree*. Given a state s, it is possible to unwind a Kripke structure. This produces a tree that starts in state s and contains all execution paths starting in state s. An *execution path* is an infinite sequence of states. For example, Figure 4.2 shows the unwinding of K in state s_0. Note that since the transition relation of a Kripke structure is total, the tree is infinite (hence the dots below the tree). CTL never reasons about finite trees, i.e., each execution path in the tree is always *infinite*.

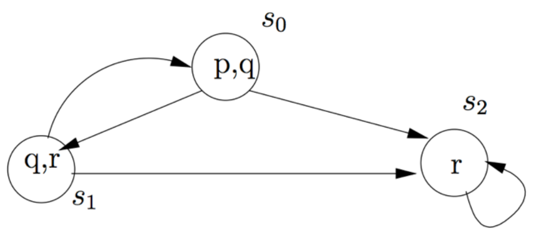


FIGURE 4.1 Kripke structure K

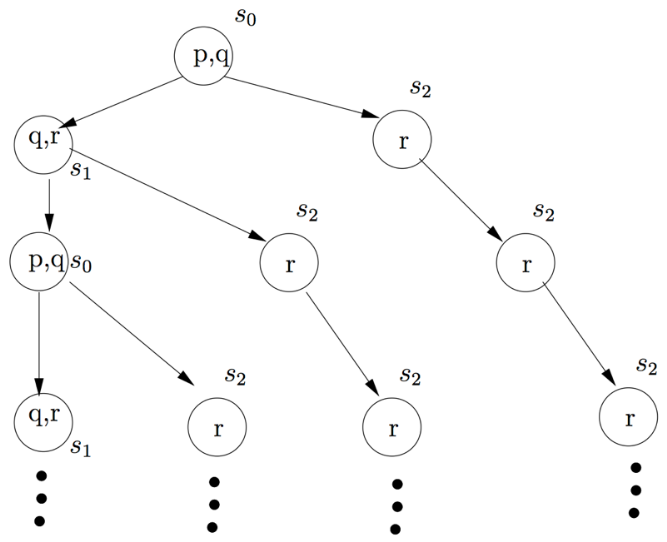


FIGURE 4.2 Unwinding of state s_0 of Kripke structure K

The temporal operators of CTL express properties over execution trees. It is possible to express that *for all* execution paths of a tree a certain path property should hold or that there must *exist* an execution path in the tree where a certain path property holds. An example of a path property is that *for all* states in the path, a certain state property p holds. Another example of a path property is that in the given execution path, there must *exist* some state where the state property holds.

The syntax of CTL is defined as follows:

$$\begin{array}{rcl} \phi & ::= & p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \\ & & \mathbf{AX}(\phi) \mid \mathbf{EX}(\phi) \mid \mathbf{AG}(\phi) \mid \mathbf{EG}(\phi) \mid \mathbf{AF}(\phi) \mid \mathbf{EF}(\phi) \mid \\ & & \mathbf{A}(\phi\mathbf{U}\phi) \mid \mathbf{E}(\phi\mathbf{U}\phi) \mid \mathbf{A}(\phi\mathbf{R}\phi) \mid \mathbf{E}(\phi\mathbf{R}\phi) \end{array}$$

A formula ϕ is a CTL formula if it is an atomic proposition p, the negation of a CTL formula, or the conjunction or disjunction of a CTL formula. Note that implication can be modelled using negation and disjunction. A CTL formula can also be a temporal operator applied to a CTL formula. Figure 4.3 illustrates the temporal operators.

Five operators express properties over *all* execution paths of a tree.

$\mathbf{AX}(p)$	For all execution paths, p holds in the next state of the path.
$\mathbf{AG}(p)$	For all execution paths, p holds in all states of the path.
$\mathbf{AF}(p)$	For all execution paths, p holds in some state of the path.
$\mathbf{A}(p\ \mathbf{U}\ q)$	For all execution paths, a state is reached where q holds. Until then, p holds.
$\mathbf{A}(p\ \mathbf{R}\ q)$	For all execution paths, p releases q, or q holds permanently.

Five operators express properties over *some* execution path of a tree.

$\mathbf{EX}(p)$	There exists an execution path where p holds in the next state of that path.
$\mathbf{EG}(p)$	There exists an execution path where p holds in all states of that path.
$\mathbf{EF}(p)$	There exists an execution path where p holds in some state of that path.
$\mathbf{E}(p\ \mathbf{U}\ q)$	There exists an execution path where a state is reached where q holds. Until then, p holds.
$\mathbf{E}(p\ \mathbf{R}\ q)$	There exists an execution path where p releases q, or q holds permanently.

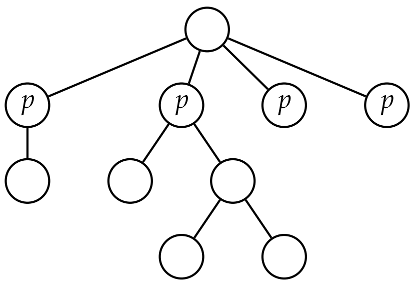
Operator "next" X

Formula $\mathbf{AX}(p)$ means that all successor states should satisfy atomic proposition p. Formula $\mathbf{EX}(p)$ means that there must exist a successor that satisfies p. In the example in Figure 4.1, the formula $\mathbf{AX}(r)$ holds in states s_0 and s_2 of the Kripke structure. In state s_1, the property does not hold, since there exists a successor where r does not hold. In that state, the formula $\mathbf{EX}(r)$ does hold.

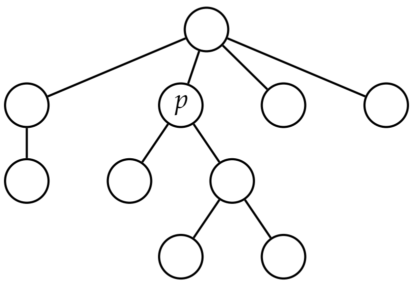
QUESTION 4.1
Does property $\mathbf{AX}(\mathit{flag} = 01)$ hold in the initial state of the solution to Exercise 2.1 given in Learning Unit 2? And $\mathbf{EX}(\mathit{flag} = 01)$?

Operator "globally" G

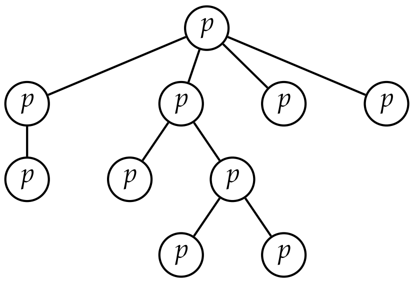
Formula $\mathbf{AG}(p)$ means that atomic proposition p holds *always globally*, i.e., it must always be true at any execution path. Formula $\mathbf{EG}(p)$ means that atomic proposition p holds *possibly globally*. In Figure 4.1 the formula $\mathbf{AG}(r)$ holds only in state s_2. Formula $\mathbf{EG}(q)$ holds in s_0 and s_1.



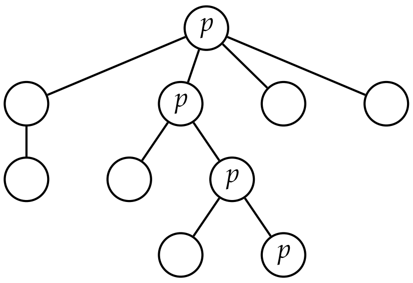
(A) $\mathbf{AX}(p)$



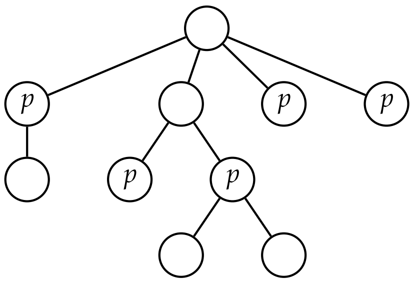
(B) $\mathbf{EX}(p)$



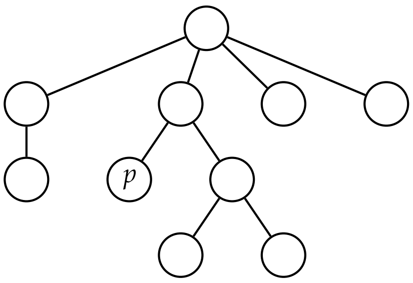
(C) $\mathbf{AG}(p)$



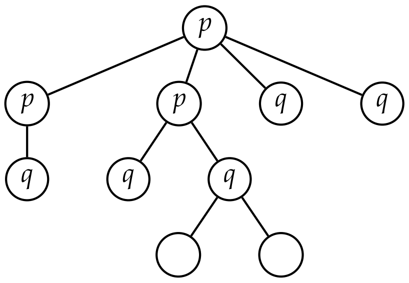
(D) $\mathbf{EG}(p)$



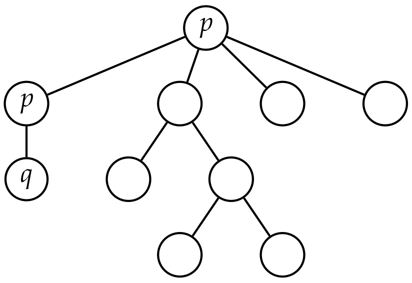
(E) $\mathbf{AF}(p)$



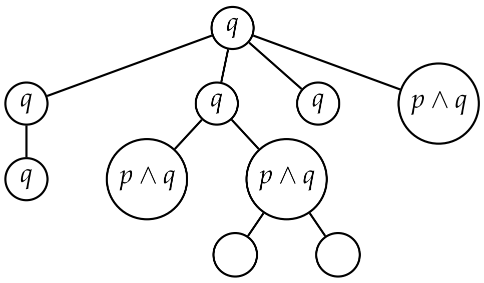
(F) $\mathbf{EF}(p)$



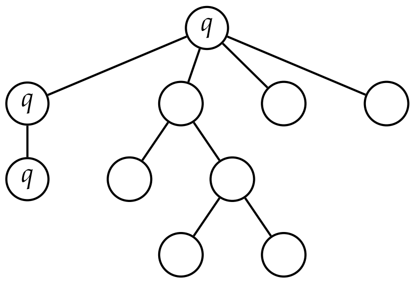
(G) $\mathbf{A}(p\ \mathbf{U}\ q)$



(H) $\mathbf{E}(p\ \mathbf{U}\ q)$



(I) $\mathbf{A}(p\ \mathbf{R}\ q)$



(J) $\mathbf{E}(p\ \mathbf{R}\ q)$

FIGURE 4.3 Illustrations of CTL

QUESTION 4.2
Consider Exercise 2.1 of Learning Unit 2. Write a property stating that the two processes never access the critical section at the same time. Does the model solution satisfy this property?

Operator "eventually" F

Formula $\mathbf{AF}(p)$ means that atomic proposition p holds *always eventually*. Formula $\mathbf{EF}(p)$ means that atomic proposition p holds *possibly eventually*. In Figure 4.1 the formula $\mathbf{AF}(r)$ holds in all states. Formula $\mathbf{AF}(p)$ holds in s_0: even though there exists an execution path that ends in state $\{c\}$, all execution paths starting in s_0 have a state where p holds: the first one. Formula $\mathbf{EF}(\neg q \wedge r)$ holds in all states: in all states it is possible to reach state s_2. Formula $\mathbf{EF}(q)$ does not hold in state s_2, since in that state, there is no execution that reaches a state in which q is true.

QUESTION 4.3
Consider Exercise 2.2 of Learning Unit 2. Write a property expressing the fact that the goat can possibly be eaten by the wolf. Is there a path in the model solution where this property holds?

QUESTION 4.4
Consider Exercise 2.2 of Learning Unit 2. Write a property expressing the fact that there exists a path where eventually the wolf, the cabbage, and the goat are on the right side of the river.

Operator "until" U

Formula $\mathbf{A}(p\ \mathbf{U}\ q)$ – *always p until q* – expresses two things. First, property q holds always eventually. Secondly, in all states that come before the state where q holds, p holds. Thus in Figure 4.1 in state s_0 formula $\mathbf{A}(q\ \mathbf{U}\ r)$ is true. Formula $\mathbf{A}(q\ \mathbf{U}\ \neg q)$ is not true in state s_0, since there exists an execution path that does not lead to $\neg q$. The existential version $\mathbf{E}(p\ \mathbf{U}\ q)$ then expresses that property q holds possibly eventually, and on a path leading to q property p holds. In Figure 4.1 formula $\mathbf{E}(q\ \mathbf{U}\ \neg q \wedge r)$ is true in state s_0, but $\mathbf{E}(r\ \mathbf{U}\ \neg q \wedge r)$ is false. Note that "until" means "up to" and not "up to and including".

QUESTION 4.5
Consider Exercise 2.1 of Learning Unit 2. Write a CTL property stating that variable *turn* starts with value 0 but will eventually change its value to 1.

Operator "release" R

Imagine a soldier standing guard. A good soldier will remain standing guard, until he is released. If he is not released, he will forever stand guard. This is expressed by the release operator. Formula $\mathbf{A}(p\ \mathbf{R}\ q)$ – *p releases q* – expresses that q holds up to and including the first state where p holds. If p never happens, then q will remain true. Its existential version $\mathbf{E}(p\ \mathbf{R}\ q)$ states the same for some execution path.

$M, s \models p$	iff	$p \in L(s)$
$M, s \models \neg f$	iff	$\neg(M, s \models f)$
$M, s \models f_0 \wedge f_1$	iff	$M, s \models f_0$ and $M, s \models f_1$
$M, s \models f_0 \vee f_1$	iff	$M, s \models f_0$ or $M, s \models f_1$
$M, s \models \mathbf{AX}f$	iff	$\forall \pi \in \Pi(M, s) \cdot M, \pi[1] \models f$
$M, s \models \mathbf{EX}f$	iff	$\exists \pi \in \Pi(M, s) \cdot M, \pi[1] \models f$
$M, s \models \mathbf{AG}f$	iff	$\forall \pi \in \Pi(M, s) \cdot \forall i \cdot M, \pi[i] \models f$
$M, s \models \mathbf{EG}f$	iff	$\exists \pi \in \Pi(M, s) \cdot \forall i \cdot M, \pi[i] \models f$
$M, s \models \mathbf{AF}f$	iff	$\forall \pi \in \Pi(M, s) \cdot \exists i \cdot M, \pi[i] \models f$
$M, s \models \mathbf{EF}f$	iff	$\exists \pi \in \Pi(M, s) \cdot \exists i \cdot M, \pi[i] \models f$
$M, s \models \mathbf{A}(f_0 \,\mathbf{U}\, f_1)$	iff	$\forall \pi \in \Pi(M, s) \cdot \exists k \cdot (M, \pi[k] \models f_1) \wedge (\forall_{i<k} \cdot M, \pi[i] \models f_0)$
$M, s \models \mathbf{E}(f_0 \,\mathbf{U}\, f_1)$	iff	$\exists \pi \in \Pi(M, s) \cdot \exists k \cdot (M, \pi[k] \models f_1) \wedge (\forall_{i<k} \cdot M, \pi[i] \models f_0)$
$M, s \models \mathbf{A}(f_0 \,\mathbf{R}\, f_1)$	iff	$\forall \pi \in \Pi(M, s) \cdot \forall k \cdot (\forall_{i<k} \cdot M, \pi[i] \models \neg f_0) \implies (M, \pi[k] \models f_1)$
$M, s \models \mathbf{E}(f_0 \,\mathbf{R}\, f_1)$	iff	$\exists \pi \in \Pi(M, s) \cdot \forall k \cdot (\forall_{i<k} \cdot M, \pi[i] \models \neg f_0) \implies (M, \pi[k] \models f_1)$

TABLE 4.1 Formal Semantics of CTL

QUESTION 4.6
For each of the trees in Figure 4.3, try to find a different assignment of atomic propositions to states such that 1.) the formula still holds, and 2.) your assignment contains a minimal number of p's.

3 CTL: Formal Semantics

In the previous section, we saw that in order to assess whether a CTL formula holds or not, we have to reason about execution paths. An execution path is an infinite sequence of states $[s_0, s_1, s_2, \ldots]$, where each consecutive pair of states s and t is such that $(s, t) \in R$. Given a Kripke structure $M = (S, S_0, R, L)$ (see page 14 of the textbook) and a state s, the set of execution paths is defined as:

$$\Pi(M, s) \equiv \{\pi \mid \pi[0] = s \wedge \forall n \cdot (\pi[n], \pi[n+1]) \in R\}$$

We use the letter π to refer to execution paths, and $\pi[n]$ to refer to the nth state of that path. Table 4.1 defines the semantics of CTL inductively.

In practice, we will often omit M when writing expressions, if it is clear from the context which Kripke structure we are talking about. Moreover, we will also omit the state s. If the state is omitted, then we are talking about the initial state s_0. For example, if we are considering the Kripke structure in Figure 4.1 and we say:

$$\models \mathbf{EG}(q)$$

then this statement is true since there exists an execution path where atomic proposition q holds globally starting in the initial state.

4 Dualities and Logical Equalities

At this point,you might wonder whether all fourteen operators are necessary. The until and the release do look alike, do we need them both? Can we express one in terms of the other? Indeed, this is possible. The minimum set of operators that is required is:

$$p, \vee, \neg, \mathbf{EX}, \mathbf{EG}, \mathbf{EU}$$

All other operators can be expressed in terms of these six. The fact that conjunction can be written in terms of disjunction and negation, is well-known: it is a law of De Morgan. Just as conjunction and disjunction are each other's dual, the universal and the existential quantifiers are each other's dual. These dualities can be used to show that in CTL other dualities exist. As an example, we show that **AG** is dual to **EF**:

$$\mathbf{AG}(f) \equiv \neg\mathbf{EF}(\neg f)$$

We first open up the definition of **EF**.

$$\mathbf{AG}(f) \equiv \neg\exists\pi \cdot \exists i \cdot \pi[i] \models \neg f$$

Then, we use the duality between $\forall$ and $\exists$.

$$\mathbf{AG}(f) \equiv \forall\pi \cdot \neg\exists i \cdot \pi[i] \models \neg f$$

Again, we apply that duality.

$$\mathbf{AG}(f) \equiv \forall\pi \cdot \forall i \cdot \neg(\pi[i] \models \neg f)$$

We then apply the second definition in Table 4.1.

$$\mathbf{AG}(f) \equiv \forall\pi \cdot \forall i \cdot (\pi[i] \models \neg\neg f)$$

We apply the rule for double negation:

$$\mathbf{AG}(f) \equiv \forall\pi \cdot \forall i \cdot \pi[i] \models f$$

This corresponds exactly to the definition of **AG** in Table 4.1.

In similar fashion, the following logical equivalences can be proven:

$$f_1 \wedge f_2 \equiv \neg(\neg f_1 \vee \neg f_2) \quad (4.1)$$
$$\forall x.f \equiv \neg\exists x.\neg f \quad (4.2)$$
$$\mathbf{AX}(f) \equiv \neg\mathbf{EX}(\neg f) \quad (4.3)$$
$$\mathbf{AG}(f) \equiv \neg\mathbf{EF}(\neg f) \quad (4.4)$$
$$\mathbf{AF}(f) \equiv \neg\mathbf{EG}(\neg f) \quad (4.5)$$
$$\mathbf{EF}(f) \equiv \mathbf{E}(\text{true } \mathbf{U} f) \quad (4.6)$$
$$\mathbf{A}[f_0 \mathbf{U} f_1] \equiv \neg\mathbf{E}(\neg f_1 \mathbf{U}(\neg f_0 \wedge \neg f_1)) \wedge \mathbf{AF}(f_1) \quad (4.7)$$
$$\mathbf{A}[f_0 \mathbf{R} f_1] \equiv \neg\mathbf{E}[\neg f_0 \mathbf{U} \neg f_1] \quad (4.8)$$
$$\mathbf{E}[f_0 \mathbf{R} f_1] \equiv \neg\mathbf{A}[\neg f_0 \mathbf{U} \neg f_1] \quad (4.9)$$

The first two equations should be evident. The **A** and **E** operators are dual, as are the **G** and **F**. The last two equations state the duality between **R** and **U**.

Equation 4.6 does not formulate a duality, but shows that **EF** can be expressed in terms of **EU**. It simply states that **EF** means that the property "true" must hold until a state is reached where f holds. Equation 4.7 is more involved. Consider Figure 4.3g. For sake of explanation, let p and q be the only atomic propositions. Equation 4.7 expresses that $\mathbf{A}[p\ \mathbf{U}\ q]$ means two things: there should not be path where a "white" state is reached through exclusively states that are white or p. Secondly, it states that always eventually q occurs.

QUESTION 4.7
Prove Equalities 4.3 and 4.8, using the definitions in Table 4.1.

5 Combining Temporal Operators

The syntax of CTL allows nesting of temporal operators. Not all combinations make sense, but some are commonly used:

$\mathbf{AGEF}(p)$: it is always possible to reach a p-state. For example, one might formulate a property that characterizes the initial state and formulate that it is always possible to reach the initial state.

$\mathbf{AGAF}(p)$: property p occurs infinitely often. This is related to the notion of *fairness*, or *starvation-freedom*. If one wants to formulate that a certain process fairly gets the turn to execute, this operator can be used.

$\mathbf{AFAG}(p)$: necessarily p will eventually hold invariably. For example, one might formulate a correctness property that should hold after initialization of a system. Then this property states that eventually but necessarily, the system satisfies the correctness property.

$\mathbf{EFAG}(p)$: possibly p will eventually hold invariably. For example, one might formulate a property that states the system is in an error state and try to see if the system can reach a state from which it is permanently in an error state.

$\mathbf{AG}(p \implies \mathbf{AF}(q))$: Invariably, if p holds then eventually q will hold. This is called *liveness*. Note that liveness is a generalization of fairness: if one chooses $p = \text{true}$, then one obtains $\mathbf{AGAF}(q)$. Liveness is a very important property of a system. We denote it with a special operator:

$$p \rightsquigarrow q \equiv \mathbf{AG}(p \implies \mathbf{AF}(q))$$

Figure 4.4 illustrates liveness. In the left tree, we see six paths, from left to right. The first path begins with p and eventually q holds. Paths 2 to 5 illustrate that whenever p holds, eventually q must hold *for all* emanating execution paths. The sixth path illustrates that liveness is a property that expresses something on all states, not just on the initial state. If p holds in the nth state of a path, then there should be a state after that nth state where q holds, and this for all n.

