Reader Software Specification
2IX20
Version 2022-2023

Jeroen Keiren, Julien Schmaltz & Anton Wijs

Last minor update: 20 March 2023

Contents

Preface

This reader was written for the course 2IX20, software specification at Technische Universiteit Eindhoven (Eindhoven University of Technology). The content of the course is based on earlier incarnations, with contributions over the years by Mohammad Mousavi, Alexander Serebrenik, Julien Schmaltz and Anton Wijs.

This reader, where the informal specification using UML is based on Seidl et al. (2015) was first used for the course in academic year 2019-2020. The first nine chapters have been revised completely. Furthermore, the formalization of the semantics in Chapter 13 has been updated such that it reflects the semantics of UML state machine diagrams as discussed in Seidl et al. (2015); the previous formalization of the semantics resembled the semantics of Harel's state charts instead. Chapter 11 has been updated to include more accurate examples, and more formal description of the semantics, as well as many exercises. This chapter pretty closely follows the description in Baier and Katoen (2008). Chapter 14 was already added to the reader for the 2018-2019 run of the course; it is closely follows the paper Tretmans (2008). The content of this chapter follows the material of part III of the course System Verification & Testing of the Open Universiteit, for which Jeroen Keiren originally developed this chapter. The chapter is used with permission from the Open University of the Netherlands.

For academic year 2020-2021, the order of chapters in the reader was changed to reflect the order in which the material is presented in the course. Also, feedback and remarks by students in the previous year were processed to improve the quality and readability of the reader.

In 2021-2022, Chapter 10, introducing Promela and the SPIN model checker, along with some exercises, was added. This chapter was tested in 2020-2021. Also, Chapter 12 was extended with additional examples and exercises.

In 2022-2023, the order of the chapters on the Spin model checker and Promela (Chapter 10), Kripke structures (Chapter 9), LTL (Chapter 11), Labelled transition systems (Chapter 12) and the semantics of UML state machine diagrams (Chapter 13) was revised to improve the integration of the different chapters with each other. Also, the chapter on Spin and Promela was updated to use the command line version of Spin.

In this reader we have selected a large number of exercises. Some of the exercises for Chapters 4–8 have been adapted from multiple-choice exercises in the online learning environment that accompanies Seidl et al. (2015). Other exercises have been constructed specifically for this course, or have been inspired by other sources. Sample solutions to the exercises are provided in the appendix. Since the course handles requirements and modelling, often there is no unique solution, and the nuance of the solution is naturally missing from the solutions in the appendix. If you disagree with solutions, or think you have better, alternative solutions, please let us know! We would be happy to improve the reader based on your feedback.

Corrections during 2022-2023:

- Clarified the description of paths and infinite paths in the Kripke structure chapter.
- Fixed the solution to Exercise 11.4.
- Updated the LTL chapter to explicitly refer to infinite paths.
- Fix an example which used a channel with incorrect signature in Chapter 10.

Acknowledgements I would like to thank Rick Erkens, Lars van den Haak, Samar Jameel and Matthijs Bosch, Yousif Mir and Ruben Verhaegh for detailed comments on the earlier versions of the reader.

Chapter 1

Introduction

This reader accompanies the TU/e course 2IX20 Software Specification. The reader is based on earlier incarnations written by Julien Schmaltz and Anton Wijs. In this first chapter we briefly sketch the context of the course, and outline the learning objectives.

1.1 Context and motivation

The context of this course is the broad field of *Software Engineering*. Software engineering broadly covers all aspects of building complex but reliable computer systems.

When developing complex systems, there are a number of things that should be kept in mind. First of all, computer systems are typically developed to address a customer's need. In order to do so, it is important to find out exactly what the customer wants. Typically, this is done using requirements elicitation, which results in a set of *requirements* that establishes the needs of the stakeholders that have to be resolved by the software.

Once the customer's needs are clear, it needs to be decided how the system is going to be implemented. To aid in this process, we can *design* the software system that is developed. The design of the system is, essentially, a *specification* of the different aspects of the system, for instance how it is structured, or how a particular component should behave. This design allows us to reason about the correctness: does the design allow us to implement a system that covers the requirements?

Finally, the system that was designed is implemented in code. Once a component of the system has been implemented we can, on a local level, check whether it adheres to its specification for instance by using unit testing. Furthermore, we can determine whether the system as a whole implements the functionality as specified by the requirements.

The motivation of the course is that software quality significantly depends on the quality of the overall design, including structure and organization of the software. Bugs in software systems are frequently due to bad design choices made at the start of the development project.

Example 1.1.1. Consider the Miller&Valasek case about Jeep[1]. These two "hackers" managed to take control over a car. They exploited an error made in the design of the system. In short, the mobile phone network access point of the car was not protected. They could use this to upload malware to the car. An important design flaw is that once they were inside, they had root access by default and commands like "execute this software" were available at the command line. They could then upload new firmware to the car and take control of every single digital device. This is an example where decisions made early on in the project (such as underestimating cellular access protection, how to structure the overall system to protect firmware, what commands to allow or not) may introduce system flaws.