The right tree in Figure 4.4 also illustrates liveness. The first path has no q state at all. Yet the liveness property holds! Liveness expresses that *if* property p holds then eventually q must hold. If p is never true, then the liveness property holds. We call this *vacuous truth.* A vacuous truth is, for example, "Whenever the traffic light is blue, a smurf crosses the intersection.".

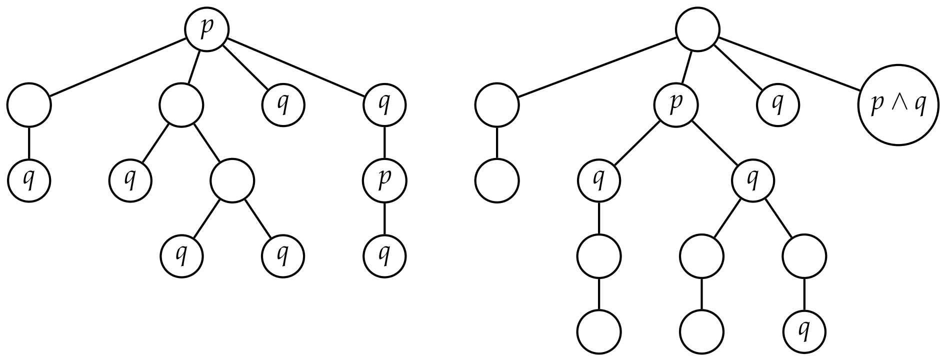


FIGURE 4.4 Illustration of liveness: $p \leadsto q$

Paths 2 to 4 illustrate that once q has occurred after a p, q does not necessarily have to occur again. Some time after each p, a q must happen. Once that has happened, if there are no more p's, then no more q's are necessary.

The fifth path illustrates that liveness does not mean that q has to be preceded by p. Liveness does not mean a causal relationship: it does not formulate that p is the cause of q. Finally, the sixth path illustrates a path where p holds, and q holds immediately. You can check that this also satisfies the definition of liveness.

QUESTION 4.8
Consider the Kripke structure in Figure 4.1. Formulate properties stating that:
a It is always possible to go to s_2.
b It might become impossible to go to s_0.
c Possibly, p is infinitely often true.
d Whenever p occurs, this will eventually be followed by an r.

QUESTION 4.9
Consider some Kripke structure M that satisfies the liveness property $p \leadsto q$. Formulate a CTL property such that if the sytem additionally satisfies your property, the liveness property is not vacuously true.

QUESTION 4.10
Consider Exercise 2.1 of Learning Unit 2. Write a CTL property stating that *whenever* variabele *turn* equals 1, it will remain 1 until process 1 is waiting. Does the model solution satisfy this property?

QUESTION 4.11
Provide a small Kripke structure that shows the difference between $\mathbf{AGAF}(p)$ and $\mathbf{AGEF}(p)$, i.e., a Kripke structure for which exactly one of these CTL formulae is true.

6 Conclusion

This learning unit introduced a temporal logic: CTL. It discussed the operators of this logic, their intuitive semantics as well as their formal definition. The next learning units introduce techniques to prove that Kripke models satisfy CTL formulas.

EXERCISES

1 (Warehouse management system)

Consider software used to manage the warehouse of a company building professional printers. The company has three types of printers. Let's name the three printers P_1, P_2, P_3. For each type of printer, the software keeps track of the amount of this printer type available in the warehouse. For each product, a lower and upper bound on the quantity that should always be available are defined. If the lower bound is reached, the system emits an alarm message "low quantity of P_i". If the upper bound is reached, the system emits an alarm message "high quantity of P_i". The alarm messages are printed on the screen of the computer used to monitor the warehouse.
Some of the informal requirements for the software are:

R1 All queries related to a printer type by a user are answered with either the quantity of printers of that type available in the warehouse or an error message.

R2 If the quantity of a type of printer is always within the bounds, the system does not emit alarm messages.

R3 If the quantity of a type of printer is outside the bounds, an appropriate alarm message is emitted in the "next" state.

Express these three requirements using CTL formulas.

2 (Peterson's algorithm)

Peterson's algorithm satisfies a liveness property. All processes trying to access the critical section will eventually gain access to the critical section.
Express this property in CTL using the notations given in Exercise 2.1 of Learning Unit 2.

FEEDBACK

Answers to questions

4.1 Formula $\mathbf{AX}(\mathit{flag} = 01)$ does not hold in the initial state, since the flag can be set to 10, but formula $\mathbf{EX}(\mathit{flag} = 01)$ does hold, since the flag can be set to 01.

4.2 We can formulate this property, e.g., by referring to the program counters. If we do so, we can formulate this property as

$$\mathbf{AG}(pc_{P_0} \neq l_3 \vee pc_{P_1} \neq l_3)$$

It can be argued that this property is more elegantly formulated in terms of the state variable `turn` and `flag`, but this requires more complex reasoning to figure out the values of the variables when both processes are in the critical section.

4.3 The state formula would be $\mathbf{EF}(\mathit{goat} = \mathit{dead})$. Path $(\mathit{left\ left\ left})$, $(\mathit{left\ left\ boat})$, $(\mathit{left\ dead\ boat})$ satisfies this property as the final state satisfies property $(\mathit{goat} = \mathit{dead})$.

4.4 The property would be: $\mathbf{EF}(\mathit{goat} = \mathit{right} \wedge \mathit{cabbage} = \mathit{right} \wedge \mathit{wolf} = \mathit{right})$. This property holds on the model solution as there exists a path leading to state $(\mathit{right\ right\ right})$.

4.5 This is expressed by the property stating that "*turn* equals 0 until it equals 1". We want that all execution paths satisfy the property. Formally, we have:

$$\mathbf{A}\ (\mathit{turn} = 0\ \mathbf{U}\ \mathit{turn} = 1)$$

4.6 See Figure 4.5.

4.7

$$\mathbf{AX}(f) \equiv \neg\mathbf{EX}(\neg f)$$

We first open up the definition of **EX**.

$$\mathbf{AX}(f) \equiv \neg\exists\pi \cdot \pi[1] \models \neg f$$

Then, we use the duality between $\forall$ and $\exists$.

$$\mathbf{AX}(f) \equiv \forall\pi \cdot \neg(\pi[1] \models \neg f)$$

We then apply the second definition in Table 4.1.

$$\mathbf{AX}(f) \equiv \forall\pi \cdot \pi[1] \models \neg\neg f$$

We apply the rule for double negation:

$$\mathbf{AX}(f) \equiv \forall\pi \cdot \pi[1] \models f$$

This corresponds exactly to the definition of **AX** in Table 4.1.

$$\mathbf{A}[f_0 \mathbf{R} f_1] \equiv \neg\mathbf{E}[\neg f_0 \mathbf{U} \neg f_1]$$

We first open up the definition of **EU**.

$$\mathbf{A}[f_0 \mathbf{R} f_1] \equiv \neg\exists\pi \cdot \exists k \cdot (\pi[k] \models \neg f_1) \wedge (\forall_{i<k} \cdot \pi[i] \models \neg f_0)$$

Then, we use the duality between $\forall$ and $\exists$ twice.

$$\mathbf{A}[f_0 \mathbf{R} f_1] \equiv \forall\pi \cdot \forall k \cdot \neg((\pi[k] \models \neg f_1) \wedge (\forall_{i<k} \cdot \pi[i] \models \neg f_0))$$

We then apply DeMorgan.

$$\mathbf{A}[f_0 \mathbf{R} f_1] \equiv \forall\pi \cdot \forall k \cdot \neg(\pi[k] \models \neg f_1) \vee \neg(\forall_{i<k} \cdot \pi[i] \models \neg f_0)$$

We then apply the second definition in Table 4.1 and apply double negation.

$$\mathbf{A}[f_0 \mathbf{R} f_1] \equiv \forall\pi \cdot \forall k \cdot (\pi[k] \models f_1) \vee \neg(\forall_{i<k} \cdot \pi[i] \models \neg f_0))$$

We apply the rules: $x \implies y \equiv \neg x \vee y$ and $x \vee y \equiv y \vee x$

$$\mathbf{A}[f_0 \mathbf{R} f_1] \equiv \forall\pi \cdot \forall k \cdot (\forall_{i<k} \cdot \pi[i] \models \neg f_0) \implies (\pi[k] \models f_1))$$

This corresponds exactly to the definition of **AR** in Table 4.1.

4.8 a $\mathbf{AGEF}(\neg p \wedge \neg q \wedge r)$

b $\mathbf{EFAG}\neg(p \wedge q \wedge \neg r)$

c This one is not accurately definable in CTL. Formula $\mathbf{EGAF}(p)$ is *not* a solution: this formula is not true for the Kripke structure, even though there does exist a path where p occurs infinitely often. Formula $\mathbf{EGEF}(p)$ is *not* a solution. As a bonus exercise: come up with a small Kripke structure for which this formula is true, even though there is no path where p occurs infinitely often.

d $p \rightsquigarrow r$

4.9 We must formulate that at least once, the premiss p of the liveness property is true. This can be formulated as $\mathbf{EF}(p)$.

4.10 Our property first starts with $\mathbf{A}(turn = 1 \ \mathbf{U}\ f)$, where f expresses that process 1 is waiting. To express that process 1 is waiting: $pc_1 = l_2 \wedge flag[0] = 1 \wedge turn = 0$. Combined, we have:

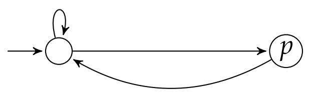
$$\mathbf{A}(turn = 1 \ \mathbf{U}\ pc_1 = l_2 \wedge flag[0] = 1 \wedge turn = 0)$$

However, this property should *always* be true, i.e., whenever variable *turn* equals 1. Thus:

$$\mathbf{AG}(turn = 1 \implies \mathbf{A}(turn = 1 \ \mathbf{U}\ pc_1 = l_2 \wedge flag[0] = 1 \wedge turn = 0))$$

The model does not satisfy the property, since there exists an execution where the property does not hold: a cycle of states where $turn = 1$.

4.11 The following Kripke structure satisfies $\mathbf{AGEF}(p)$ but not $\mathbf{AGAF}(p)$.



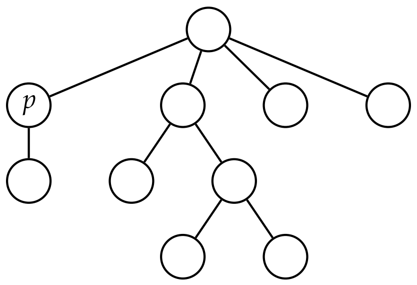
Answers to exercises

1 For each product P_i, there are three formulas of the following form:
R1 $query(P_i) \rightsquigarrow (quantity(P_i) \vee error)$
R2 $\mathbf{AG}(low \leq quantity(P_i) \leq high \implies no\_alarm)$
R3 $\mathbf{AG}((low > quantity(P_i) \vee quantity(P_i) > high) \implies \mathbf{AX}\, alarm)$

2 A process is trying to gain access to the critical section when its program counter is in location l_2. It has access to the critical section when its program counter is in l_3.
The property is $l_2 \rightsquigarrow l_3$.
An additional check is required to ensure that location l_2 is always reached from the initial state.
The property would be $\mathbf{AF}(l_2)$.

Not possible

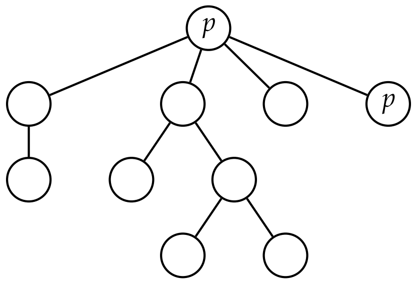
(A) **AX**(p)



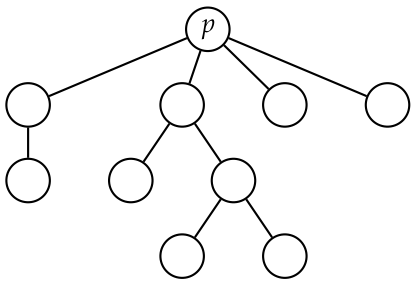
(B) **EX**(p)

Not possible

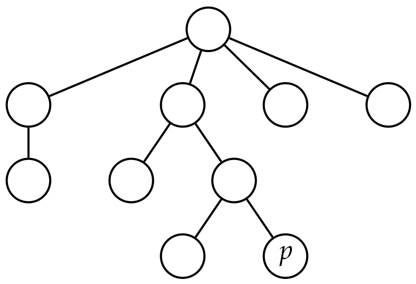
(C) **AG**(p)



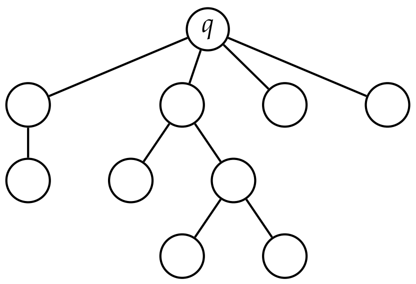
(D) **EG**(p)



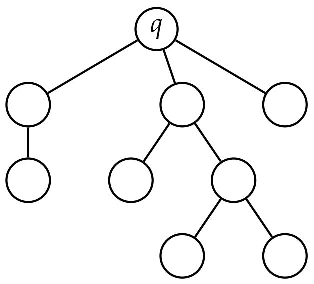
(E) **AF**(p)



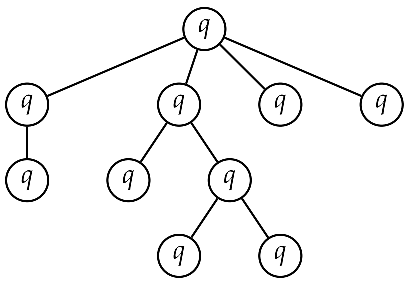
(F) **EF**(p)



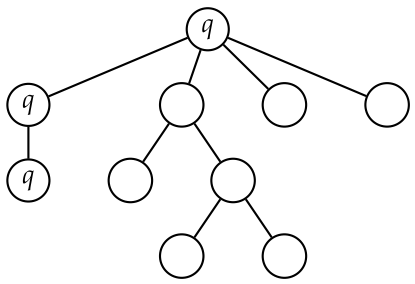
(G) **A**(p **U** q)



(H) **E**(p **U** q)



(I) **A**(p **R** q)



(J) **E**(p **R** q)

FIGURE 4.5 Different illustrations of CTL

Explicit model-checking

Learning unit 5

Explicit model-checking

INTRODUCTION

The previous learning unit introduced a temporal logic used to specify the correct behaviours of systems. This learning unit presents a class of algorithms used to prove that a system represented as a Kripke structure satisfies a temporal property. The goal is to have algorithms that either answer "yes, this property holds for that system" or "no, the property does not hold". In the case the property does not hold, the algorithms give a counter-example to the property. Counter-examples are very useful for debugging but can also be used to generate test cases. The algorithm presented in this learning unit belongs to the class of explicit-state algorithms and is used to prove CTL properties.

LEARNING GOALS
After having studied this learning unit you should be able to:
– understand the basic explicit-state algorithm for CTL
– understand the notion of counter-example
– apply the CTL algorithm on concrete examples.

Reading — Section 4.1 of Chapter 4 of the book "Model Checking" belongs to this learning unit.

Exercises — There are three exercises at the end of this learning unit. You should work on the first two exercises. The third one is optional.

Time — The expected time needed to study this learning unit is about 3 hours.

KERNEL

1 Introduction

Reading — Read the introduction of Chapter 4 of the book "Model Checking".

The model-checking problem is clearly defined at the beginning of Chapter 4. Given a model representing a system and a temporal property, does the model satisfy the property? As quickly sketched in the book, the idea of model-checking is to search for all the states that satisfy a property. The technique presented in this learning unit uses an explicit representation of the state-space. Explicit state exploration techniques have been implemented in popular tools, for instance, SPIN used at NASA.

2 Explicit-state CTL model-checking

Section 4.1 presents an explicit algorithm to solve the model-checking problem for CTL formulas. The term explicit means that the algorithm considers an explicit representation of the Kripke structure as a graph. This algorithm is also referred to as "explicit state". Before reading Section 4.1 we first look at the transition graph depicted in Figure 4.3 on page 39 of the book. This graph shows the possible states of a microwave oven. State 1 is the initial state. It represents the system when the door is open and the microwave is not heating. The usual behaviour is then to close the door and go to state 3. Once the door is closed, a user should push the start button. This moves the microwave to state 6. The microwave will then start warming up (state 7) and then start cooking (state 4). When cooking is done, the door can be opened and the system moves back to the initial state, or the door can be kept closed and the microwave simply turns itself off (state 3). This describes the expected behaviour. From the initial state, it is also possible to reach non-expected behaviours if a user starts the microwave with the door open (state 2). The only way to get back to the expected behaviour is to close the door (state 5) and push the reset button to get back to state 3.

Reading

We suggest that you get familiar with the example. Once you think you understand how the microwave works and its model pictured in Figure 4.3, you should read the first three paragraphs of section 4.1.

The principle of the algorithm is to visit every state and to label every state with the set of formulas that are true in this state (for state s, this set is called $label(s)$). In practice, the algorithm computes the set of states satisfying a formula. For formula f, let us call this set $S(f)$.

Only six cases need to be considered, since in the previous learning unit, we have seen that any CTL formula can be expressed in terms of p, $\neg$, $\vee$, **EX**, **EG**, and **EU**. Note that on page 35 line 10, when describing the procedure for $\vee$ the textbook should mention that states labelled with both f_1 and f_2 are also labelled with $f_1 \vee f_2$.

Consider the formula $(heat \wedge close)$ and the microwave example in Figure 4.3 on page 39. Intuitively, this formula holds in all states where the microwave is heating with a closed door. In order to apply the CTL algorithm, we first convert it to the allowed subset of operators using the rules of double negation and De Morgan. We get $heat \wedge close = \neg(\neg heat \vee \neg close)$. The CTL algorithm first computes the set of states satisfying *heat*, then the set of states satisfying *close*, followed by the sets satisfying their negations. It will then take the union of these sets, and compute the set satisfying the negation of the formula. If we work out the details, we get the following.

$$\begin{aligned}
S(heat) &= \{4,7\}\\
S(close) &= \{3,4,5,6,7\}\\
S(\neg heat) &= S \setminus \{4,7\} = \{1,2,3,5,6\}\\
S(\neg close) &= S \setminus \{3,4,5,6,7\} = \{1,2\}\\
S(\neg heat \vee \neg close) &= S(\neg heat) \cup S(\neg close)\\
&= \{1,2,3,5,6\} \cup \{1,2\} = \{1,2,3,5,6\}\\
S(\neg(\neg heat \vee \neg close)) &= S \setminus \{1,2,3,5,6\} = \{4,7\}
\end{aligned}$$

So, only states 4 and 7 satisfy $heat \wedge close$.

In general, the labelling algorithm starts by labelling states corresponding to the smallest subformulas, working towards the entire formula. Therefore, when computing a set of states for a corresponding formula, we already need to know the set of states corresponding to the subformulas.

QUESTION 5.1
Compute the set of states satisfying the formulas (note that you can express $\implies$ in terms of disjunction and negation):

a $(error \wedge heat)$
b $\neg(heat \wedge close)$
c $(error \implies \neg heat)$

Consider the formula $\mathbf{EX}(heat \wedge close)$. To check it, we first need to transform the formula to $\mathbf{EX}(\neg(\neg heat \vee \neg close))$. The CTL algorithm than first computes the labelling of the smallest subformulas, until we find the labelling $S(\neg(\neg heat \vee \neg close)) = \{4,7\}$ as computed before. Then, a state satisfies $\mathbf{EX}(\neg(\neg heat \vee \neg close))$ if it has a successor satisfying $\neg(\neg heat \vee \neg close)$. In other words, the formula $\mathbf{EX}(\neg(\neg heat \vee \neg close))$ is satisfied by all states that are a predecessor of some state in $S(\neg(\neg heat \vee \neg close))$. We have $S(\mathbf{EX}(heat \wedge close)) = S(\neg(\neg heat \vee \neg close)) = \{4,7,6\}$.

QUESTION 5.2
Compute the set of states satisfying the following formulas:

a $\mathbf{EX}((error \implies \neg heat) \wedge error)$
b $\mathbf{EX}(error \wedge heat)$

Dualities between the CTL operators imply that checking for the six cases mentioned above is enough to check any CTL property. This means, for instance, that there is a procedure to check **EX** but there is no need for a procedure to check **AX**.

From the explanation so far, and the previous to questions, it becomes clear that disallowing $\wedge$ from the operators, and needing to remove it using double negation and De Morgan's law is cumbersome. Therefore, when computing a solution by hand, we allow the direct computation of the set of states satisfying $S(f \wedge g)$, when $S(f)$ and $S(g)$ are known. The set of states can be computed as $S(f \wedge g) = S(f) \cap S(g)$, the intersection of the set of states satisfying f and the set of states satisfying g.

Note that all other operators will need to be removed before applying the model checking algorithm.

QUESTION 5.3
Compute $S(\mathbf{AX}(heat \wedge close))$.

Reading

Read the fourth paragraph of Section 4.1.

The fourth paragraph of Section 4.1 gives an informal description of checking the until operator. To check $\mathbf{E}(f\ \mathbf{U}\ g)$, one first finds all states satisfying f and all states satisfying g. This computes the sets $S(f)$ and $S(g)$. Subsequently, for every state s in $S(g)$, the algorithm computes set of states that satisfy f and from which s can be reached by passing only states that satisfy f. This is a form of backwards reachability.

To do this, the algorithm first labels all states in $S(g)$ with $\mathbf{E}(f\ \mathbf{U}\ g)$, since they immediately satisfy the property. The algorithm maintains a todo-list T, which is initialised with all states in $S(g)$. As long as there are still states in the todo-list, one state s is taken from the set, for all predecessors of s, if it was not yet labelled $\mathbf{E}(f\ \mathbf{U}\ g)$ *and* it is in $S(f)$, then it is labelled $\mathbf{E}(f\ \mathbf{U}\ g)$, and it is added to the todo-list.

Let us consider the property $\mathbf{EF}(\neg start \wedge close \wedge heat)$. Remember that $\mathbf{EF}\ p$ is an abbreviation for $\mathbf{E}(True\mathbf{U}\ p)$. Hence, the property is equivalent to $\mathbf{E}(True\ \mathbf{U}\ \neg start \wedge close \wedge heat)$. This means that there exists a path towards a state where the microwave is actually cooking. If the initial state satisfies this property then it is possible to actually use the microwave.

First, the algorithm computes $S(\neg start)$, $S(close)$, and $S(heat)$. Then, it computes the intersection of these three sets. Only one state is in this intersection, namely, state 4. From state 4 the algorithm works backwards and finds all states satisfying *True* and reaching state 4. It first finds 7 and 4, then 6, then 3, then 5 and 1, and finally 2. So, the property is true in states $\{1,2,3,4,5,6,7\}$, that is, in every state. From any state, it is possible to reach the state where the microwave is actually cooking.

Reading

Read the fifth paragraph of Section 4.1, and look at Figure 4.1 showing the procedure to check $\mathbf{E}(f\ \mathbf{U}\ g)$.

Can you match the informal computations described above to executions of the procedure in Figure 4.1? We advise you to run this procedure on the examples discussed so far.

QUESTION 5.4
For the microwave oven example, compute the set of states satisfying the formula $\mathbf{E}(start\ \mathbf{U}\ heat)$.

Reading

Read pages 36 to 40 and consider the procedure described in Figure 4.2. You can skip the proof of Lemma 1 but the conditions in Lemma 1 are important.

As explained in the textbook, checking $\mathbf{EG}(f)$ involves a search for strongly connected components (SCCs). The algorithm is justified by Lemma 1. The first step of the procedure restricts the set of states to those satisfying f. This corresponds to the first condition of Lemma 1. To satisfy $\mathbf{EG}(f)$, that is "there exists a path where f holds globally", a state must satisfy f. The rest of the algorithm corresponds to the second condition of Lemma 1.

QUESTION 5.5
For the microwave oven example, compute the set of states satisfying the formula $\mathbf{EG}(\mathit{close})$.

Theorem 1 on page 38 gives the time complexity of checking an arbitrary formula. This complexity is polynomial, which is rather efficient. The catch is that the number of states is often extremely large. The number of states often grows exponentially with the size of the system to verify. The state-explosion problem is a major challenge to the large-scale application of model-checking techniques. Nevertheless, model-checking is routinely applied by the hardware industry and is becoming more popular in software validation.

3 Conclusion

This learning unit presented an explicit-state algorithm to prove CTL formulas over Kripke structures. Each Kripke structure is explicitly represented as a transition graph. This graph is explored exhaustively to prove properties. The graph exploration algorithm is efficient but the size of the Kripke structures often grows exponentially with the size of the system under validation.

EXERCISES

1 (Peterson's algorithm)

Consider the Kripke model of Peterson's algorithm given in Learning Unit 2. Consider the property $\mathbf{AG}(\neg l_3 l_3)$. (Note that $l_3 l_3$ is the full label that we are checking for.) Describe a run of the CTL explicit-state algorithm for this property and Peterson's model.

2 (The Wolf Goat Cabbage puzzle)

Consider the Kripke model of the solution to that puzzle proposed in Learning Unit 2. Consider the property specifying that there exists a path leading to a state where all participants have crossed the river: $\mathbf{EF}\ (goat = right \land cabbage = right \land wolf = right)$. Describe a run of the CTL explicit-state algorithm for this property and the solution's model.

3 (The Wolf Goat Cabbage puzzle (again and optional))

This exercise is optional. You can do it if time allows, otherwise you might directly look at the solution.
Consider the Kripke model of the solution to that puzzle proposed in Learning Unit 2. Consider the property specifying that all paths lead to a state where eventually all participants have crossed the river, that is, $\mathbf{AF}\ (goat = right \land cabbage = right \land wolf = right)$. Explain how the CTL algorithm shows that this property does not hold on the model. What is the reason why the property does not hold ?

FEEDBACK

Answers to questions

5.1 The answers are:

a Observe $error \wedge heat = \neg(\neg error \vee \neg heat$. We have

$$\begin{aligned}
S(error) &= \{2,5\}\\
S(heat) &= \{4,7\}\\
S(\neg error) &= \{1,3,4,6,7\}\\
S(\neg heat) &= \{1,2,3,5,6\}\\
S(\neg error \vee \neg heat) &= S(\neg error) \cup S(\neg heat)\\
&= \{1,3,4,6,7\} \cup \{1,2,3,5,6\} = S
\end{aligned}$$

$$S(\neg error \vee \neg heat) = \emptyset$$

Hence, we obtain $S(error \wedge heat) = \emptyset$.

b $\{1,2,3,5,6\}$. In the learning unit, we already know the solution for $heat \wedge close$, which is $\{4,7\}$. The solution to $\neg(heat \wedge close)$ is the complement of this set, that is, all the other states. Thus, the answer is $\{1,2,3,5,6\}$.

c We have that $error \implies \neg heat$ is an abbreviation for $\neg error \vee \neg heat$. Then, we have

$$\begin{aligned}
S(error) &= \{2,5\}\\
S(heat) &= \{4,7\}\\
S(\neg error) &= \{1,3,4,6,7\}\\
S(\neg heat) &= \{1,2,3,5,6\}\\
S(\neg error \vee \neg heat) &= S(\neg error) \cup S(\neg heat)\\
&= \{1,3,4,6,7\} \cup \{1,2,3,5,6\} = S
\end{aligned}$$

Therefore, we obtain $S(error \implies \neg heat) = S = \{1,2,3,4,5,6,7\}$.

5.2 The answers are:

a First observe that $(error \implies \neg heat) \wedge error = \neg(\neg(\neg error \vee \neg heat) \vee \neg error)$. In the last answer of the previous question, we computed $S(error \implies \neg heat) = S(\neg error \vee \neg heat) = \{1,2,3,4,5,6,7\}$. We moveover have $S(error) = \{2,5\}$. We now get

$$\begin{aligned}
S(\neg(\neg error \vee \neg heat)) &= S \setminus S(\neg error \vee \neg heat)\\
&= S \setminus \{1,2,3,4,5,6,7\} = \emptyset\\
S(\neg error) &= S \setminus S(error)\\
&= S \setminus \{2,5\} = \{1,3,4,6,7\}\\
S(\neg(\neg error \vee \neg heat) \vee \neg error) &= S(\neg(\neg error \vee \neg heat)) \cup S(\neg error)\\
&= \{1,3,4,6,7\}\\
S(\neg(\neg(\neg error \vee \neg heat) \vee \neg error)) &= S \setminus \{1,3,4,6,7\} = \{2,5\}\\
S(\mathbf{EX}(\neg(\neg(\neg error \vee \neg heat) \vee \neg error))) &= \{1,2,5\}
\end{aligned}$$

Note that at this last step we have included all predecessors of the states 2 and 5. Hence, $\mathbf{EX}((error \implies \neg heat) \wedge error) = \{1,2,5\}$.

b From the previous question we computed that $S(error \wedge heat) = \emptyset$. The set of predecessors from $\emptyset$ is empty, thus the answer is the empty set, that is, $S(\mathbf{EX}(error \wedge heat)) = \emptyset$.

5.3 We use the duality between **AX** and **EX**, to get

$$\mathbf{AX}(heat \wedge close) = \neg\mathbf{EX}(\neg(heat \wedge close)).$$

In Question 5.1 we already computed $S(\neg(heat \wedge close)) = \{1,2,3,5,6\}$. To compute $S(\mathbf{EX}(\neg(heat \wedge close)))$ we simply determine all predecessors of $\{1,2,3,5,6\}$. We find that state 1 has predecessors 3 and 4, state 2 has predecessors 1 and 5, state 3 has predecessors 1, 4 and 5, state 5 has predecessor 2 and state 6 has predecessor 3. Thus, $S(\mathbf{EX}(\neg(heat \wedge close))) = \{1,2,3,4,5\}$.
Finally, $S(\neg\mathbf{EX}(\neg(heat \wedge close))) = S \setminus \{1,2,3,4,5\} = \{6,7\}$. So, the answer is $\{6,7\}$.

5.4 We first compute the sets of states satisfying the atomic propositions:

$$S(start) = \{2,5,6,7\}$$
$$S(heat) = \{4,7\}$$

Next, we label states 4 and 7, with $\mathbf{E}(start\ \mathbf{U}\ heat)$, and perform the backward reachability algorithm. The predecessors of state 4 are states 4 and 7, which were already labelled, so they do not need to be considered further. The predecessor of 7 is 6, which is labelled *start*, so state 6 is labelled $\mathbf{E}(start\ \mathbf{U}\ heat)$ and added to the todo list. Next, the predecessor of state 6 is state 3, which is not labelled *start*, so it is not labelled, and not added to the todo-list. The todo-list is now empty, and the algorithm is done. Hence we have computed $S(\mathbf{E}(start\ \mathbf{U}\ heat)) = \{4,7,6\}$.

5.5 We first compute the set of state $S(close) = \{3,4,5,6,7\}$. For computing $S(\mathbf{EG}(close))$, we only consider the part of the Kripke structure containing those states. Within this subgraph, we first compute the *nontrivial* strongly connected components. In this case, there are two SCCs: $\{5\}$ and $\{3,4,6,7\}$. Since there are no transitions within the first SCC, this is trivial. Now, observe that since $\{3,4,5,6\}$ are within a non-trivial SCC, all states have a path that globally satisfies *close*, by simply choosing a transition that stays within the SCC. All these states are labelled $\mathbf{EG}(close)$ and added to the todo-list. We again perform a backward reachability by considering the predecessors of the states in the todo-list (but requiring that the predecessors are in the subgraph!), and labelling states that were note yet labelled and adding those states to the todo-list. In this case, state 3 has predecessor states 4 and 5, the latter state is labelled $\mathbf{EG}(close)$ and added to the todo-list (note: state 3 also has predecessor 1, but this is outside the graph). For states 4, 6, and 7, the only predecessors are already labelled. For state 5, the only predecessor, 2, is outside the subgraph and is therefore not considered.
The todo-list is now empty, hence we have computed $S(\mathbf{EG}(close)) = \{3,4,5,6,7\}$.

Answers to exercises

1 The algorithm does not know how to check **AG** properties. The first operation is to consider the dual of the initial claim, that is, the algorithm will check $\neg(\mathbf{EF}\ l_3 l_3)$. In fact, the algorithm will check the property $\neg\mathbf{E}(\mathit{True}\ \mathbf{U}\ l_3 l_3)$.
The first line of the procedure is to search for all states satisfying the formula $l_3 l_3$. There is no such state, so after one traversal of the state graph, the algorithm for checking $l_3 l_3$ terminates. The set of states satisfying $\mathbf{E}(\mathit{True}\ \mathbf{U}\ l_3 l_3)$ is therefore empty. The negation of this property is therefore satisfied in all states. Property $\mathbf{AG}(\neg l_3 l_3)$ holds.

2 Formally, property $\mathbf{EF}(\mathit{goat} = \mathit{right} \wedge \mathit{cabbage} = \mathit{right} \wedge \mathit{wolf} = \mathit{right})$ is an abbreviation for property $\mathbf{E}(\mathit{True}\ \mathbf{U}\ (\mathit{goat} = \mathit{right} \wedge \mathit{cabbage} = \mathit{right} \wedge \mathit{wolf} = \mathit{right}))$.
The first step of the procedure is to search for all states satisfying the state formula $\mathit{goat} = \mathit{right} \wedge \mathit{cabbage} = \mathit{right} \wedge \mathit{wolf} = \mathit{right}$. There is one such state which constitutes the only element in set T of the procedure in Figure 4.1.
In the while loop, the procedure starts from this state and adds all predecessors satisfying *True*, so it just keep adding predecessor states until it reaches a point where no new states are visited. At each step, set T contains all predecessor states that still need to be explored. The procedure terminates when no additional predecessor states can be added. When this happens, set T is empty.

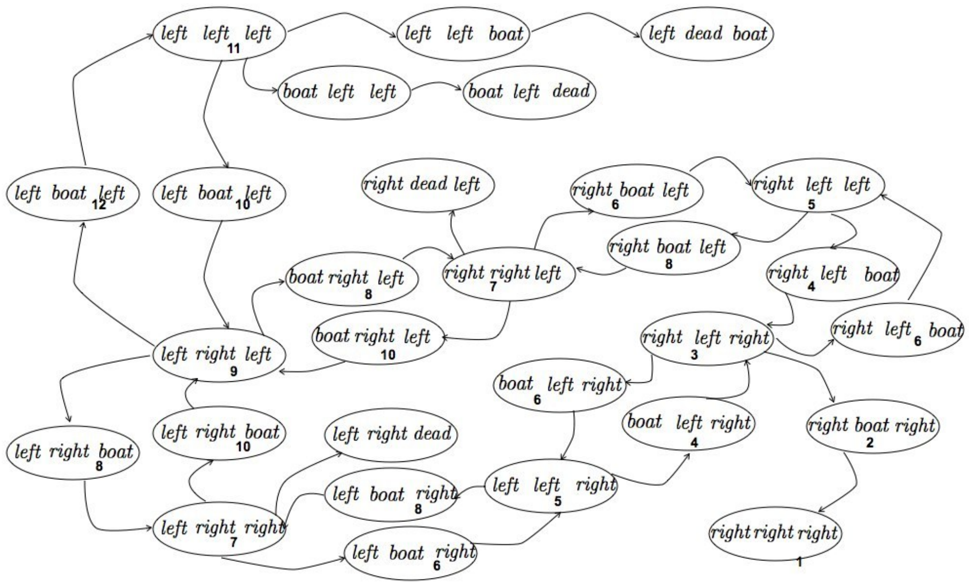


FIGURE 5.1 Unrolling of the algorithm.

Figure 5.1 shows the unrolling of the while loop. Each state is numbered with the step in loop where it is visited. As the initial has been labelled, the model satisfies the property.

3 The reason why the property does not hold is that it is possible to always keep loading a participant on the boat, to cross the river, bring the same participant back to the other side, and do that again and again. To check **AF**, the first operation will be to consider the dual and try to prove ¬**EG** which means to first check for **EG**. The procedure will look for strongly connected components where at least one participant is not on the right side. One such SCC is

$$\{(\mathit{left}, \mathit{right}, \mathit{left}), (\mathit{boat}, \mathit{right}, \mathit{left}), (\mathit{right}, \mathit{right}, \mathit{left}), (\mathit{boat}, \mathit{right}, \mathit{left})\}$$

This means that there exists an infinite execution where the model stays in this SCC. As the negation of the initial property holds, the original property does not hold.

Symbolic model-checking

Learning unit 6

Symbolic model-checking

INTRODUCTION

The previous learning unit introduced an explicit state algorithm to prove CTL formulas. Kripke structures were represented as an explicit state graph. In this learning unit, we present an alternative algorithm where Kripke structures are symbolically represented using Binary Decision Diagrams. This learning unit presents an algorithm for CTL model-checking using this symbolic representation. This learning unit focuses on Binary Decision Diagrams, fixpoints, and counter-examples and witnesses.

LEARNING GOALS

After having studied this learning unit you should be able to:

– create and manipulate Binary Decision Diagrams
– understand the basic symbolic state algorithm for CTL
– apply the symbolic state algorithm for CTL
– understand the notions of counter-examples and witnesses.

Reading — Chapters 5 and 6 of the book "Model Checking" belong to this learning unit.

Exercises — You should do the exercises at the end of the learning unit.

Time — The expected time needed to study this learning unit is about 6 hours.

KERNEL

1 Introduction

Chapter 5 presents a symbolic representation of Kripke structures and Chapter 6 introduces a CTL model-checking algorithm based on this symbolic encoding. The objective of the symbolic representation is to be more efficient than an explicit one. In practice, this is not always the case. Symbolic and explicit techniques are both used.

Reading — Read the introduction of Chapter 5 on page 51 of the textbook “Model Checking”.

2 Binary decision diagrams

Reading — Read Section 5.1 of Chapter 5 of the book "Model Checking".

Chapter 5 introduces Binary Decision Diagrams (BDDs) as a symbolic representation of transition systems. BDDs are important in model-checking and enable efficient verification of large systems.

The fundamental notion underlying this section is Shannon's expansion. For a Boolean function f and a given variable x of f, it states that function f can be decomposed into two sub-functions considering the cases where either x is false or x is true. Formally, we have the following equation:

$$f = (\neg x \wedge f|_{x=0}) \vee (x \wedge f|_{x=1})$$

Using Shannon's decomposition, it is possible to build a binary decision tree. Figure 5.1 shows the tree for a two-bit comparator, that is, for the following Boolean function:

$$f(a_1, a_2, b_1, b_2) = (a_1 \leftrightarrow b_1) \wedge (a_2 \leftrightarrow b_2)$$

We propose to consider another, simpler example:

$$f(a, b, c) = a \vee (b \wedge c)$$

We use Shannon's expansion to build a Binary Decision Tree for this Boolean function. To apply the expansion we need to determine an ordering of the variables. Let us consider the ordering a, b, c.

We first decompose according to a, this gives the following equation:

$$f(a, b, c) = (\neg a \wedge f(0, b, c)) \vee (a \wedge f(1, b, c))$$

The first part of the tree from this equation is pictured in Figure 6.1.

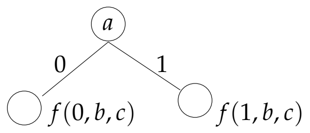


FIGURE 6.1 First part of the binary decision tree.

We can recursively proceed with decomposing each sub-formula. For instance, we can expand the subtree $f(0, b, c)$ and obtain:

$$f(0, b, c) = (\neg b \wedge f(0, 0, c)) \vee (b \wedge f(0, 1, c))$$

A similar expansion is obtained for $f(1, b, c)$. We then insert these terms in the tree and obtain the tree pictured in Figure 6.2.

We proceed with the four leafs and then compute the values of f where all literals have been assigned a definite value. We obtain the Binary Decision Tree (BDT) pictured in Figure 6.3.

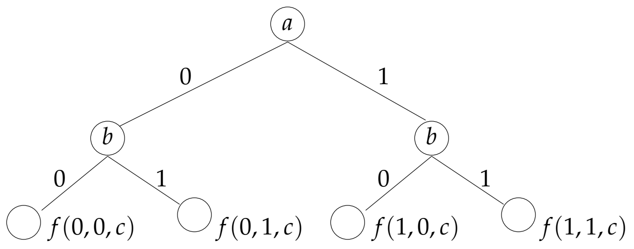


FIGURE 6.2 The binary decision tree continued ...

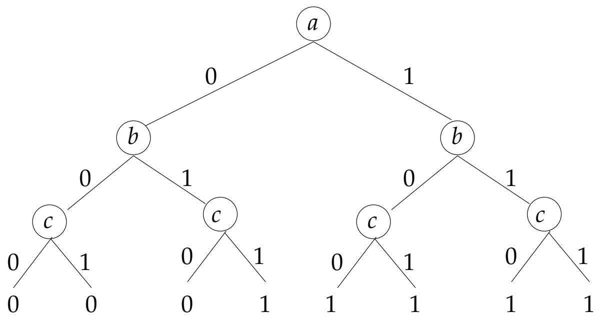


FIGURE 6.3 The binary decision tree

We apply the rules proposed by Bryant to transform this BDT into a Reduced Ordered BDD. We first apply the rule "remove redundant tests"[1], this means that we eliminate all nodes where the value at the end of the left and the right branches is the same. There are three cases where this applies. We obtain the graph in Figure 6.4.

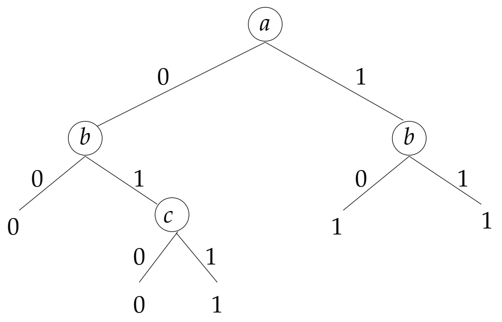


FIGURE 6.4 Reducing the BDT with “remove redundant tests”.

The same rule applies once more to the true branch of a, removing the b node in the right subtree. After applying this rule, we apply the rule “remove duplicate terminals” and obtain the ROBDD in Figure 6.5.

QUESTION 6.1
Let f be the formula $f = (a \wedge c) \vee (\neg a \wedge b)$.

Provide an ROBDD for formula f, under variable ordering a, b, c.

The size of a ROBDD depends on the variable ordering. The following question illustrates this point.

[1]The order in which the rules are applied does not matter.

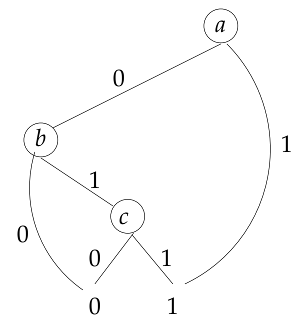


FIGURE 6.5 Reducing the BDT to a ROBDD.

QUESTION 6.2
For which ordering(s) is the size of the ROBDD for our function f maximal? (Note: The size is counted in the number of nodes and edges)?

Finding an optimal ordering is infeasible and there exist functions with no optimal ordering, like bitwise multiplication. In such cases, explicit algorithms might perform better than symbolic ones.

3 Operations on ROBDDs

The main advantage of ROBDDs is that they can be manipulated efficiently. In particular, checking that two ROBDDs are equal requires constant time. Negation is also very easy and simply requires to negate the two leaves of a ROBDD.[2]

Other operations are performed using the Apply algorithm described on pages 56 and 57 of the textbook "Model Checking". We illustrate Apply on the following example:

$$f(a) \vee f'(b,c)$$

where $f(a) = a$ and $f'(b,c) = b \wedge c$.

We consider the ordering a, b, c. We assume we have the ROBDDs of each function, that is, the graphs depicted in Figure 6.6.

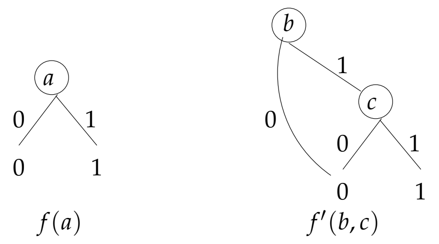


FIGURE 6.6 ROBDDs for f and f'.

[2]Note that any ROBDD has only two leaves, namely, 1 for *true* and 0 for *false*.