[1] Remote exploitation of an unaltered passenger vehicle, C.Miller and C. Valasek, DEFCON 23. See https://www.youtube.com/watch?v=OobLb1McxnI.

This course is focused on theories and techniques for writing *specifications* of computer systems, assessing the quality of the specification, and testing a system against its specification. We will cover the following aspects:

Informal specification. We will look at requirements, formulated in natural language, and discuss criteria that good requirements should satisfy. Furthermore, we discuss types of diagrams drawn from the UML standard that allow us to describe how the system is used, how it is structured, and how (a component of) the system behaves. These diagram types are typically used in a relatively informal way, and allow us to provide a high-level specification of (aspects of) the system, and to discuss and analyse them.

Formal specification. We use linear temporal logic to give a precise logical characterization of requirements. Automata are used precisely describe the behavior of a system. We also use tools to automatically verify that a system described using automata is guaranteed to satisfy requirements formulated in temporal logic.

Model-based testing. The formal specification of the behavior of the system and its requirements allows us to check that the specification is valid w.r.t. the requirements. However, the implementation is typically written separately, and the validation that was done on the specification does not carry over to the implementation. We therefore study model-based testing techniques that take our formal specification of the behavior, and determine whether the implementation conforms to the formal specification.

Developing specifications early on in a project is useful. Specifications are useful, because they force developers to carefully think about the systems under development, and they provide a way to assess that the systems ultimately meet their requirements. But in addition to this, a specification is useful as a *contract* between a customer and a software provider. In practice the specification is a formal contract that describes the product a customer is paying for, in particular what it should and should *not* do.

1.2 Learning objectives

The context and motivation behind the course inspire the following main learning objectives. At the end of this course, the student:

1. can describe existing software systems and software systems to be developed by means of informal and formal specification techniques;
2. can use tools to create specifications and to verify that specifications meet requirements expressed as formal properties;
3. can apply model-based testing techniques to relate software systems to their specifications.

1.3 Required knowledge & related courses

In this course it is assumed that students have:

- Basic knowledge of object-oriented programming. This is for instance treated in the courses *Programming* (2IP90) and *Programming methods* (2IPC0). The informal descriptions given in this course, based on UML, follow an object-oriented approach as well, and often have a direct translation into an object-oriented language like Java. In 2IPC0, class diagrams and sequence diagrams were already introduced in order to explain design patterns.
- Basic knowledge of first-order logic (predicate logic). This is treated in the course *Logic and Set theory* (2IT60). Essentially, first-order logic is one of the fundamental languages that can be used to precisely specify (parts of) systems. In this course, we introduce temporal logic. This is an instance of first-order logic that conveniently facilitates reasoning about systems. We can automatically verify whether a system satisfies properties expressed in temporal logic.

There are many other courses that provide useful backgrounds for topics treated in this course. We discuss some examples. The course *automata, language theory & complexity* (2IT90) introduces automata to represent languages (e.g. (non)deterministic finite automata to capture regular languages). The labelled transition systems and Kripke structures discussed in this course are also state based systems, but here they represent behavior of a system.

Data modelling and databases (2ID50) introduced ER-diagrams to give and abstract representation of the data model. They are closely related to class diagrams, that represent the static structure and behavior of the proposed system.

This course teaches techniques that are either used in, or extended upon in courses later in the programme. The *DBL app development* requires you to describe your design using (some of) the UML models introduced in this course. The *Software engineering project* (2IPE0) relies on your ability to elicit requirements, formulate good requirements, and create good software designs. Modelling skills are further explored in *Process modeling and simulation* (2IIH0). *Process theory* further studies *system behavior*, and ways to reason about it.

1.4 Practical organization

Current information about the course organization can be found on CANVAS. This also lists the lectures, instructions and assignments that belong to this course.

Chapter 2

Preliminaries

Before we are ready to dive into requirements and specifications, we should understand how the different formalisms that we introduce in this course are used. In this chapter, we therefore first look at the general topic of software quality, and the software development process.

2.1 Software quality

The content of the course is centered around the thesis that *high-quality specification is a prerequisite for high-quality software.*

There are many models that are used to characterize software quality. Early quality models were described by McCall et al. (1977) and Boehm et al. (1973). They were followed by models described in international standards such as ISO/IEC 9126 (ISO, 2001), which was later succeeded by ISO/IEC 25010:2011 (ISO, 2011). The interested reader is referred to the survey of the key software quality models by Al-Qutaish (2010).

These software quality models all describe a hierarchy that, on the top-level, contains a number of quality attributes, that may be subdivided further. On the lower levels, metrics are assigned to the quality attributes that can be used to assess the extent to which the software has a certain quality.

The ISO 25010 standard (ISO, 2011), for example, characterizes software quality using the following eight quality attributes:

Functional suitability The functionality delivered by the software should satisfy the user's needs. This encompasses, among others, functional completeness and functional correctness.

Performance efficiency The performance of the system relative to the amount of resources used. Covers, e.g., resource utilization.

Compatibility The system should be able to exchange information with other systems of components if it operates in a larger context. It should furthermore be able to perform its functions while sharing the same hard-/software environment with other systems/components.