We now use Apply and Shannon's expansion to compute the disjunction of these two ROBDDs. Since the roots of the ROBDDs are not terminal vertices and since the variables in the two roots are not the same we need to determine which of the two variables comes first in the ordering. Since we consider the ordering a, b, c, we have $a < b$ (that is, a comes before b). We apply the formula from the third case mentioned on page 56:

$$f(a) \vee f'(b,c) = (\neg a \wedge (f(0) \vee f'(b,c))) \vee (a \wedge (f(1) \vee f'(b,c)))$$

We now start to build a new ROBDD as illustrated in Figure 6.7.

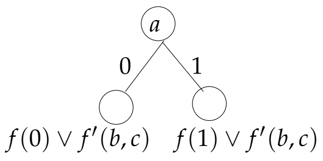


FIGURE 6.7 Partial ROBDDs for $f \vee f'$.

Now, we have $f(0) = 0$ and $f(1) = 1$ (since $f(a) = a$). Therefore, the left branch simplifies to $f'(b,c)$ and the right branch to 1. We already have a ROBDD for $f'(b,c)$. We simply insert it as the left branch. The result is not yet an ROBDD: it contains two terminals for 1, so we still need to remove the duplicate terminals to obtain an ROBDD. Finally, we obtain the ROBDD pictured in Figure 6.5 for the formula:

$$f(a,b,c) = a \vee (b \wedge c)$$

4 Transition systems and ROBDDs

Reading

Read section 5.2 of Chapter 5 of the textbook "Model Checking".

The most interesting part of this section is the example about representing transition systems using ROBDDs. As we introduced it in Learning Unit 2, it is possible to encode the set of states and the transition relation of a Kripke structure as a first order formula. This formula can be seen as a Boolean function that can be encoded as a ROBDD. This is the essential aspect of symbolic model-checking: to manipulate ROBDDs instead of the explicit state graph.

As an example, consider the Kripke structure shown in Figure 6.8. Since

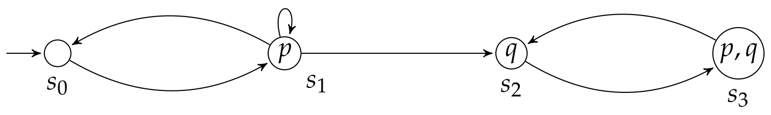


FIGURE 6.8 Kripke structure

each state in the figure is uniquely labelled by the atomic propositions, we can directly use the atomic propositions in our first order encoding, and hence also in our (RO)BDD encoding of the Kripke structure.

The ROBDD encoding for the entire set of states is simply the ROBDD for 1, since every combination of p and q is in the set of states S. However, we can also represent subsets of the sets of states as an ROBDD. Consider, for example, the set of states $\{s_0, s_1, s_3\}$. When we represent this set as a BDD, we consider the function that returns *true* (or 1) for every state that is in this set.

Let us consider variable ordering p, q. As an intermediate step in constructing the ROBDD we can construct the BDT, which is shown in Figure 6.9.

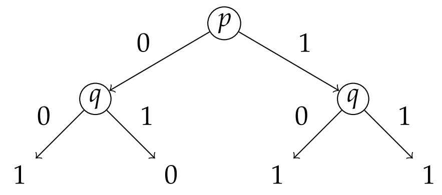


FIGURE 6.9 BDT for the set of states $\{s_0, s_1, s_3\}$ of the Kripke structure in Figure 6.8 under variable ordering $p < q$

From this, we can remove redundant tests and duplicate terminals to obtain the ROBDD shown in Figure 6.10.

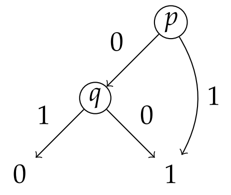


FIGURE 6.10 ROBDD for the set of states $\{s_0, s_1, s_3\}$ of the Kripke structure in Figure 6.8 under variable ordering $p < q$

The other essential part of representing a Kripke structure using BDDs is representing the transition relation. Recall that in the first-order encoding we used primed variables, for instance p' to represent the value in the new state. This same idea is used in the BDD encoding. For the transition relation of the Kripke structure in Figure 6.8 we therefore get a BDD with variables p, q, p', and q'. Consider the variable ordering $p < q < p' < q'$. We can again first construct a binary decision tree, which is shown in Figure 6.11.

When we apply the rules to order the BDD, we obtain the ROBDD given in Figure 6.12.

Note that, in general, for small Kripke structures, we give an OBDD (not necessarily reduced) directly.

QUESTION 6.3
Reconsider the Kripke structure of the oven example from Figure 1.3, for which you have given the first-order encoding in Question 2.2. Give the OBDD for the set of *reachable* states in this Kripke structure.

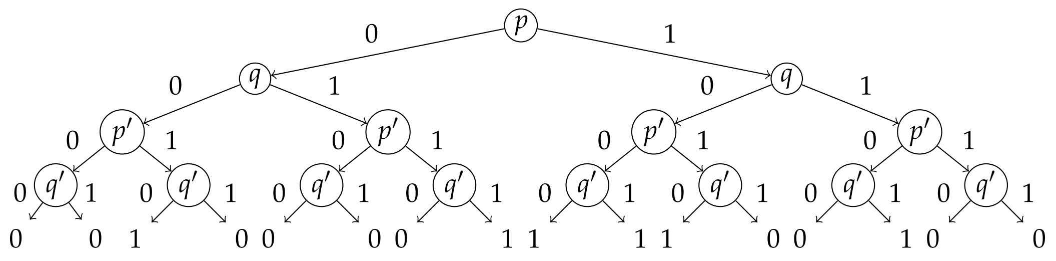


FIGURE 6.11 BDT for the transition relation of the Kripke structure in Figure 6.8

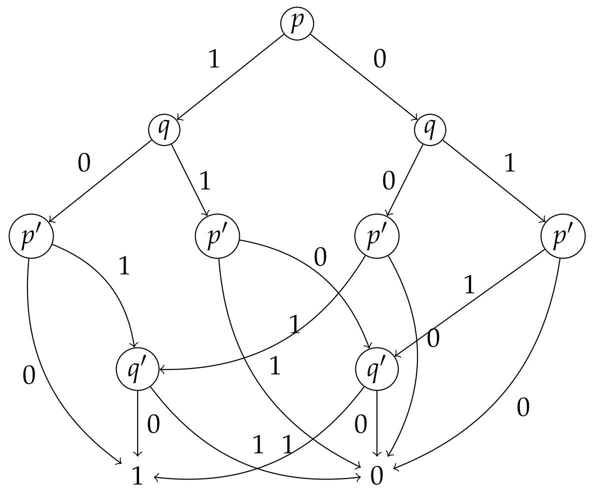


FIGURE 6.12 ROBDD for the transition relation of the Kripke structure in Figure 6.8 using variable order p, q, p', q'.

QUESTION 6.4
Reconsider the Kripke structure of the oven example from Figure 1.3, for which you have given the first-order encoding in Question 2.2. Give the OBDD for the transition relation of this Kripke structure.

5 Fixpoints and CTL

An important concept in symbolic model-checking is the notion of fixpoints. The basic idea behind fixpoints is to recursively apply a function to a set of elements until two applications of the function return the same result, that is, the same set. We have reached a limit and this limit is exactly the fixpoint.

Reading

Read the introduction of Chapter 6 and Section 6.1. Do not read all details and proofs of all lemmas. You should first have a quick read to get an impression of the material. Then, you should read the workbook while keeping an eye on the textbook.

The important content of this section is the relation between CTL operators and fixpoints. This relation is given on page 63 where all basic CTL operators are defined in terms of fixpoints. Some operators are expressed as least- or greatest fixpoints of predicate transformers. A predicate transformer is a function that transforms subsets of the set of states S into subsets of S. For instance, $\tau(Z) = f_2 \vee (f_1 \wedge \mathbf{EX}\ Z)$ is a predicate transformer that extracts from set Z states that either satisfy f_2 or satisfy f_1 and have successors in Z. This corresponds exactly to states satisfying $\mathbf{E}(f_1\ \mathbf{U}\ f_2)$. This predicate has reached a fixpoint when two successive applications do not transform Z anymore. For instance, we will see that $\mathbf{E}(f_1\ \mathbf{U}\ f_2)$ is the least fixpoint of the predicate transformer τ defined as above.

5.1 LEAST FIXPOINT COMPUTATION

Figure 6.1 on page 62 gives a procedure to compute the least fixpoint of a predicate transformer τ. As we can see from the procedure, the least fixpoint is the limit of the following increasing sequence:

$$\mathit{False} \subseteq \tau(\mathit{False}) \subseteq \tau(\tau(\mathit{False})) \subseteq \tau(\tau(\tau(\mathit{False}))) \ ...$$

where *False* is the boolean formula representing the empty set.

Figure 6.13 illustrates this computation. From the empty set, new elements are added using the predicate transformer until a point is reached where no new elements are introduced by applying the predicate transformer.

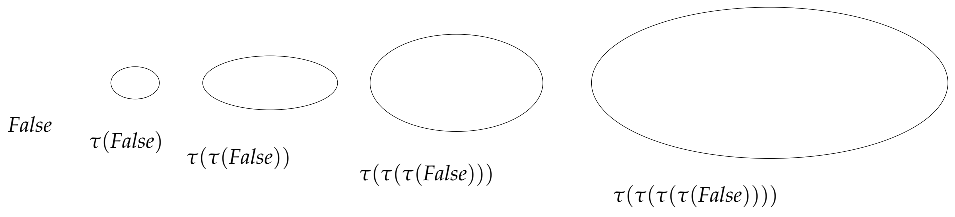

False
$\tau(\mathit{False})$
$\tau(\tau(\mathit{False}))$
$\tau(\tau(\tau(\mathit{False})))$
$\tau(\tau(\tau(\tau(\mathit{False}))))$

FIGURE 6.13 Least fixpoint computation.

QUESTION 6.5
An example at the bottom of page 65 and the top of page 66 of the textbook shows the computation of **EU** using the least fixpoint.

Using symbolic-model checking, compute the set of states satisfying $\mathbf{E}\ (p\ \mathbf{U}\ q)$ for the Kripke structure in Figure 6.3 on page 65. Only the initial state s_0 is named, assume the other states are numbered s_1, s_2, s_3 in clockwise order. Show all the different steps of the computation using fixpoints of the set of states.

Recall that p and q are not the names of the states, but atomic propositions (hence, boolean formulas) that for each state can be true or false.

5.2 GREATEST FIXPOINT COMPUTATION

Figure 6.2 on page 63 gives a procedure to compute the greatest fixpoint of a predicate transformer. The greatest fixpoint is the limit of the following decreasing sequence:

$$...\ \tau(\tau(\tau(\mathit{True}))) \subseteq \tau(\tau(\mathit{True})) \subseteq \tau(\mathit{True}) \subseteq \mathit{True}$$

where *True* is the boolean formula representing the entire set S.

Figure 6.14 illustrates this computation. From the largest set, elements are removed using the predicate transformer until no more elements are removed by applying the predicate transformer.

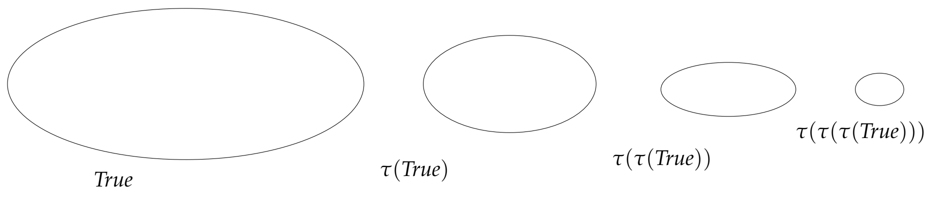


FIGURE 6.14 Greatest fixpoint computation.

QUESTION 6.6
Apply the greatest fixpoint computation for the property $\mathbf{EG}(p \vee q)$ to the example in Figure 6.3 on page 65.

6 Symbolic model checking for CTL

Reading

Read Section 6.2 starting on page 66.

Quantified Boolean formulas are introduced in Section 6.2.1 and used in Section 6.2.2 to encode the CheckEX procedure. The end of Section 6.2.2 shows how to use CheckEX to compute the other two basic CTL operators **EU** and **EG**.

Quantified Boolean formulas simply state that a formula f with a variable x must be true for one of the possible values of x or for all values of x. Vectors of variables are used to quantify over more than one variable.

We now give a bit more details about the **EX** operator and its encoding as a QBF. Given a set of states satisfying predicate P, one can compute the set of states that can reach P with one application of the transition function R. Figure 6.15 illustrates the backward computation.

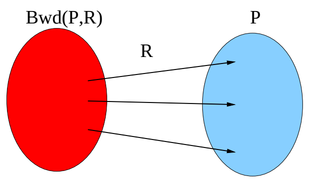


FIGURE 6.15 Backward computation.

Formally, the backward computation is defined as follows:

$$Bwd(P,R) = \{\overline{v} | \exists \overline{v}' : \overline{v}' \in P \wedge (\overline{v}, \overline{v}') \in R\}$$

This can be encoded as a QBF and then as a ROBDD:

$$f(\overline{v}) = \exists \overline{v}' : (P(\overline{v}) \wedge T(\overline{v}, \overline{v}')$$

The backward operation exactly corresponds to the **EX** operator. The states satisfying $(\mathbf{EX}\ p)$ are the states with a successor state satisfying p. Starting from all states satisfying p, one computes all states satisfying $(\mathbf{EX}\ p)$ by applying the backward computation once.

This gives a symbolic encoding of the main operator for CTL model-checking. The remainder of the section encodes the CheckEU and CheckEG procedures as fixpoints. They both use the basic CheckEX operation.

QUESTION 6.7
For the microwave oven example shown in Figure 4.3 on page 39 of the textbook, for each of the following formulas apply the algorithm for symbolic model checking, by showing the evoluation of the set Z for each step of the algorithm. You do not have to give the actual BDDs, but you can give the sets of states represented by the BDDs.
a $\mathbf{E}(\mathit{start}\ \mathbf{U}\ \mathit{heat})$
b $\mathbf{EG}(\mathit{close})$

7 Counter-examples and witnesses

Reading

You can read Section 6.4 (you can skip section 6.3).

Counterexamples and witnesses constitute an essential feature of model-checking. The capability of exhibiting witnesses and counterexamples is crucial in the analysis of designs to find and correct errors.

Section 6.4 gives details about how counter-examples are generated. In contrast, we come back to the intuition behind witnesses and counter-examples.

Witnesses are naturally associated to "existential" formulas, i.e., **EX**,**EG**, and **EU**. The intuition is that $(\mathbf{EX}\ p)$ is true in state s if there exists a successor state satisfying property p. A witness is simply a path starting from s with one successor state satisfying p. Formally, a witness for $(\mathbf{EX}\ p)$ in state s_0 is a path $\pi = s_0, s_1$ such that $s_1 \models p$. Figure 6.16 illustrates such a path.

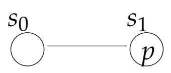


FIGURE 6.16 A witness for **EX**.

Witnesses to **EU** formulas are similar. For instance, a witness for the formula $\mathbf{E}(p\ \mathbf{U}\ q)$ would be a path with a prefix where states satisfy p and the final state of the path satisfies q. Formally, a witness for $\mathbf{E}(p\ \mathbf{U}\ q)$ in state s_0 is a path $\pi = \pi_0, s_k$ such that $\forall j, \pi_0^j \models p$ and $s_k \models q$. Figure 6.17 illustrates such a path.

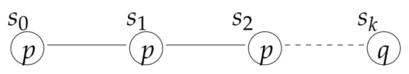


FIGURE 6.17 A witness for **EU**.

Witnesses for **EG** have a specific form, named a *lasso*. Formula (**EG** p) is true if there exists an execution path where p is true in all states. The idea is that such a path must have a cycle and all states of the path must satisfy p. Formally, a witness path for (**EG** p) is a path $\pi = \pi_0, s_k, s_i$ such that $\forall j, \pi_0^j \models p$ and $s_k \models p$ and $s_i \in \pi_0$. Figure 6.18 illustrates such a path.

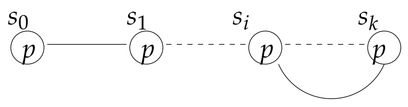


FIGURE 6.18 A witness for **EG**.

A witness of an existentially quantified formula is a counter-example to the dual universally quantified formula.

QUESTION 6.8
Show for which type of formulas the witnesses for **EX** and **EG** are counter-examples. A counter-example is the dual of a witness. It explains why a property does not hold.

8 An ALU example

Reading

To conclude this learning unit, read section 6.5 starting page 75 of the textbook. This section presents a concrete example, namely a simple ALU.

This concludes this learning unit about symbolic model-checking.

EXERCISES

1 (Majority function)

Consider the majority function defined as follows:

$$MAJ(b_0, b_1, b_2, b_3, b_4) = \begin{cases} 1 & \text{if } b_0 + b_1 + b_2 + b_3 + b_4 \geq 3 \\ 0 & \text{otherwise} \end{cases}$$

Depict a ROBDD representing this function. Is it possible to find a variable ordering giving a smaller ROBDD ?

2 (Symbolic encoding of Kripke structures)

Consider the Kripke structure pictured in Figure 5.4 on page 58 of the textbook. The authors give the Boolean function encoding the Kripke structure:

$$(a \wedge b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge b')$$

Depict a ROBDD encoding these transitions.

FEEDBACK

Answers to questions

6.1 We first decompose according to a, this gives the following equation:

$$f(a,b,c) = (\neg a \wedge f(0,b,c)) \vee (a \wedge f(1,b,c))$$

We can recursively proceed with decomposing each sub-formula. When we expand the subtree $f(0,b,c)$ we obtain:

$$f(0,b,c) = (\neg b \wedge f(0,0,c)) \vee (b \wedge f(0,1,c))$$

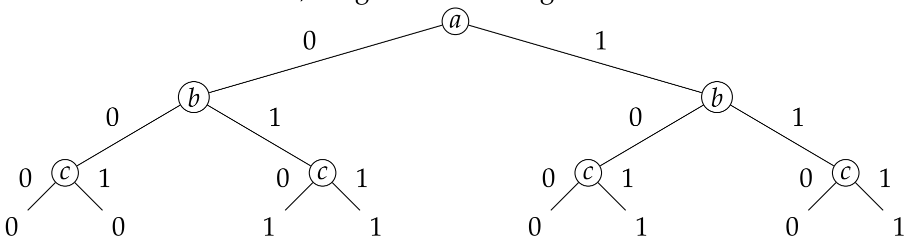
When we expand the subtree $f(1,b,c)$ we obtain:

$$f(1,b,c) = (\neg b \wedge f(1,0,c)) \vee (b \wedge f(1,1,c))$$

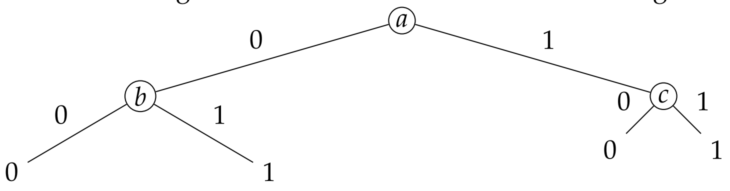
Finally we proceed with the leafs, and then compute the values of f where all literals have been assigned a definite value.

$$\begin{aligned}
f(0,0,c) &= (\neg c \wedge f(0,0,0)) \vee (c \wedge f(0,0,1)) \\
f(0,1,c) &= (\neg c \wedge f(0,1,0)) \vee (c \wedge (f(0,1,1)) \\
f(1,0,c) &= (\neg c \wedge f(1,0,0)) \vee (c \wedge f(1,0,1)) \\
f(1,1,c) &= (\neg c \wedge f(1,1,0)) \vee (c \wedge f(1,1,1)) \\
f(0,0,0) &= 0 \\
f(0,0,1) &= 0 \\
f(0,1,0) &= 1 \\
f(0,1,1) &= 1 \\
f(1,0,0) &= 0 \\
f(1,0,1) &= 1 \\
f(1,1,0) &= 0 \\
f(1,1,1) &= 1
\end{aligned}$$

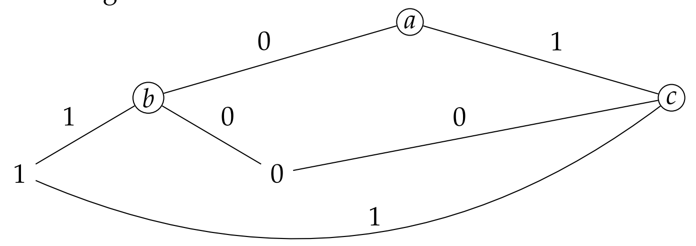
When we combine all this, we get the following BDT.



To simplify this BDT, and obtain the ROBDD, we first remove redundant tests. Observe that we can apply this to the two left-hand c-nodes, as well as to the right-hand b-node. We obtain the following BDT.



Finally we apply the rule to remove duplicate terminals and obtain the following ROBDD.



6.2 The orderings giving the largest BDD are *b*,*a*,*c* and *c*,*a*,*b*. To discover this we apply Shannon's expansion for different orderings. In the workbook, we used the ordering a,b,c. We now perform the decomposition according to the ordering *b*,*a*,*c*.
We first decompose according to *b*, this gives the following equation:

$$f(a,b,c) = (\neg b \wedge f(a,0,c)) \vee (b \wedge f(a,1,c))$$

We can build the first part of the tree from this equation as pictured in Figure 6.19.

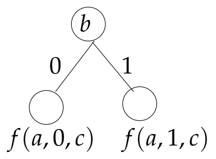


FIGURE 6.19 First part of the tree.

We now apply Shannon's decomposition for *a* in each sub-formulas. We obtain the tree in Figure 6.20.

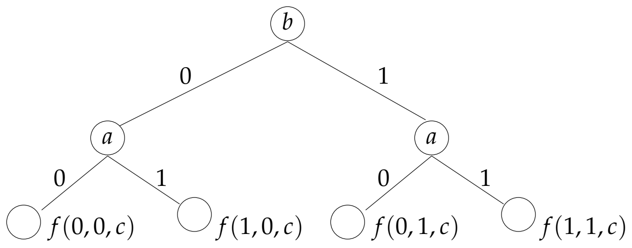


FIGURE 6.20 Second part of the tree.

We would then proceed with the four leafs and then compute the values of *f* where all literals have been assigned a definite value. We would obtain the Binary Decision Tree in Figure 6.21.
We first apply the rules proposed by Bryant to transform this BDT into a Reduced Ordered BDD. We first apply the rule "remove redundant tests", this means that we eliminate all nodes where the value is the same at the end of the left and the right branches. There are three cases where this applies. We obtain the graph pictured in Figure 6.22.
After applying this rule, we apply the rule "remove duplicate terminals" and obtain the ROBDD in Figure 6.23.

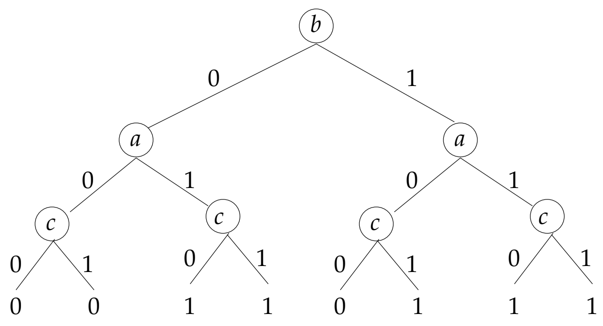


FIGURE 6.21 Last part of the tree.

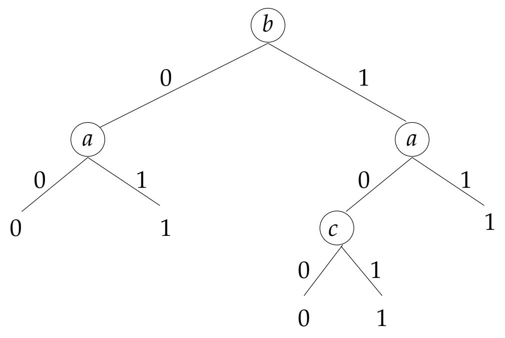


FIGURE 6.22 First reduction towards a ROBDD.

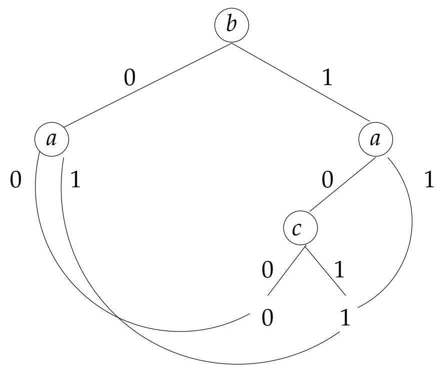
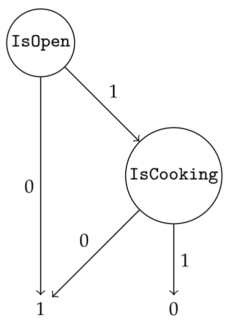


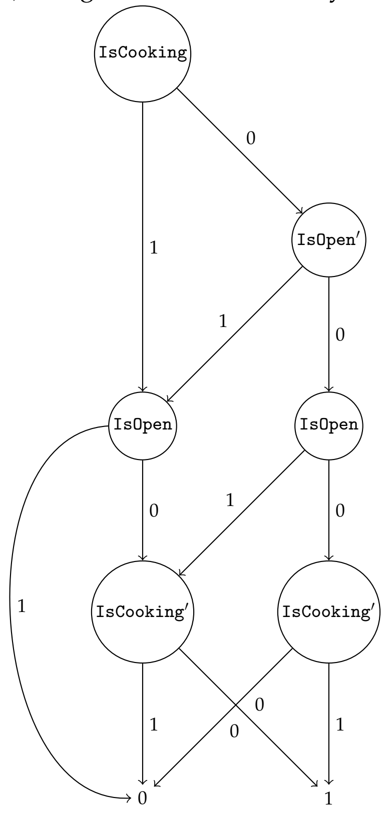
FIGURE 6.23 Reduction to a ROBDD.

We see that this ROBDD is larger than the one obtained with the ordering *a*,*b*,*c*. It has 4 nodes and 8 edges. With the ordering *a*,*b*,*c*, the ROBDD has 3 nodes and 6 edges.
A similar ROBDD would be obtained for the ordering c,a,b.

6.3 The reachable states in the Kripke structure are all, except the one with labelling $\neg$`IsOpen` $\wedge$ $\neg$`IsCooking`.
For the variable ordering `IsOpen`, `IsCooking`, the OBDD is as follows.



6.4 We give the OBDD under variable ordering , IsCooking, IsOpen′, IsOpen, IsCooking′, since this gives the smallest OBDD (other orders are fine as well). We give the OBDD directly.



6.5 To check property $\mathbf{E}\ (p\ \mathbf{U}\ q)$ using symbolic model-checking, we translate it into an equivalent property involving a predicate transformer. We will compute the least fixpoint of the predicate transformer defined as $\tau(Z) = q \vee (p \wedge \mathbf{EX}\ Z)$. We start with an empty set and will add states satisfying q or satisfying p and with successors in Z.
As initially Z is empty, i.e., $Z_0 = \emptyset$, there is no state satisfying p with successors in the empty set. There is one state satisfying q. Let us name this state s_2 and add it to Z. We have:

$$Z_1 = \tau^1(\mathit{false}) = \{s_2\}$$

At the next step, there is no new state satisfying q. There is one state satisfying p and that has state s_2 as successor. This is a state satisfying p with a successor in Z_1. Let us name that state s_1 and add it to Z. Now, we have

$$Z_2 = \tau^2(\mathit{false}) = \{s_1, s_2\}$$

State s_0 satisfies p and has a successor in the set $\{s_1, s_2\}$, namely s_1. It is therefore added to Z. Now, we have

$$Z_3 = \tau^3(\mathit{false}) = \{s_0, s_1, s_2\}$$

It is now not possible to add new states to Z, i.e., $Z_4 = Z_3$. The least fixpoint has been reached and the computation stops. The set of states satisfying $\mathbf{E}\ (p\ \mathbf{U}\ q)$ is $\{s_0, s_1, s_2\}$.

6.6 Let us first number the states, starting from the initial state, with s_0, s_1, s_2, s_3. The objective is to check $\mathbf{EG}(p \vee q)$. We have to compute the greatest fixpoint of the predicate transformer $\tau(Z) = (p \vee q) \wedge\ \mathbf{EX}\ Z$, where Z is initially the set of all states, that is, $Z_0 = \{s_0, s_1, s_2, s_3\}$.
The computation of the greatest fixpoint removes those states that do not satisfy $p \vee q$, and those states that do not have a successor in Z_0, that is, $Z_1 = Z_0 \setminus \{s_3\}$.
Applying the predicate transformer on Z_1 is $Z_2 = \tau(Z_1) = (p \vee q) \wedge \mathbf{EX}(Z_1) = \{s_0, s_1, s_2\} \cap \{s_0, s_1, s_2, s_3\} = Z_1$. A fixpoint is reached and Z_1 is the set of states satisfying $\mathbf{EG}(p \vee q)$. Indeed, this path enters the loop s_1, s_2 where $p \vee q$ always holds.

6.7 a Observe that $\mathbf{E}(\mathit{start}\ \mathbf{U}\ \mathit{heat}) = \mu Z.\mathit{heat} \vee (\mathit{start} \wedge \mathbf{EX}(Z))$. The predicate transformer that we need to use is $\tau(Z) = \mathit{heat} \vee (\mathit{start} \wedge \mathbf{EX}(Z))$, and we compute its least fixpoint. The evolution of Z is the following.

$$\begin{aligned} Z_0 &= \tau^0(Z) = \mathit{False} \\ Z_1 &= \tau^1(Z) = \tau(\mathit{False}) = \{4, 7\} \\ Z_2 &= \{4, 6, 7\} \\ Z_3 &= \{4, 6, 7\} = Z_2 \end{aligned}$$

So, the computation is stable, and the property is satisfied in the states 4, 6, and 7.

b Observe that $\mathbf{EG}(\mathit{close}) = \nu Z.\mathit{close} \wedge \mathbf{EX}(Z)$. The predicate transformer is therefore $\tau(Z) = \mathit{close} \wedge \mathbf{EX}(Z)$, and we compute its greatest fixpoint. The evolution of Z is the following.

$$\begin{aligned} Z_0 &= \tau^0(Z) = \mathit{True} \\ Z_1 &= \tau^1(Z) = \tau(\mathit{True}) = \{3, 4, 5, 6, 7\} \\ Z_2 &= \tau^2(Z) = \tau(Z_1) = \{3, 4, 5, 6, 7\} = Z_1 \end{aligned}$$

So, the computation is stable, and the property is satisfied in states 3, 4, 5, 6, and 7. Note that computing $\tau(Z_1) = \mathit{close} \wedge \mathbf{EX}(Z_1)$ involves computing *Bwd* for Z_1 with the transition relation of the Kripke structure, and effectively computes all predecessors of states in Z_1.

6.8 The witness for $\mathbf{EX}(p)$ is a counter-example to $\mathbf{AX}(\neg p)$. The latter holds if all successor states does not satisfy p. To invalidate this property it is enough that only one of these states does satisfy p.
The witness for $\mathbf{EG}(p)$ is a counter-example to $\mathbf{AF}(\neg p)$. The latter means that on all paths there is eventually a state where p does not holds. The existence of one path always satisfying p is enough to invalidate this property.

Answers to exercises

1 A solution with variable ordering b_0, b_1, b_2, b_3, b_4 is given in Figure 6.24. Note that all solutions have the same structure independently of the variable ordering.

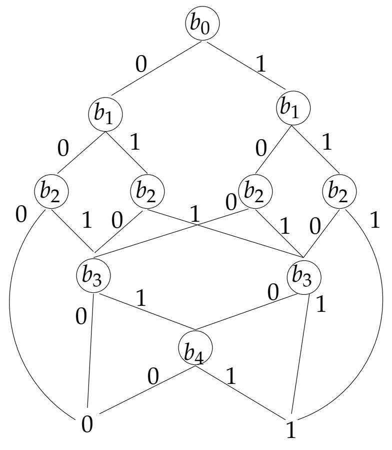


FIGURE 6.24 Solution to exercise 1.

2 We first build the BDDs for each transition. The result is given in Figure 6.25.

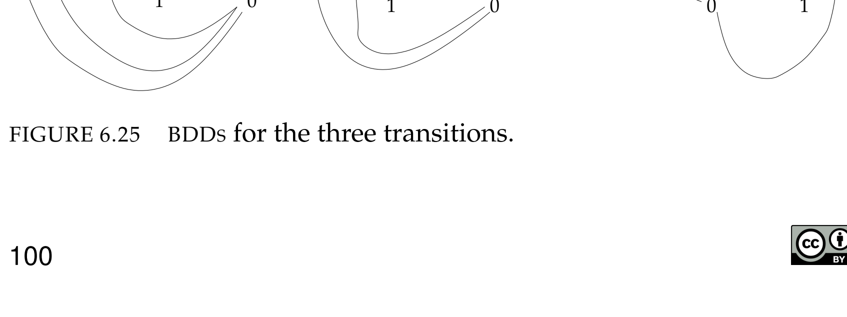


FIGURE 6.25 BDDs for the three transitions.

Let us name these ROBDDs as t_1, t_2, amd t_3. We then first compute the disjunction of the first two transitions (t_1 and t_2) using Apply. Procedure Apply basically takes the disjunction between nodes. Figure 6.26 shows the result and its reduction to a ROBDD. We now detail how this result is obtained.
We follow the procedure described on page 56 of the textbook. Node a is the root of t_1 and t_2. We use Shannon expansion which means that we create a new BDD representing $t_1 \vee t_2$ and starting with node a. If a is false, then t_1 and t_2 lead to 0. Therefore, the left branch of $t_1 \vee t_2$ also leads to 0. If a is true then the results depend on node b. At node b, the false branch of t_1 leads to false, meaning that the result of $t_1 \vee t_2$ depends on the left branch of t_2. We simply insert the sub-BDD of t_2 starting with a' in the BDD for $t_1 \vee t_2$. Similarly, the right branch of t_2 leads to 0 and we insert the sub-BDD of t_1 starting with a'. Now, in the BDD of $t_1 \vee t_2$ the two branches lead to the same sub-BDD. Node b is irrelevant. It can be removed and only one copy of the sub-BDD starting with a' is kept. The computation stops.

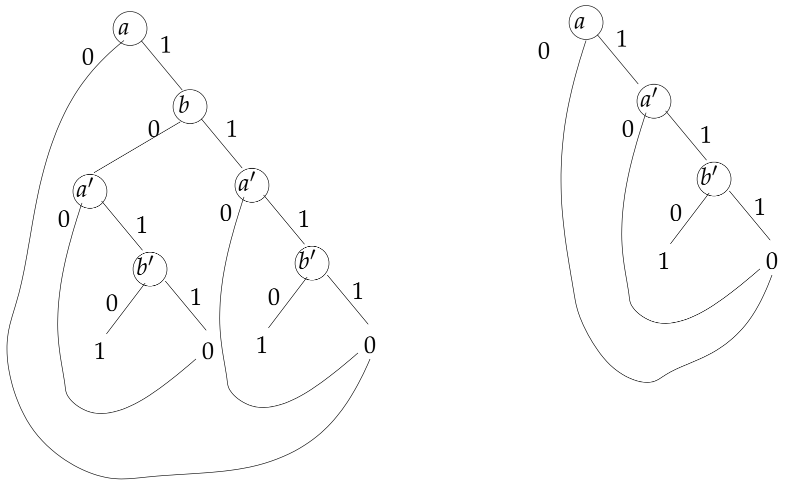


FIGURE 6.26 BDDs for the first disjunct and its reduction to a ROBDD.

Finally, we compute the disjunction with the last transition. Figure 6.27 shows the result and its reduction to a ROBDD. The delicate part is that one ROBDD has variable b and the other not. When taking the disjunction, the branch when b is true is leads to 0 and therefore the result only depends on the a' branch of the first ROBDD.

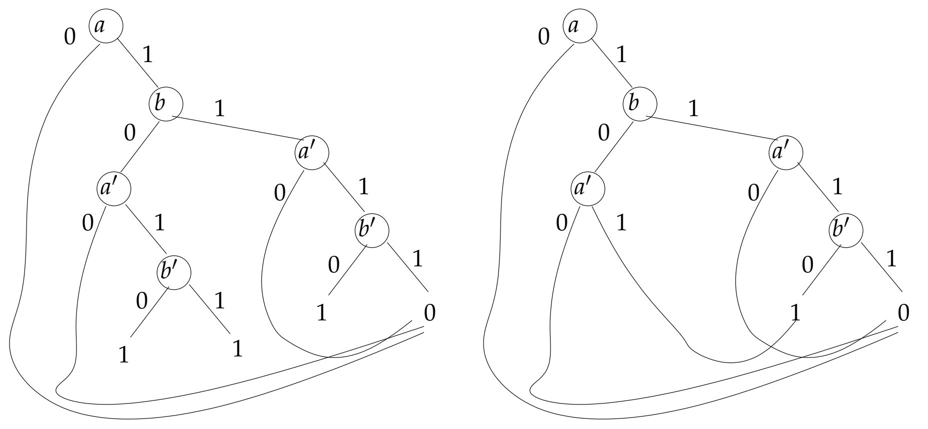


FIGURE 6.27 BDDs for the final result and its reduction to a ROBDD.

Block 3

Formal testing

Model-based testing

Learning unit 7

Model-based testing

INTRODUCTION

The third block of SVT is about testing. This learning unit starts this block with a short introduction of the principles of model-based testing with labelled transition systems.

LEARNING GOALS
After having studied this learning unit you should be able to:
– give an informal description of model-based testing
– informally describe the main components of the model-based testing framework

Reading — Section 2 of the paper "Model-based testing with labelled transition systems" by Tretmans belongs to this learning unit.

Exercises — There are five small exercises at the end of this learning unit. You should work on all of them.

Time — The expected time needed to study this learning unit is about 1 hour.

KERNEL

1 General testing approach

Reading — Read the first paragraph of Section 2 of the paper "Model-based testing with labelled transition systems" by Tretmans.

The first paragraph gives an overview of the general principles of model-based testing (MBT). This overview is shown in Figure 1. In the remainder of this learning unit, we follow the explanation of this general approach.

2 Implementation

Reading — Read the second paragraph of Section 2 of the paper "Model-based testing with labelled transition systems" by Tretmans.

The second paragraph deals with implementations. There are already several important aspects.

Implementations Under Test (IUTs) are real, physical objects that "one can hit with a hammer". In block II, model-checking was applied to models. The objective was to model the expected behaviour of a system and check that this behaviour exhibits desired properties. Testing is performed on a running implementation, not on a model.

IUTs are considered black-box systems. The only way to interact with them is to send them inputs and look at their responses (outputs) or at the absence of responses. Interaction is always performed via the interfaces.

3 Specification

Reading

Read the third paragraph of Section 2 of the paper "Model-based testing with labelled transition systems" by Tretmans.

The third paragraph deals with specifications. Specifications describe the expected behaviour of the IUT. The objective of model-based testing is to check whether the behaviour of the IUT conforms to the specification. Specifications are written in a formal language. In our case, we will consider specifications written as labelled transition systems (LTS). LTSs are introduced in learning unit 8.

4 Conformance

Reading

Read the fourth paragraph of Section 2 of the paper "Model-based testing with labelled transition systems" by Tretmans.

The fourth paragraph deals with conformance. Conformance is a central notion in model-based testing. It formally defines when an IUT conforms to the specification. The most important concept introduced in the paragraph is the test hypothesis or test assumption. Model-based testing aims at checking real systems, not their models. Another objective is to provide a formal testing framework. Later in the course, we will prove a theorem stating that an IUT is conforming to its specification if and only if the IUT passes all tests. This proof cannot involve the IUT itself but a model thereof. The formal test hypothesis - introduced later in the course - will add the assumption that the formal model of an IUT passes a test if and only if the real IUT passes a test.

5 Testing

Reading

Read the fifth paragraph of Section 2 of the paper "Model-based testing with labelled transition systems" by Tretmans.

The fifth paragraph deals with testing. The process of testing is to send inputs to the IUT and observe the resulting outputs. A test case is composed of inputs and outputs. Test cases are also described using a formal language. Executing a test case results in a verdict, either pass or fail. A test suite is a set of test cases.

6 Conformance testing

Reading

Read the sixth paragraph of Section 2 of the paper "Model-based testing with labelled transition systems" by Tretmans.

The sixth paragraph defines conformance testing. Conformance is defined between a model of the IUT and the specification. This model is conforming to the specification if and only if it passes all tests that can possibly be generated by the specification. The paragraph introduces an important notion: completeness. This means that testing can detect all possible wrong implementations of a given specification. In practice this is not possible. The issue is that complete test suites are infinite. Completeness is decomposed in two weaker properties: soundness and exhaustiveness. Each notion corresponds to one direction of Equation (1) on page 6 of the paper "Model-based testing with labelled transition systems" by Tretmans.

Soundness is the left-to-right implication. If an implementation is conforming then it passes all tests. In the contrapositive form, this means that if an implementation fails a test then it is not conforming. Soundness expresses the fact that testing detects only 'real' errors. It shows that there are no false negatives.

Exhaustiveness is the right-to-left implication, that is, if an implementation passes all tests then it is conforming to the specification. In the contrapositive form, this means that if an implementation is not conforming then there exists a test case in the test suite that will result in a verdict *fail*. This really means that for all non-conforming implementations the test suite contains a test detecting this error. There are often infinitely many incorrect implementations of a specification and only infinite test suites contain a test for all possible errors.

Exhaustiveness mainly is of theoretical interest. Soundness can be used to prove a test generation algorithm correct. One goal of testing is to detect errors. A sound testing algorithm only detects 'true' errors.

7 Test generation

Reading

Read the seventh paragraph of Section 2 of the paper "Model-based testing with labelled transition systems" by Tretmans.

Test generation is formally defined as a function taking as input a specification and producing a test suite. It is important to note that test generation only depends on the specification. It does not depend on the IUT.

8 Conclusion

Reading

Read the end of Section 2 of the paper "Model-based testing with labelled transition systems" by Tretmans.

Model-based testing is based on the following ingredients:
– a formal specification language
– a domain model for implementations

– an implementation - or conformance - relation
– a test execution procedure
– a test generation algorithm
– the test hypothesis or the test assumption

In the remainder of this block, we present an instantiation of this general framework where specifications, implementations, and test cases are defined as labelled transition systems.

EXERCISES

1 One goal of model-based testing is to test whether a formal model of an implementation is conforming to a formal specification. True or false? Justify your answer.

2 The generation of test cases depends on the IUT. True or false? Justify your answer.

3 To apply model-based testing, one must first discharge the test hypothesis. This means that one must build a precise model of the IUT. True or false? Justify your answer.

4 Model-based testing can prove the absence of errors. True or false? Justify your answer.

5 A test suite is sound if and only if it can detect all possible incorrect implementations. True or false? Justify your answer.

FEEDBACK

Answers to exercises

1 False. The main objective of model-based testing is to test if a real system conforms to its specification.

2 False. Test generation only depends on the specification.

3 False. The test hypothesis can in practice never be discharged. It is also not necessary. Model-based testing is a black-box technique and does not require a model of the IUT. The test hypothesis only is necessary to prove the theoretical correctness of the test generation procedure.

4 False. Like any testing technique, model-based testing can in practice only detect errors. In theory, a complete – so often infinite – test suite could prove the absence of error. This might require an infinite amount of time!

5 False. The question defines exhaustiveness. A test suite is sound if and only if it only detects true errors.

Labelled transition systems

Learning unit 8

Labelled transition systems

INTRODUCTION

This learning unit introduces the notion of labelled transition systems (LTSs). LTSs are fundamental models in computer science. They constitute the theoretical foundations of the testing theory presented in the third block.

LEARNING GOALS
After having studied this learning unit you should be able to:
– understand the formal definition of LTSs
– understand the main operators of LTSs
– understand the intuition behind quiescence
– understand the formal definition of quiescence
– understand the notions of inputs and outputs

Reading

Parts of Section 3 of the paper "Model-based testing with labelled transition systems" belong to this learning unit. Sections 3.2 and 3.6 are not part of this course.

Exercises

There is one exercise about the specification of an electronic passport at the end of this learning unit. You should do this exercise. A solution is given at the end of the learning unit.

Time

The expected time needed to study this learning unit is about 4 hours.

KERNEL

1 Introduction

Reading

Read the introduction of Section 3 of the paper "Model-based testing with labelled transition systems".

As written in this introduction, labelled transition systems (LTSs) constitute the foundational formalism for the model-based testing theory presented in this course. LTSs are also a classical class of models commonly used in computer science. The main difference between Kripke structures and LTSs is that the former focuses on states while the latter focuses on observable actions. The purpose of Kripke structures is to explore a state space to prove or disprove temporal formulas. The purpose of LTSs (in this course) is to support a formal approach to testing actual systems. Systems under tests (SUTs) are considered black boxes. One can observe their actions but their internal structure – in particular their state – is not visible.

2 Labelled transition systems

Reading

Read Section 3.1 of the paper "Model-based testing with labelled transition systems".

The main usage of LTSs is to describe the visible behaviour of a system. Labels are used to identify which actions can be performed in which states. A system may perform internal actions that cannot be observed by the system's environment. An internal transition is always represented by the greek letter τ.

The section in the textbook introduces definitions and notations about LTSs. We come back to a few important ones.

N.B.: In Definitions 3 and 4 a state q_0 is mentioned inside an existential quantifier. This state is not the initial state. It simply is a state named q_0. In the paragraph just above Definition 5 Tretmans writes "we will not always distinguish between a transition system and its initial state". In Definition 5 p simply denotes a state of a transition system. It is a bit confusing as in Definitions 5.7 to 5.11, p seems to be a transition system. All definitions are about a state and a transition system satisfies the definition if its initial state satisfies the definition.

2.1 NON-DETERMINISM VS. CHOICE

Non-determinism is an important feature of LTSs. It is possible to describe systems that can reach more than one state after performing an action. System r in Figure 2 on page 8 is such an example. After pushing a button the system either reaches a state from where it can generate liquorice or it reaches a state from where it can wait for another push on the button.

System q in the same Figure is completely deterministic. It is never possible to go from one state to two or more different states by performing the same action. On the other hand, after pushing a button it can either produce liquorice or chocolate. In practice and because of this choice between actions, system q might be considered a non-deterministic system. Formally, it does not satisfy the definition of non-determinism (Definition 5.9 on page 10). Non-determinism and choice are different notions. A choice usually is between two different actions, whereas non-determinism gives two different executions for the same action.

2.2 STRONGLY CONVERGING AND IMAGE FINITE

An important notion of LTSs is the **after**-set. This set denotes the set of states reachable after performing a sequence of visible actions. This set is computed using the **after**-operator. Note that the **after**-operator is defined using $\overset{\sigma}{\Longrightarrow}$, hence the **after**-set is computed taking into account the internal τ-transitions. For example, in Figure 2 on page 8, v_0 **after** $but = \{v_0, v_1\}$ due to the τ-transition $v_1 \xrightarrow{\tau} v_0$. The **after**-set of a set of states consists of the union of the **after**-sets of the initial states, for example, in Figure 2 $\{r_0, r_2\}$ **after** $but = \{r_1, r_2, r_4\}$.

QUESTION 8.1
Consider the LTSs in Figure 2 on page 8 of "Model-based testing with labelled transition systems".

Give the following sets:
– $traces(q)$
– q **after** $\{but\}$
– u **after** $\{but \cdot but\}$
– v **after** $\{but \cdot but\}$

If a system is not strongly convergent, there might be an infinite sequence of invisible actions and the computation of the **after**-set will not terminate. Similarly, if infinitely many states can be reached after an action, the computation of the **after**-set will also not terminate. For these reasons, we will always consider LTSs that are strongly converging and image finite. Figure 3 and Example 3 on page 10 illustrate these notions.

QUESTION 8.2
Is it possible for an LTS to be image finite and not finite state?

Another important notion related to the **after**-set is the notion of a refusal characterised by the predicate **refuses**. The idea is to identify states where some actions are not possible. Given a set of labels L, a deadlock state s - that is, a state without outgoing transitions - is identified by s **refuses** L. In general, for a set of states P, and set of labels A, P **refuses** A whenever there is some state in P that cannot perform any of the actions in A, and that also cannot perform a τ-transition. For example, in Figure 2, $\{r_1, r_2\}$ **refuses** $\{liq\}$ since r_2 cannot perform a liq action or a τ-action. Note that in the same figure, v_1 does not refuse but, since v_1 can perform a τ-transition. In general for a state s, if s has an outgoing τ-transition, $\neg(\{s\}$ **refuses** $A)$ for all sets of actions A.

QUESTION 8.3
Consider the LTSs in Figure 2 on page 8 of "Model-based testing with labelled transition systems".

For each of the following predicates, state whether it is *true* or *false*.
– q **after** $\{but\}$ **refuses** liq
– u **after** $\{but \cdot but\}$ **refuses** liq
– v **after** $\{but \cdot but\}$ **refuses** liq

3 Inputs and outputs

Reading

Read Section 3.3 of the paper "Model-based testing with labelled transition systems".

This section introduces the notions of input and output actions. An input is an action initiated by the environment and an output is an action initiated by the system. A convention is defined where input actions are prefixed with '?' and output actions with '!'. It is important to observe that '?' and '!' are *not a part of the elements of* L_I and L_U; they are simply syntax within an LTS to distinguish between inputs and outputs. This is important later, when we define test cases is Section 1!

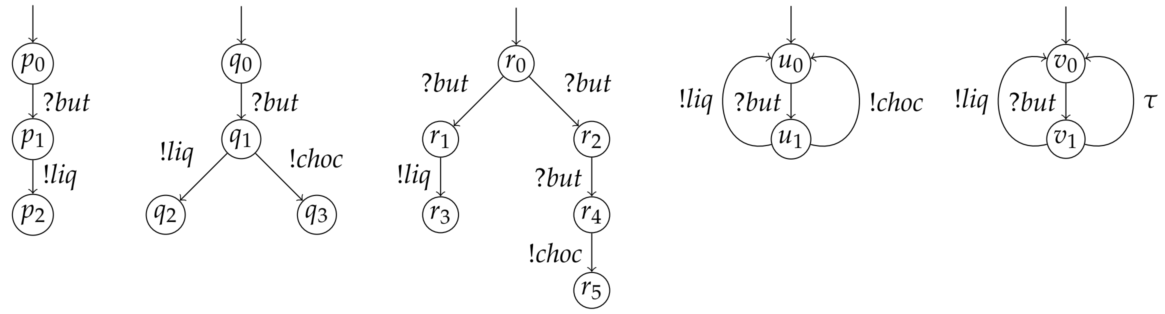


FIGURE 8.1 LTSs from Figure 2 in "Model-based testing with labelled transition systems" reinterpreted as LTS with inputs and outputs using $L_I = \{but\}$ and $L_U = \{liq, choc\}$

Recall that we have already seen this convention in learning unit 3. UPPAAL-timed automata use inputs and outputs to define synchronization.

A labelled transition system with inputs and outputs is simply an LTS in which the inputs L_I and outputs L_U have been explicitly identified. Note that $L_I \cap L_U = \emptyset$, hence within an LTS, every action is either an input or an output, but never both.

We can reinterpret the LTSs from Figure 2 in the paper as LTSs with inputs and outputs, by defining $L_I = \{but\}$ and $L_U = \{liq, choc\}$. Using the convention that inputs are prefixed with '?' and outputs are prefixed with '!' we typically draw them as shown in Figure 8.1.

4 Input-output transition systems

Reading

Read Section 3.4 of the paper "Model-based testing with labelled transition systems".

Input-Output Transition Systems (IOTS) are LTSs with inputs and outputs, with an additional requirement. The idea is that the environment can never refuse an output produced by the system and the system can never refuse an input sent by the environment, as a consequence, an IOTS is *input-enabled*. This means that in every state that can be reached from the initial state (denoted in Definition 7 by $der(q_0)$) every *input action* must be enabled (possibly preceded by one or more τ-transitions).

As explained in the section, IOTSs are used to model implementation under test and LTSs are used to model specifications. The main difference is that an IOTS is input-enabled while an LTS may not be input-enabled. As explained later this is used to allow for partial specifications. Angelic and demonic completions can be used to transform an LTS into an IOTS. A system runs with its environment by synchronizing on matching input/output pairs.

QUESTION 8.4
Consider the LTSs with inputs and outputs from Figure 8.1. Which of these LTSs is, in fact, an IOTS?

When considering systems with inputs and outputs, quiescence is a very important notion, in particular to the testing theory proposed by Tretmans and presented in this course. A system might reach a state where it only accepts inputs. In this state, the system cannot perform any internal or output actions. In such a state, the system does not have the initiative for making progress. It must wait for the environment to supply an input. Such a state is defined as a quiescent state. The formal definition of a quiescent state is given in Definition 8.1 on page 16.

QUESTION 8.5
Consider the LTSs with inputs and outputs q, u, and v from Figure 8.1.

Which states in these three LTSs are quiescent?

Quiescence is considered a visible output action in Tretmans' theory. The environment can observe that a system does not produce any output. This is achieved in Definition 9 on page 16 by extending quiescent states with a self-loop (a transition from a state to itself) labelled by δ. The set of transitions extended with δ is denoted T_δ. Traces over T_δ (instead of T) are called *suspension traces*, or *Straces*.

QUESTION 8.6
Consider the LTS with inputs and outputs q from Figure 8.1.

Give the set $Straces(q)$.

It is questionable whether considering quiescence a visible output action is realistic or not. How long should one wait to observe quiescence? A detailed answer to this question falls outside the scope of this course. A practical answer is to wait long enough so that one is sure that no output will be observed anymore. This is a solution used in many tools, like JTORX. Another solution is to equip a SUT with an indicator turning a light on when a quiescent state is reached. Finally, quiescence can also be ignored, as it is done in the tool UPPAAL-TRON. TRON is a special tool dealing with real-time systems and is discussed in more detail in the project assignment.

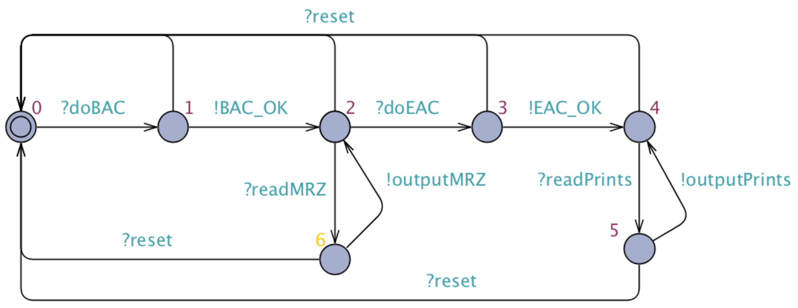


FIGURE 8.2 A specification for a simple electronic passport

EXERCISES

Consider the specification of an electronic passport in Figure 8.2. Note that the specification is drawn in the UPPAAL style of timed automata, however, since the specification does not use any timers or constraints, it is simply an LTS.

The idea of an electronic passport is to include a chip in the passport. The information stored on that chip are typically a "memory readable zone" (MRZ) containing basic data like passport number, date and country of issue, and more sensitive data like fingerprints. Access to these data are protected by two protocols. A "basic access control" (BAC) protocol rules the access to the MRZ while an "extended access protocol" (EAC) rules the access to sensitive data. Reading the information in the chip takes place via a wireless card reader. The passport has to be put close to the reader to be read.

In the initial state, users can initiate a BAC which is then accepted by the passport using a "BAC OK" message. Once the BAC protocol is completed, users can read the MRZ. After successful completion of a BAC step, the EAC protocol can be initiated to access sensitive data. In every state, it is possible to reset the passport, often by moving the passport away from the reader terminal.

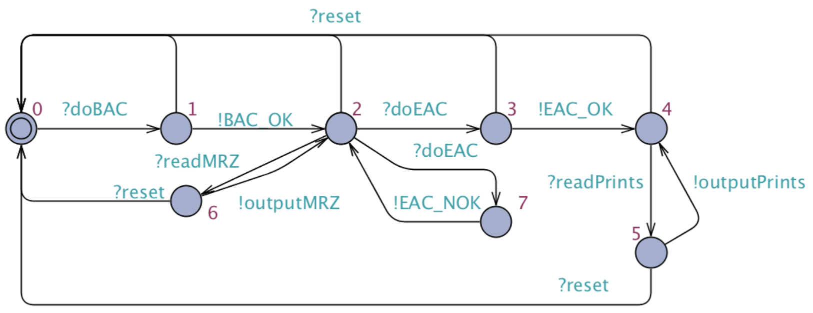


FIGURE 8.3 Another specification for a simple electronic passport

Figure 8.3 shows a slight variation of the specification. When performing an EAC step, the passport might reach a state from which the EAC will fail, indicated using a "EAC not OK" message. In practice, due to temperature variation and heavy cryptographic computations, this behaviour can actually be observed.

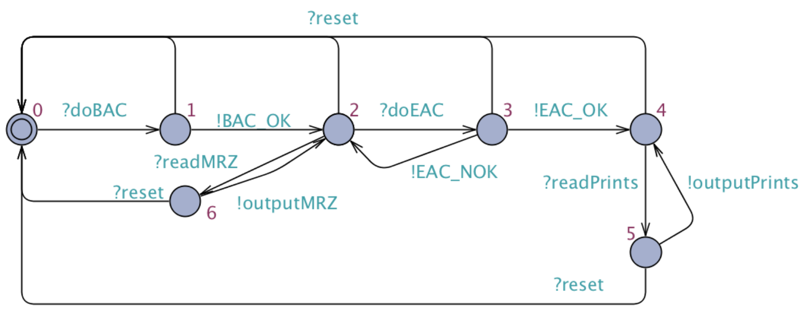


FIGURE 8.4 Another specification for a simple electronic passport

Finally, Figure 8.4 shows a third specification where the EAC step might fail.

1 For which specification(s) are the following traces valid:
  a $\sigma_1 = ?doBAC \cdot !BAC\_OK \cdot ?readMRZ \cdot !outputMRZ \cdot ?reset$
  b $\sigma_2 = ?doBAC \cdot !BAC\_OK \cdot ?doEAC$
  c $\sigma_3 = ?doBAC \cdot !BAC\_NOK$
  d $\sigma_4 = ?doBAC \cdot !BAC\_OK \cdot ?doEAC \cdot !EAC\_NOK$

2 From each initial state of the three specifications, compute the **after**-sets for the traces mentioned in the previous questions.

3 Which of the specification(s) is(are) non-deterministic?

4 Indicate for each specification which state(s) is(are) quiescent.

5 For which specifications are the following suspension traces valid:
  a $\sigma_5 = ?doBAC \cdot \delta \cdot !BAC\_OK \cdot ?readMRZ \cdot !outputMRZ \cdot ?reset$
  b $\sigma_6 = ?doBAC \cdot !BAC\_OK \cdot \delta \cdot \delta \cdot \delta \cdot ?readMRZ \cdot !outputMRZ \cdot ?reset$
  c $\sigma_7 = ?doBAC \cdot !BAC\_OK \cdot \delta \cdot ?doEAC \cdot !EAC\_NOK$

FEEDBACK

Answers to questions

8.1 Recall that we identify an LTS with its initial state.

- $traces(q) = traces(q_0) = \{\epsilon, but, but \cdot liq, but \cdot choc\}$
- q **after** $\{but\} = \{q_1\}$
- u **after** $\{but \cdot but\} = \emptyset$
- v **after** $\{but \cdot but\} = \{v_1\}$

8.2 Yes it is possible for an LTS to be image finite but infinite state. Infinite state means that the number of states is infinite. Image finite means that the set of reachable states after *one* action is finite.

8.3 For each of the cases, first compute the **after**-set, and then determine whether **refuses** holds or not.

- q **after** $\{but\}$ **refuses** $liq = \{q_1\}$ **refuses** $liq = false$
- u **after** $\{but \cdot but\}$ **refuses** $liq = \emptyset$ **refuses** $liq = false$
- v **after** $\{but \cdot but\}$ **refuses** $liq = \{v_0\}$ **refuses** $liq = true$
- v **after** $\{but\}$ **refuses** $liq = \{v_0, v_1\}$ **refuses** $liq = true$

8.4 Only the LTS for v is an IOTS. In every (reachable) state of v (v_0 and v_1), the $?but$ action is enabled. Since the set of inputs $L_I = \{?but\}$, the LTS is input-enabled. All other LTSs have at least one state in which $?but$ is not enabled, for instance, p_2, q_2, r_3 and u_1.

8.5 States q_0, q_2, q_3, u_0 and v_0 are quiescent since these states neither have an enabled output, nor a τ-transition.

8.6 Recall that we identify an LTS with its initial state, so, for instance, $Straces(q) = Straces(q_0)$. $Straces(q) = \{\epsilon, \delta, but, \delta \cdot but, but \cdot liq, but \cdot choc, \delta \cdot but \cdot liq, \delta \cdot but \cdot choc, but \cdot liq \cdot \delta, but \cdot choc \cdot \delta, \delta \cdot but \cdot liq \cdot \delta, \delta \cdot but \cdot choc \cdot \delta\}$
Note that technically, this is not the full set of suspension traces, since each occurrence of δ can be repeated infinitely many times (so for any LTS with inputs and outputs that has a quiescent state, the set of suspension traces is infinite).

Answers to exercises

1 a Trace σ_1 is a valid trace of all the specifications.
b Trace σ_2 is a valid trace of all the specifications.
c Trace σ_3 is a valid trace of none of the specifications.
d Trace σ_4 is a valid trace of specifications in Figures 8.3 and 8.4.

2 The **after**-sets are as follows:
a $(0$ **after** $\sigma_1) = \{0\}$ for all specifications.
b $(0$ **after** $\sigma_2) = \{3\}$ for specifications 8.2 and 8.4. It is $\{3, 7\}$ for 8.3.
c $(0$ **after** $\sigma_3) = \emptyset$ for all specifications.
d $(0$ **after** $\sigma_4) = \emptyset$ for 8.2 and $(0$ **after** $\sigma_4) = \{2\}$ for 8.3 and 8.4.

3 Only specification 8.3 is non-deterministic.

4 In all specifications, states 0, 2, and 4 are quiescent.

5 a Trace σ_5 is a valid suspension trace of none of the specifications.
b Trace σ_6 is a valid suspension trace of all the specifications.
c Trace σ_7 is a valid trace of specifications 8.3 and 8.4.

The ioco implementation relation

Learning unit 9

The ioco implementation relation

INTRODUCTION

This learning unit presents one instantiation of the conformance relation of the model-based testing framework. This relation assumes that specifications and implementations are represented as labelled transition systems. The relation is named **ioco**, for **i**nput **o**utput **co**nformance.

LEARNING GOALS
After having studied this learning unit you should be able to:
– explain the intuition behind **ioco**
– give the formal definition of **ioco**
– apply **ioco** on small examples
– give important properties of **ioco**
– discuss weaknesses and variations of **ioco**

Reading

Parts of Section 4 of the paper "Model-based testing with labelled transition systems" by Tretmans belong to this learning unit.

Exercises

There are several exercises and additional questions at the end of this learning unit. You should work on all of them.

Time

The expected time needed to study this learning unit is about 6 hours.

KERNEL

1 The implementation relation ioco

Reading

Read the introduction of Section 4 and Section 4.1 of the paper "Model-based testing with labelled transition systems" by Tretmans.

The intuition behind **ioco** is that whenever an implementation produces an output the specification can produce the same output and if the implementation cannot produce any output - that is, the implementation is quiescent - the specification is also quiescent.

The first element of the formal definition of **ioco** (Definition 12) is the **out**-set. Notice that in the expression $\textbf{out}(p \textbf{ after } \sigma)$, p means the initial state of transition system p. The set $\textbf{out}(q)$ identifies all outputs that can be produced when a system (the IUT or the specification) is in a given state q (Definition 11-1). The **out**-set is composed of two parts: (1) the output actions ($x \in L_U$) that are possible in state q (i.e., all x for which $q \xrightarrow{x}$) and (2) the quiescence action δ if state q is quiescent (i.e., if $\delta(q)$ holds).

The second part of the definition generalises the **out**-set to a set of states (Definition 11-2). This is the consequence of one important aspect of **ioco**, namely, non-determinism. Let us consider example k_3 in Figure 4 on page 14 of the paper. This system is non-deterministic. When pushing a button from the initial state, two different states can be reached. As shown in Example 8 on page 20 of Tretmans' paper, the **out**-set after this input action $?but$ is composed of the output action $!liq$ and quiescence δ. Indeed, action $!liq$ is possible in state r_1 of system k_3 and state r_2 is quiescent. These two states can be reached after pushing the button, that is, after input $?but$.

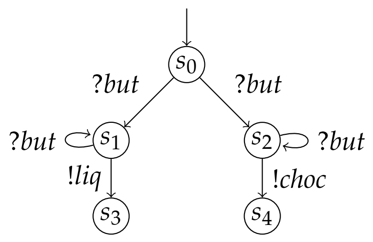


FIGURE 9.1 LTS with inputs and outputs where $L_I = \{?but\}$, and $L_U = \{!liq, !choc\}$

QUESTION 9.1
Consider the LTS with inputs and outputs in Figure 9.1.

Determine the following **out**-sets.
– **out**(s_0)
– **out**(s_1)
– **out**(s_2)
– **out**(s_3)
– **out**$(\{s_0, s_1\})$
– **out**$(\{s_1, s_2\})$
– **out**$(\{s_1, s_2, s_3\})$

In the definition of **ioco**, the **out**-set is typically used in combination with **after**. Here, **after** results in the sets of states reached from the initial states of the implementation and the specification.

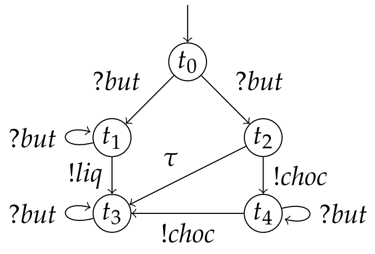


FIGURE 9.2 IOTS where $L_I = \{?but\}$, and $L_U = \{!liq, !choc\}$

QUESTION 9.2
Consider the IOTS in Figure 9.2.

Determine the following **out**-sets.
a $\mathbf{out}(\{t_0\}\ \mathbf{after}\ ?but)$
b $\mathbf{out}(\{t_0\}\ \mathbf{after}\ ?but \cdot ?but)$
c $\mathbf{out}(\{t_0\}\ \mathbf{after}\ ?but \cdot \delta)$
d $\mathbf{out}(\{t_0\}\ \mathbf{after}\ \varepsilon)$
e $\mathbf{out}(\{t_0\}\ \mathbf{after}\ ?but \cdot !choc)$

Caution!

Example 10 in the paper confuses non-determinism with branching. These two should *not* be confused. In Figure 4, system k_2 is deterministic but expresses a choice after pushing ?*but*: a correct implementation may either produce liquorice (!*liq*) or chocolate (!*choc*). After each one of these actions, a single state is reached.

Definition 12 formally defines **ioco** and the intuitive idea that implementation must only produce outputs – including quiescence – that are foreseen by the specification. The formalisation is that the **out**-set of the implementation must be a subset of the **out**-set of the specification. One should also note that implementations are IOTSs, and are therefore input-enabled ($i \in \text{IOTS}(L_I, L_U)$). This means that in an implementation in every state every input action is possible. This comes from the test hypothesis discussed in the next learning unit.

Further, it is important to observe that conformance is only checked for those suspension traces that are suspension traces of the *specification* ($Straces(s)$). This restriction allows for partial specifications. For all unspecified traces, the implementation is free to implement anything it likes. We come back to underspecified traces and complete specification later.

Examples 9 (in combination with Figure 8) and 10 show implementations and specifications. The examples show which pairs are conforming and which ones are not.

QUESTION 9.3
Consider the LTS with inputs and outputs s from Figure 9.1 as specification and IOTS t from Figure 9.2 as implementation.

Does the implementation t conform to the specification s, according to the **ioco** definition? Argue your answer.

QUESTION 9.4
Figure 9.3 shows an IOTS i as implementation, and an LTS with inputs and outputs s as specification.

Does the implementation i conform to the specification s, according to the **ioco** definition? Argue your answer.

QUESTION 9.5
In practice, using relation **ioco** means that one can observe internal stability of the IUT. True or false? Justify your answer.

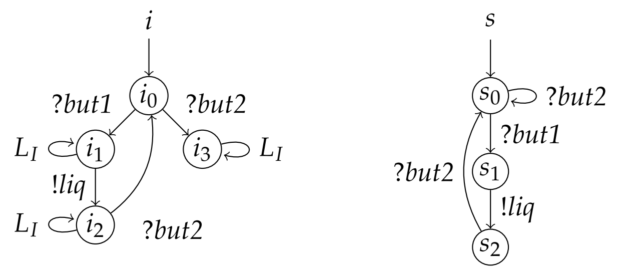


FIGURE 9.3 LTSs of implementation i and specification s. Input alphabet $L_I = \{?but1, ?but2\}$. Output alphabet $L_U = \{!liq\}$

QUESTION 9.6
Testing with **ioco** can only be applied to deterministic specifications. True or false? Justify your answer.

QUESTION 9.7
Consider the following two statements:
a An **out**-set of the specification that is considered by **ioco** is never empty.
b An **out**-set of the implementation that is considered by **ioco** is never empty.

True of false? Justify your answer.

2 Underspecified specifications and uioco

Reading

Read paragraph "Underspecified traces and uioco" of Section 4.2 of the paper "Model-based testing with labelled transition systems" by Tretmans.

Underspecified traces are an issue in **ioco**. As explained in the paper, there exist unspecified traces with well-specified prefixes. The objective of **uioco** is to clarify this ambiguity and to consider traces with any underspecified part as underspecified. It is then clear that **uioco** is not stronger than **ioco**. Example 12 illustrates the fact that **uioco** is strictly weaker than **ioco**.

We have seen different relations and they all give different verdicts. They are more or less permissive in what is a correct implementation of a given specification. It is up to users to determine which relation fits their needs. Which relation do you find more intuitive in Example 12: **uioco** or **ioco**?

QUESTION 9.8
An underspecified trace ($\sigma \in Utraces(s)$) is always a suspension trace ($\sigma \in Straces(s)$) True or false? Justify your answer.

3 Variants

Reading

Read paragraph "Variants" of Section 4.2 of the paper “Model-based testing with labelled transition systems” by Tretmans.

The article gives a list of different variations and extensions of the **ioco** theory. Discussing these extensions in detail falls outside the scope of this course.

4 Conclusion

Reading

Read Section 4.3 of the paper “Model-based testing with labelled transition systems” by Tretmans.

In this learning unit we have presented one conformance relation for model-based testing. This conformance relation is called **ioco**. Implementations are represented as input-enabled input output transition systems. Specifications are input output transition systems and may be underspecified and/or non-deterministic. An implementation is conforming to its specification if and only if whenever it produces an output the specification can also produce this output and whenever it cannot produce any output the specification also is quiescent.

The definitions of this learning unit have provided instances for the specification and implementation models and the relation "imp" in Figure 1, page 4 of the paper. In the next learning unit, we will instantiate the notion of test cases and how they are generated and executed.

EXERCISES

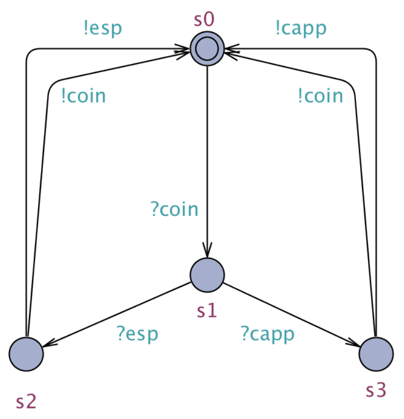


FIGURE 9.4 Specification s

1 Specification s in Figure 9.4 (note that the double circle means that s_0 is the initial state) specifies the behaviour of a coffee machine. The inputs and outputs are defined by $\mathcal{L}_I = \{?coin, ?capp, ?esp\}$ and $\mathcal{L}_U = \{!coin, !capp, !esp\}$. After inserting a coin, users choose between espresso and cappuccino. The machine then either returns the inserted money or the selected beverage. Action $?coin$ represents the insertion of a coin, action $?esp$ represents the selection of espresso, and action $?capp$ represents the selection of cappuccino. Action $!coin$ represents returning the coin, $!esp$ represents the production of espresso, and $!capp$ represents the production of cappuccino.

- Is this specification deterministic ?
- Is this specification input-enabled ?
- Which state(s) is (are) quiescent ?

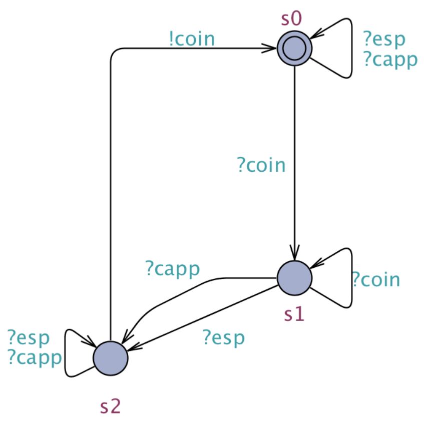


FIGURE 9.5 Implementation i_1

2 Consider implementation i_1 in Figure 9.5.

- Is i_1 deterministic ?
- Which states are quiescent ?

3 What is the **out**-set of i_1 after the following suspension traces:
- $\sigma_1 = ?coin$
- $\sigma_2 = ?coin \cdot ?esp$
- $\sigma_3 = ?coin \cdot ?capp$

4 Is i_1 **ioco**-conforming to specification s? If not, give a suspension trace justifying your answer.

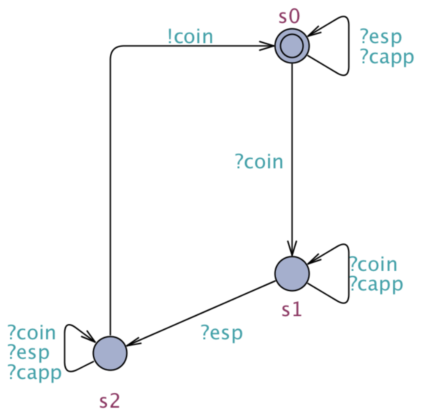


FIGURE 9.6 Implementation i_2

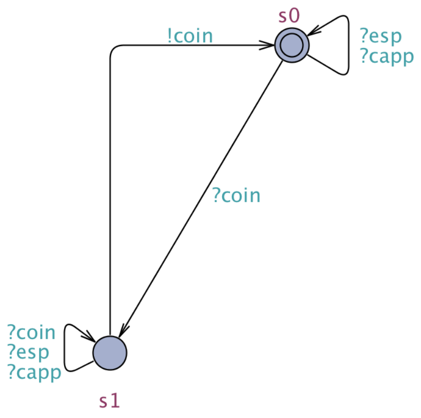


FIGURE 9.7 Implementation i_3

5 Consider implementations i_2 and i_3 in Figures 9.6 and 9.7.
- Is i_2 deterministic ? Is i_3 deterministic ?
- Which states are quiescent?

6 Is i_2 **ioco**-conforming to specification s? Is i_3 **ioco**-conforming to specification s? If not, give a suspension trace justifying your answer.

7 Consider specification p in Figure 9.8 and implementation j in Figure 9.9. The languages are $\mathcal{L}_I = \{?coin, ?button\}$ and $\mathcal{L}_U = \{!tea, !coffee\}$.
- Is specification p deterministic? Is implementation j deterministic ?
- Which state(s) of these two systems is (are) quiescent?

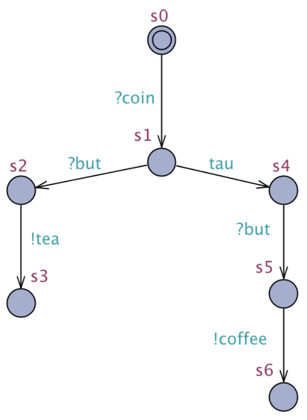


FIGURE 9.8 Specification p.

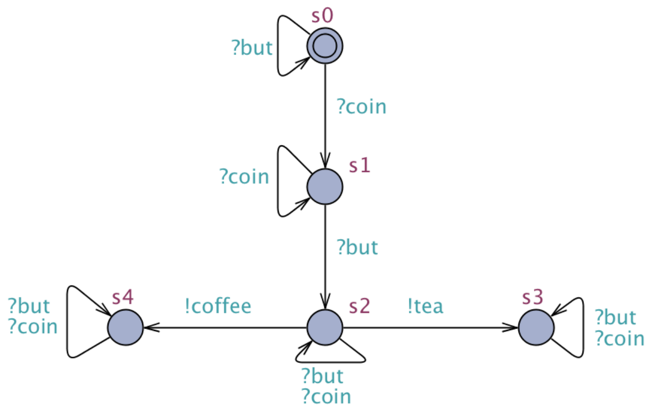


FIGURE 9.9 Implementation j.

8 Is j **ioco**-conforming to specification p? If not, give a suspension trace justifying your answer.

9 Consider specification p' in Figure 9.10. It corresponds to specification p where the internal step is now labelled with an input action. After one press on the button the machine produces coffee or tea. After pressing the button twice the machine only produces coffee.

Is j **ioco**-conforming to specification p'? If not, give a suspension trace justifying your answer.

10 Is the trace $?coin \cdot ?button \cdot ?button$ an underspecified trace of j
- according to **ioco**?
- according to **uioco**?

11 Is j **uioco**-conforming to specification p'? If not, give a suspension trace justifying your answer.

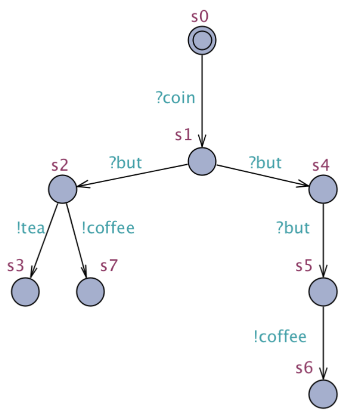


FIGURE 9.10 Implementation p'.

FEEDBACK

Answers to questions

9.1 We obtain the following **out**-sets.

- $\mathbf{out}(s_0) = \{\delta\}$
- $\mathbf{out}(s_1) = \{!liq\}$
- $\mathbf{out}(s_2) = \{!choc\}$
- $\mathbf{out}(s_3) = \{\delta\}$
- $\mathbf{out}(\{s_0, s_1\}) = \{\delta, !liq\}$
- $\mathbf{out}(\{s_1, s_2\}) = \{\delta, !choc\}$
- $\mathbf{out}(\{s_1, s_2, s_3\})\{\delta, !liq, !choc\}$

9.2 We obtain the following **out**-sets.

- $\mathbf{out}(\{t_0\}\ \mathbf{after}\ ?but) = \mathbf{out}(\{t_1, t_2, t_3\}) = \{!liq, !choc, \delta\}$
- $\mathbf{out}(\{t_0\}\ \mathbf{after}\ ?but \cdot ?but) = \mathbf{out}(\{t_1, t_3\}) = \{!liq, \delta\}$
- $\mathbf{out}(\{t_0\}\ \mathbf{after}\ ?but \cdot \delta) = \mathbf{out}(\{t_3\}) = \{\delta\}$
- $\mathbf{out}(\{s_0\}\ \mathbf{after}\ \varepsilon) = \mathbf{out}(\{t_0\}) = \emptyset$
- $\mathbf{out}(\{t_0\}\ \mathbf{after}\ ?but \cdot !choc) = \mathbf{out}(\{t_4\}) = \{!choc\}$

9.3 No, the implementation does not conform to the specification. For t **ioco** s to hold, we need to have $\forall \sigma \in Straces(s)\colon \mathbf{out}(t\ \mathbf{after}\ \sigma) \subseteq \mathbf{out}(s\ \mathbf{after}\ \sigma)$. Now consider for instance the suspension trace $\sigma = ?but \cdot ?but$. We compute the following output sets:

- $\mathbf{out}(t\ \mathbf{after}\ \sigma) = \{!liq, \delta\}$
- $\mathbf{out}(s\ \mathbf{after}\ \sigma) = \{!liq, !choc\}$

Hence $\mathbf{out}(t\ \mathbf{after}\ \sigma) \not\subseteq \mathbf{out}(s\ \mathbf{after}\ \sigma)$, and thus not t **ioco** s.

9.4 No, the implementation does not conform to the specification. Recall that i **ioco** $s = \forall \sigma \in Straces(s).\mathbf{out}(i\ \mathbf{after}\ \sigma) \subseteq \mathbf{out}(s\ \mathbf{after}\ \sigma)$. The trace $?but2 \cdot ?but1$ is a suspension trace of the specification, i.e., $?but2 \cdot ?but1 \in Straces(s)$. We have:

- $\mathbf{out}(i\ \mathbf{after}\ ?but2 \cdot ?but1) = \{\delta\}$
- $\mathbf{out}(s\ \mathbf{after}\ ?but2 \cdot ?but1) = \{!liq\}$

so, $\mathbf{out}(i \textbf{ after } ?but2 \cdot ?but1) \not\subseteq \mathbf{out}(s \textbf{ after } ?but2 \cdot ?but1)$, thus not i **ioco** s.

9.5 True. Relation **ioco** considers suspension traces and therefore assumes that quiescence is a visible action. In theory, an IUT should have a way to output that it has reached a quiescent state. In practice, quiescence is often implemented as a time-out. If an IUT does not output anything during a given time limit, it is considered quiescent. Anyone is of course free to question this implementation.

9.6 False. Specifications and implementation may be non-deterministic. This is one strength of the **ioco** relation. Non-determinism is not a problem, since **ioco** is defined using an **out**-set that can contain several states.

9.7 a True. This is formally expressed in Proposition 2.1 on page 25. By definition, **ioco** only considers traces of the specification. By Proposition 2.1 it directly follows that the **out**-set of a specification is never empty.

b False. For implementations, it is important to observe that the **out**-sets that **ioco** considers are **out**-sets for suspension traces of the *specifications*. There are are two cases: (1) a correct implementation and (2) an incorrect implementation. The first is easy, as the suspension trace of the specification is also a suspension trace of the implementation. For incorrect implementations, the trace of the specification may not be a trace of the implementation. Suppose the trace is σ, since implementation **after** $\sigma = \emptyset$, the **out**-set is empty.

9.8 True. It follows from the definition of underspecified traces (Definition 15.1 on page 25).

Answers to exercises

1 According to the formal definition of determinism (Definition 5.9 on page 9 of the paper "Model Based Testing with Labelled Transition Systems" by Tretmans) specification s is deterministic. The reason is that after every action at most one state is reached.

Specification s is not input-enabled. The initial state does not accept inputs $?capp$ and $?esp$; state s_1 does not accept input $?coin$; state s_2 does not accept input $?capp$; and state s_3 does not accept input $?esp$ or input $?capp$.

States s_0 and s_1 are quiescent. They do not accept any internal steps (τ transitions) nor outputs.

2 Implementation i_1 is deterministic.

States s_0 and s_1 of implementation i_1 are quiescent.

3
- $\mathbf{out}(i_1 \textbf{ after } ?coin) = \{\delta\}$
- $\mathbf{out}(i_1 \textbf{ after } ?coin \cdot ?esp) = \{!coin\}$
- $\mathbf{out}(i_1 \textbf{ after } ?coin \cdot ?capp) = \{!coin\}$

4 Even though i_1 never produces beverage and only returns the inserted money, it is **ioco**-conforming to specification s. The reason is that after all suspension traces of s the **out**-set of i_1 is a subset of the **out**-set of s.

5 Implementations i_2 and i_3 are deterministic.

States s_0 and s_1 of implementation i_2 are quiescent. State s_0 of implementation i_3 is quiescent.

6 Implementation i_2 is not **ioco**-conforming to specification s. The reason is that after inserting a coin and choosing for cappuccino, implementation i_2 is quiescent and nothing is produced. Formally, we have the following:
– $\textbf{out}(i_2 \textbf{ after } ?coin \cdot ?capp) = \{\delta\}$
– $\textbf{out}(s \textbf{ after } ?coin \cdot ?capp) = \{!coin, !capp\}$
– $\{\delta\} \not\subseteq \{!coin, !capp\}$

Implementation i_3 is also not **ioco**-conforming to specification s. After inserting a coin, this implementation is not quiescent while the specification is quiescent. Formally, we have the following:
– $\textbf{out}(i_3 \textbf{ after } ?coin) = \{!coin\}$
– $\textbf{out}(s \textbf{ after } ?coin) = \{\delta\}$

7 – The specification is not deterministic. After inserting a coin and pushing the button, two states are reachable. Formally, we have $p \textbf{ after } ?coin \cdot ?but = \{s_2, s_5\}$. The implementation is deterministic. After any trace there is always at most one state that is reachable.

– States s_0, s_3, s_4, s_6 of the specification are quiescent. Note that state s_1 is not quiescent because it has an outgoing τ transition. Formally (see Definition 8.1 on page 16 of the paper "Model-based testing with labelled transition systems" by Tretmans), a state is quiescent if it has no outgoing output transitions and no outgoing τ transitions. States s_0, s_1, s_3, s_4 of the implementation are quiescent.

8 Implementation j is not **ioco**-conforming to specification p. The reason is subtle and it comes from the observation of quiescence. The specification is non-deterministic but using quiescence it is possible to distinguish between entering the left or the right branch, that is, producing coffee or tea. Consider the trace $?coin \cdot ?but$. The states of the specification reached after it are s_2 and s_5. State s_1 is not quiescent because of the internal step. Therefore after the trace $?coin \cdot \delta \cdot ?but$ only state s_5 is reached (since $?coin \cdot \delta \cdot ?but$ can be seen as an abstraction of $?coin \cdot \tau \cdot \delta \cdot ?but$). Regarding the implementation, state s_1 is quiescent and these two traces both lead to state s_2 where coffee and tea can be produced. Formally, we have the following derivation:
– $\textbf{out}(p \textbf{ after } ?coin \cdot \delta \cdot ?but) = \{!coffee\}$
– $\textbf{out}(j \textbf{ after } ?coin \cdot \delta \cdot ?but) = \{!coffee, !tea\}$
– $\{!coffee, !tea\} \not\subseteq \{!coffee\}$

9 Implementation j is not **ioco**-conforming to specification p'. The reason is that after pushing the button twice the implementation can produce coffee and tea while the specification can only produce coffee. Formally, we have the following derivation:
- $\mathbf{out}(p' \textbf{ after } ?coin \cdot ?but \cdot ?but) = \{!coffee\}$
- $\mathbf{out}(j \textbf{ after } ?coin \cdot ?but \cdot ?but) = \{!coffee, ?tea\}$
- $\{!coffee, !tea\} \nsubseteq \{!coffee\}$

10
- Trace $?coin \cdot ?but \cdot ?but$ is an element of $Straces(j)$. Thus, it is not an underspecified trace according to **ioco**.
- Trace $?coin \cdot ?but \cdot ?but$ is not an element of $Utraces(j)$. It is thus an underspecified trace of j according to **uioco**. The issue is that there is a branch in j that can refuse the second $?but$ action. We use Definition 15.1 on page 25 of the paper "Model-based testing with labelled transition systems" with $\sigma_1 = ?coin \cdot ?but$, $a = ?but$ and $\sigma_2 =$. We obtain that j **refuses** $?but$ **after** σ_1. Definition 15.1 is not satisfied and thus this trace is an underspecified trace of j according to **uioco**.

11 Yes, implementation j is **uioco**-conforming to specification p'. Because trace $?coin \cdot ?but \cdot ?but$ is not an underspecified trace, it can no longer be a reason for non-conformance according to **uioco**.

Test generation and execution

Learning unit 10

Test generation and execution

INTRODUCTION

The previous learning unit introduced conformance relation **ioco**. In this learning unit, we define the notion of test cases and test execution. We give an algorithm that generates sound and exhaustive test suites for the **ioco** relation.

LEARNING GOALS
After having studied this learning unit you should be able to:
– formally define the notion of test cases and test execution
– explain the details of the test generation algorithm
– discuss completeness of the test generation algorithm

Reading

Sections 3.5, 5, and 6 of the paper "Model-based testing with labelled transition systems" by Tretmans in the reader belong to this learning unit.

Exercises

A number of questions are given in this learning unit. You should work on these questions. You should work on the exercises at the end of this learning unit.

Time

The expected time needed to study this learning unit is about 6 hours.

KERNEL

1 Test cases

Reading

Read Section 3.5 of the paper "Model-based testing with labelled transition systems" by Tretmans.

Test cases are input output transition systems (IOTSs) with special properties. They are deterministic. In any state they offer at most one input to the IUT. Their only final states are **pass** or **fail** verdict states, and these are sink states.

Inputs and outputs are mirrored in a test case. The inputs of a test cases are the outputs of the specification and the implementation. The outputs of a test case are the inputs of the specification and the implementation. We will, however, follow the paper, and stick to the convention that inputs and output always are relative to the specification and the implementation. For example, when the specification and implementation have inputs $L_I = \{?but\}$ and outputs $L_U = \{!liq, !choc\}$, then in the test cases we will have *output* $?but$ and *inputs* $!liq$ and $!choc$.

Definition 10.1 on page 18 of "Model-based testing with labelled transition systems" defines test cases formally. It introduces a special label θ to detect quiescence. Note that the second point of the definition means that a verdict can only be reached after observing an output action or a quiescence action. Understanding the details of the formulas is not necessary here. It is also important to note the fourth point in the definition. This specifies that in every state of the test case, either

- *exactly one* input from L_I is provided to the IUT, and *all outputs* from L_U are accepted, but quiescence is *not* observed, or
- *no input* is provided, but *all outputs* from L_U are accepted, and quiescence is observed.

So, every state in a test case that is not **pass** or **fail**, if it provides an input to the IUT, it also specifies all outputs from L_U, and does not specify quiescence. Also, when no input is provided to the IUT for every possible output in L_U, and quiescence, the test case specifies how to proceed.

IOTSs that satisfy the requirements from Definition 10.1 are called *test cases*, or TTSs.

QUESTION 10.1
Assume that we have a specification with $L_I = \{?but\}$ and $L_U = \{!liq, !choc\}$. Figure 10.1 shows four IOTSs.

a For each of the IOTSs in Figure 10.1 indicate whether it is a TTS. In other words, does it satisfy the requirements of test cases. If not, argue your answer.

b Suppose that we remove the transition $p_0 \xrightarrow{!choc} p_3$ from the IOTSfor p in the figure. Is the result a TTS? Why (not)?

c Suppose that we remove the transition $p_1 \xrightarrow{!liq} p_4$ from the IOTSfor p in the figure. Is the result a TTS? Why (not)?

2 Test execution

Reading

Read Section 5.1 of the paper "Model-based testing with labelled transition systems" by Tretmans.

Test execution is defined as a particular parallel composition. This composition takes into account quiescence and the special labels δ in implementations and θ in test cases. These two labels synchronise with each other. Note that the sets of actions in the test case and the implementation or specification are the same (but inputs and outputs have been exchanged), therefore, the only transitions performed without synchronisation are the internal actions.

An important aspect of the **ioco** testing theory is to allow for non-deterministic implementations. Because of this, multiple executions of the same test run may end with different verdicts. An implementation passes a test if and only if all possible test runs end with verdict pass. An important assumption is that an implementation cannot hide its behaviours. The repetitive execution of a test case will force the implementation to reveal all its non-deterministic behaviours. A

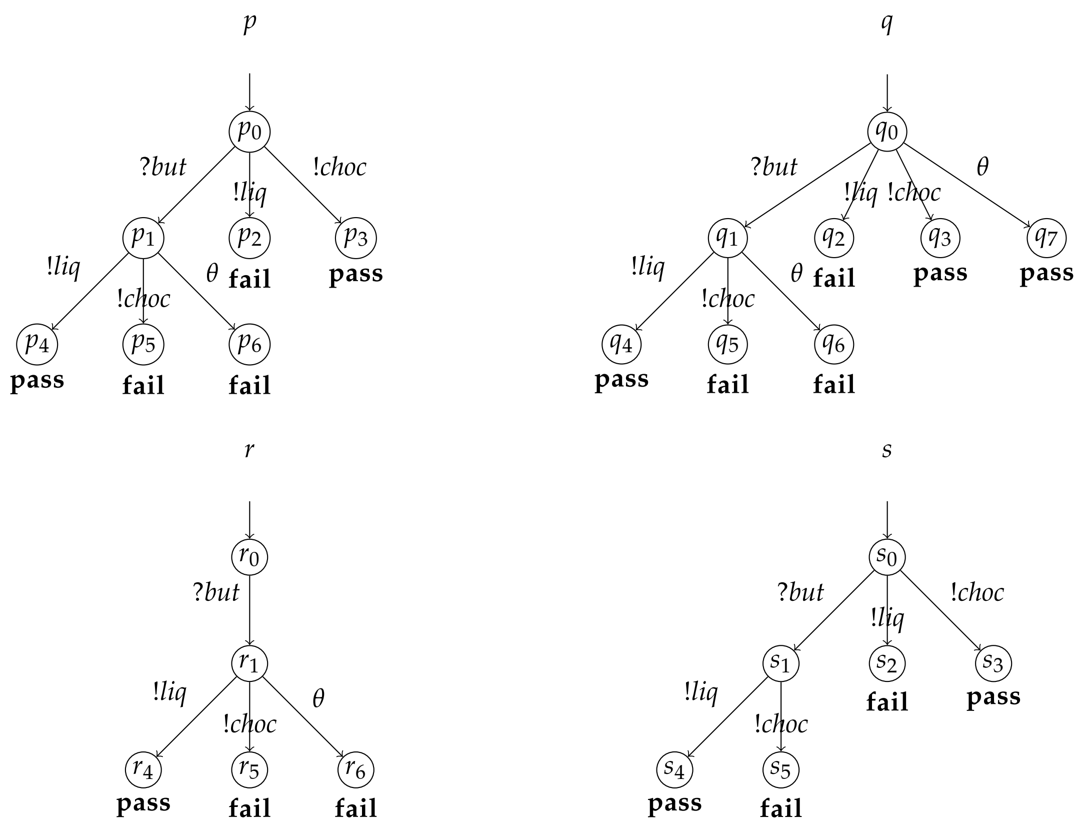


FIGURE 10.1 Four IOTSs with $L_I = \{?but\}$, $L_U = \{!choc, !liq\}$

non-deterministic IUT must by definition exhibit a non-deterministic behaviour. In practice, we do observe non-deterministic behaviours, with for instance, operating systems. It is often unclear why suddenly a system fails to behave properly. It is then hard to repeat this failure. The hypothesis assumes that re-execution of the same test case will test all behaviours. It does not make precise the number of times a test case should be executed. In practice, re-executing test cases will miss behaviours. Discussing when one should actually stop re-executing test cases is outside the scope of this course.

Definition 16 on page 30 of the paper "Model-based testing with labelled transition systems" formally defines test execution. Let us have a look at the three rules of the parallel composition.

The first rule defines how internal transitions of the implementation are executed. Test cases do not have internal transitions and a similar rule for test cases is not necessary.

$$\frac{i \xrightarrow{\tau} i'}{t\rceil|i \xrightarrow{\tau} t\rceil|i'}$$

This rule says that if the IUT can make an internal step, it can take this transition without modifying the test case. The test case and the implementation together reach a new state where the test case is unchanged and the implementation is in the state after the internal step.

The second rule defines how the test case and the implementation synchronise on an action a. Action a can be either an input or an output.

$$\frac{t \xrightarrow{a} t', i \xrightarrow{a} i'}{t \rceil | i \xrightarrow{a} t' \rceil | i'}$$

The rule reads as follows. If the test case and the implementation can both perform action a, their composition performs action a. In other words, if each member of the composition can perform action a, then the composition must also perform action a.

The last rules defines synchronisation between the IUT and the test case if quiescence is observed.

$$\frac{t \xrightarrow{\theta} t', i \xrightarrow{\delta} i'}{t \rceil | i \xrightarrow{\theta} t' \rceil | i'}$$

The rule is similar to the previous one but for the special actions θ and δ. If the test case accepts the observation of quiescence and the implementation is indeed quiescent, the composition can make progress and reach a new state.

A test run terminates with a verdict, **pass** or **fail** (Definition 16.2).

An IUT passes a test if and only if all possible test runs do not fail (Definition 16.3). An IUT passes a test suite if and only if it passes all tests of the test suite.

QUESTION 10.2
A test run always terminates with a verdict pass or fail.True or False ?
Justify your answer.

3 Test generation

Reading

Read Section 5.2 of the paper "Model-based testing with labelled transition systems" by Tretmans.

This part of the paper presents an algorithm that generates test cases for **ioco**. This algorithm (Algorithm 1) repeats three choices:
1 either it stops with verdict **pass**, no error has been found so far
2 it sends an input to the IUT and listens for outputs from the IUT
3 or it listens for outputs from the IUT and detects quiescence.

In steps 2 and 3, outputs (and quiescence) that is not allowed by the specification always leads to a **fail**. For all other states, the procedure is repeated.

QUESTION 10.3
Consider the LTS with inputs and outputs from Figure 10.2. Use the **ioco** test-generation algorithm to derive a test case that contains at least !, ?, θ, **pass**, and **fail**.

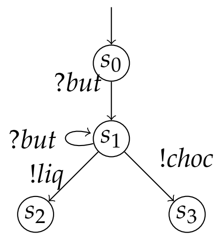


FIGURE 10.2 LTS with inputs and outputs where $L_I = \{?but\}$, and $L_U = \{!liq, !choc\}$

QUESTION 10.4
Consider the LTS with inputs and outputs from Figure 10.3. Use the **ioco** test-generation algorithm to derive a test case that contains at least $!$, $?$, θ, **pass**, and **fail**.

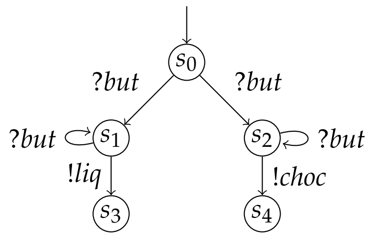


FIGURE 10.3 LTS with inputs and outputs where $L_I = \{?but\}$, and $L_U = \{!liq, !choc\}$

QUESTION 10.5
The test cases generated using Algorithm 1 satisfy the definition of test cases (Definition 10 on page 18). True or False ? Justify your answer.

It is important to notice that test cases are finite entities. This comes from the beginning of Algorithm 1: "a finite number of recursive applications of one of the following non-deterministic choices". Test cases nevertheless are unbounded, they can be arbitrary (but finitely) long. A test suite may also contain infinitely many unbounded - but finite - test cases.

QUESTION 10.6
An infinite test suite contains infinite test cases. True or False? Justify your answer.

4 Completeness: soundness and exhaustiveness

Reading

Read Section 5.3 of the paper "Model-based testing with labelled transition systems" by Tretmans.

An important property of the algorithm presented in the previous section is that it is complete. This means that it is a decision procedure for **ioco**. An IUT conforms to its specification if and only if it passes all tests. The "catch" is that a complete test suite usually contains infinitely many test cases.

Completeness is decomposed into soundness and exhaustiveness. In Definition 17.2, a test suite is sound if and only if the fact that an IUT is **ioco**-correct implies that it passes all tests of the test suite. In the contrapositive form, if an IUT fails a test then it is surely not **ioco**-correct. A sound test suite detects "real" errors.

In Definition 17.3, a test suite is exhaustive if and only if the fact that an IUT passes all tests implies that it is **ioco**-correct. In contrapositive form, if an IUT is not **ioco**-correct then the test suite contains a test case that can detect this error. An exhaustive test suite can detect all erroneous implementations of a specification. Such a test suite usually contains infinitely many test cases.

QUESTION 10.7
A test suite is sound if and only if all possible correct implementations pass the test suite. True or False ? Justify your answer.

QUESTION 10.8
Only incorrect implementations fail test cases generated using the **ioco** test generation procedure. True or False ? Justify your answer.

QUESTION 10.9
Consider the test case you specified in the answer to Question 10.3.

Is the test case sound? Argue your answer.

If the test case is sound, what is the smallest change that can be made to the test case to make it unsound? Give the resulting test case.

The paper also raises the interesting point of "test selection". This is still a research question and no satisfying answers has been given yet. The topic of test selection falls outside the scope of this course.

5 The global picture: completeness and test hypothesis

In this section, we put the different pieces together. The primary objective of model-based testing is to check if a real system – named *IUT* – is conforming to its formal specification expressed as an LTS – named *s*. In the context of the **ioco** relation, the goal is to check the following claim:

$$IUT \ \mathbf{ioco} \ s$$

This is achieved by executing test cases derived by Algorithm 1. When we apply these test cases we assume that if all tests pass then the real system is **ioco**-conforming to its specification. Let T_s be the set of all possible test cases that can be generated using Algorithm 1. We assume the following equivalence:

$$IUT \textbf{ passes } T_s \Leftrightarrow IUT \textbf{ ioco } s$$

The real system passes a test suite if and only if it is **ioco**-conforming to the specification. The issue is that **ioco** is defined for models of the IUT and not the IUT itself. So, let $i_{IUT} \in$ IOTS be a model of the IUT. Then, the correct formulation is the following:

$$IUT \textbf{ passes } T_s \Leftrightarrow i_{IUT} \textbf{ ioco } s$$

The test hypothesis for **ioco** is formally defined as follows.

$$\forall IUT.\exists i_{IUT} \in IOLTS.\forall T \in T_s.IUT \textbf{ passes } T \Leftrightarrow i_{IUT} \textbf{ passes } T$$

This formula states that for any real system there exists a model of that system such that the real system passes a test suite if and only if the model passes the test suite. The real system and its model are considered black boxes. It is only possible to observe their interfaces. The test hypothesis stipulates that using test cases only it is impossible to distinguish between the real system and its model.

We now use the test hypothesis together with completeness of the test generation procedure (Theorem 2, page 34) to prove that running test cases actually checks for **ioco**-conformance.

Assume a real system passes T_s. This means that infinitely many test cases have been run and they all terminated with verdict pass. We have established:

$$IUT \textbf{ passes } T_s$$

We now use the test hypothesis to deduce that a model of the real system also passes T_s.

$$IUT \textbf{ passes } T_s \Leftrightarrow \ \{\text{test hypothesis}\}\ i_{IUT} \textbf{ passes } T_s$$

Now we use the fact that T_s is complete.

$$i_{IUT} \textbf{ passes } T_s \Leftrightarrow \ \{\text{completeness}\}\ i_{IUT} \textbf{ ioco } s$$

From these equivalences, we can conclude that the IUT passes T_s if and only if the model of the *IUT* is **ioco**-conforming to the specification. So, we have formally proven the following formula:

$$IUT \textbf{ passes } T_s \Leftrightarrow i_{IUT} \textbf{ ioco } s$$

Formally, we cannot proceed any further. The intuition behind the test hypothesis is that the *IUT* and its model are equivalent. Therefore, we can informally conclude the *IUT* is **ioco**-conforming to its specification.

The main conclusion of this proof exercise is to convince ourselves that performing concrete test runs is checking for **ioco**-conformance.

6 Conclusion

Reading

Read Section 5.4 of the paper “Model-based testing with labelled transition systems” by Tretmans.

In this learning unit, we presented an algorithm generating and executing test cases for **ioco**. This algorithm serves as a basis for tool development.

EXERCISES

We consider specification s and implementations i_2 and i_3 from the exercise of the previous learning unit.

1 Using Algorithm 1 presented on page 31 of the paper "Model-based testing with labelled transition systems" build a test case t_2 demonstrating that implementation i_2 is not **ioco**-conforming to s. Describe a test run ending with verdict **fail**.

2 Using Algorithm 1 presented on page 31 of the paper "Model-based testing with labelled transition systems" build a test case t_3 demonstrating that implementation i_3 is not **ioco**-conforming to s. Describe a test run ending with verdict **fail**.

3 Is the test suite $\{t_2, t_3\}$ sound for specification s?

4 Is the test suite $\{t_2, t_3\}$ exhaustive for specification s? If not, can you construct an implementation that passes t_1 and t_2 but that is not **ioco**-conforming to s?

FEEDBACK

Answers to questions

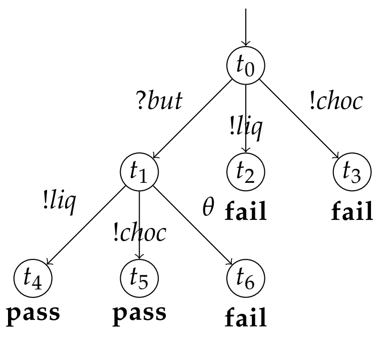
10.1 a
- p is a TTS, it satisfies all requirements for test cases.
- q is not a TTS. From q_0 it can provide an input, but it also checks for quiescence. This combination is not allowed.
- r is not a TTS. From r_0 it provides an input, but is does not specify what to do for any of the outputs. The latter should be specified.
- s is not a TTS. From s_1 it cannot proved any inputs, therefore it must specify what to do for *all* outputs and quiescence. Since it does not check for quiescence, this is not a test case.

b Note that p is a TTS. Removing the transition $p_0 \xrightarrow{!choc} p_3$ leads to the situation where not all outputs are handled from p_0, and the result is not a TTS.

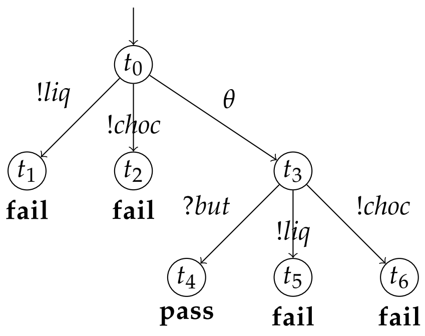
c Removing the transition $p_1 \xrightarrow{!liq} p_4$ leads to the situation where not all outputs are handled from p_1, and the result is not a TTS.

10.2 True. A test case can only stop if a verdict is reached.

10.3 The question implies that we need to all rules in the algorithm at least once, since without applying rule 1 there is no **pass**, without rule 2, no ? will be in the test case, and without rule 3 there will be no θ. There are many correct answers. The simplest test case that we can come up with applies rule 2 and rule 3 exactly once.
If we first apply rule 2 and then rule 3, we obtain the following test case.



First applying rule 3 and then rule 2 results in the following.



10.4 Observe that the specification is similar to the specification in Figure 10.2. The difference is in the moment the choice is made. In order to determine the valid continuation, in the test generation algorithm we keep track of all states the specification can be in. For instance, when starting by a $?but$ action, the specification can be in s_0 **after** $?but = \{s_1, s_2\}$; from this set of states, both $!choc$ and $!liq$ are valid continuations.
The test cases provided in the answer to Question 10.3 are hence also valid for this specification.

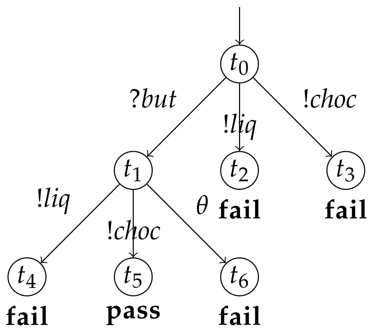
10.5 True. The most important property is that test cases are finite. Algorithm 1 generates test cases after a finite number of recursive applications of the three choices. In the three choices, test cases are deterministic and have no cycle. The test generation procedure ends only with verdict pass or fail. At most one input at a time is sent to the IUT. Test cases are output-enabled because in choices 2 and 3 all outputs are accepted.

10.6 False. An infinite test suite contains infinitely many finite test cases. By definition, test cases are finite.

10.7 False. The question defines exhaustiveness of a test suite.

10.8 True. By Theorem 2.1 all test cases generated by Algorithm 1 are sound.

10.9 The test case is sound, since it was generated using Algorithm 1 from the paper. Theorem 2 in "Model-based testing with labelled transition systems" proves that all testcases generated using the algorithm are sound.
A testcase is unsound if there is a correct implementation that fails the test case. The smallest change that can be made to the testcase is thus replacing one occurrence of **pass** by **fail**. The resulting testcase is the following. (We have changed the verdict for t_4.)



Answers to exercises

1 Figure 10.4 shows a possible test case. This test case is produced according to Algorithm 1 on page 31 by choice 2 twice, choice 3, and finally stopping with choice 1.
The following test run would find the error: $t_2 \rceil | i_2 \xRightarrow{?coin\cdot ?capp\cdot \theta} \textbf{fail} \rceil | s_1$.

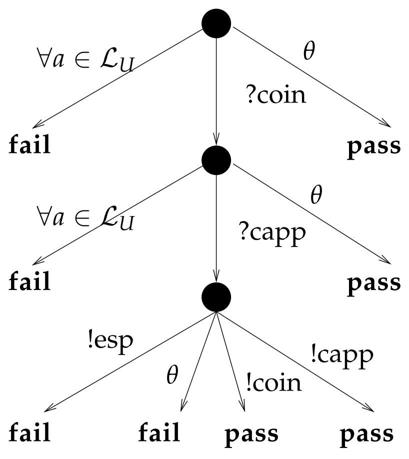


FIGURE 10.4 A possible test case t_2

2 Figure 10.5 shows a possible test case. This test case is produced according to Algorithm 1 on page 31 by choice 2, choice 3, and finally stopping with choice 1.

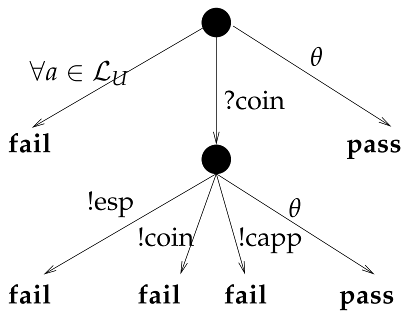


FIGURE 10.5 A possible test case t_3

The following test run would find the error: $t_3 \rceil | i_3 \xRightarrow{?coin \cdot \theta} \textbf{fail} \rceil | s_1$.

3 Because t_1 and t_2 are produced using Algorithm 1, they are sound. Indeed Algorithm 1 has been proven to only generate sound test cases (Theorem 2.1 on page 34 of the paper “Model-based testing with labelled transition systems”).
We can nevertheless check that they output a verdict **fail** only for invalid implementations. For instance, any implementation producing the output espresso after inserting a coin is invalid.

4 Figure 10.6 shows an implementation that passes t_1 and t_2 but it is not **ioco**-conforming to s. The reason is that after inserting a coin and pushing the "esp" button it produces cappuccino.

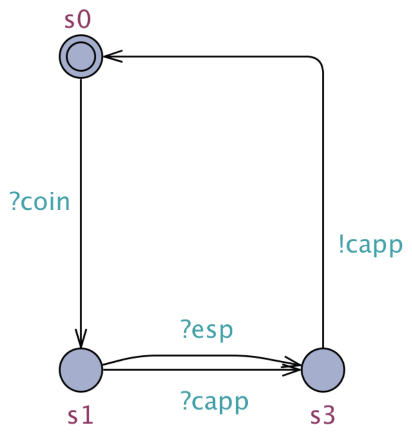


FIGURE 10.6 An invalid implementation passing t_1 and t_2