Usability Users of the system should be able to effectively and efficiently reach specified goals. This includes, e.g., how easy it is to learn the system, and protecting the user from errors.

Reliability The degree to which a system, product or component performs its functions under specified conditions for a specified period of time. This, for example, includes availability and fault tolerance.

Security The system should protect information and data. This includes e.g., confidentiality and integrity.

Maintainability The degree in which the system can be changed effectively and efficiently; for this aspects such as reusability, modifiability and testability are considered.

Portability Covers how hard it is to port the system from one environment (e.g., hardware platform) to another.

In addition to these general standards, there are domain specific standards for software quality. For example, DO-178C in the US and ED-12C in Europe are software standards for the avionics industry. All software that is included in aircraft needs to be certified according to these standards. There are similar standards for the automotive sector (ISO 26262) and other standards dedicated to security (Common Criteria). Most of these standards require the following objectives to be met:

1. Test coverage of the software structure, e.g. condition/decision coverage;
2. Absence of dead code (code that is never executed);
3. Traceability from requirements down to code.

In short, they require developers to show that they have a high-level specification of their software and that this specification is correctly implemented. Assessment can be done through various means:

1. reviews and other manual inspections;
2. software testing;
3. formal methods.

In this course, we will consider these three aspects: (informal) inspections (of specifications), testing, and formal methods.

2.2 Software development processes

Software development projects are large and complex. To keep a project in check, it is crucial to have a well-defined process. In particular, we would like to keep track of *time*, meaning both the amount of effort spent (man-months) as well as planning; *information* which is mostly the documentation of the project; *organization*, how are people and teams organized; *quality* of the process and the product that is delivered; and *money*, i.e., what are the costs of the project (since time is money, typically these are mainly personnel costs).

In this course, we will largely ignore the reasons for using a particular software development process. However, we will see that the specification techniques that we introduce each have their own place in software development. To put these specification techniques in the context of the software development process, we first introduce the main software development process models.

The overview paper by Ruparelia (2010) discusses the most important software development process models. The paper is available via Canvas, and you need to be in the TU/e network in order to download its PDF.

At this point, you should *read Ruparelia (2010)* to familiarize yourself with the most important models.

2.2.1 Software development life cycle

A software development life cycle (SDLC) models the process that is used to produce software. There are several popular software development life cycle models, including e.g., the waterfall model, the V-model and the agile model.

Different life cycle models characterize different phases, e.g., some models are more detailed than others. Also, different sources use different names for the same stages. At their core, however, all software development life cycle models cover the following four stages (our description is inspired by Mauw et al. (2001)):

Requirements engineering In this phase, what the system should do and how it interacts with its environment are clarified. Note that requirements do not only refer to the *functional requirements*, that describe what the system should do, but also to *non-functional* requirements such as security.

Specification In this phase, the user requirements are analysed and refined to obtain the developer's view of the system (sometimes referred to as system requirements). The result of this state generally is a specification of the system. This specification typically combines a natural language description with a description (of parts of the system) in a formal specification language.

Design In this phase, the specification is refined. Decisions are taken, e.g., on how to partition the system into subsystems. Also, the interaction of subsystems and interfaces with clearly specified behavior are defined. The design is, in essence, a blueprint of the system to be implemented.

Implementation The design is realized (implemented). This means the design is turned into code. If hardware plays an essential part in the design, this is also part of the implementation.

We can say that the traditional SDLC models such as waterfall and the V-model are document-driven: they go through each of the phases in the development process. At the end of each phase a new pile of paper has been produced, and the process moves on to the next phase. Furthermore, these traditional models often do not have a lot of customer involvement in the software development process between requirements elicitation and acceptance of the product.

Today, a lot of software is developed using modern SDLC models such as agile. Agile[1] is a model of iterative development, in which individuals and interactions are more important than processes and tools. This manifests itself in short iterations in which development focuses on developing a limited set of features described using *user stories*. Essentially, requirements and the software evolve together in the iterations. These methodologies focus much less on a priori definition of requirements as well as documentation. Since it is hard to describe good requirements up front, this can be an advantage. At the same time, since requirements are not available, it can also be a disadvantage since it is difficult to estimate the cost of development.

2.2.2 The V-model for software development

The software specification techniques introduced in this course are mostly independent of the software engineering process that is used. However, to illustrate the place of each of the techniques, we use the V-model. We therefore fix the specific version of the V-model illustrated in Figure 2.1. The dotted box in the picture emphasizes the parts of the V-model considered in this course.

On the left, from top to bottom, the process starts with an analysis of the *requirements*, from which a *specification* is created that is refined into a *design* which is then *implemented* in some programming language. Our picture is missing other steps like integration in an entire system. Indeed, code is compiled into machine code and run by an operating system on some run-time environment. Going back from bottom to top, testing efforts go from unit tests, system tests, integration tests, to, ultimately, acceptance tests. Tests should here be understood with a general meaning of "Quality Assurance" or QA for short. Quality assurance involves much more than testing alone.

We will talk about *validation* when assessing the quality of one level, for example, unit testing specific components from the implementation. We will talk about *verification* when assessing quality between two different levels, for instance verifying that a design meets its specification.

[1] https://agilemanifesto.org

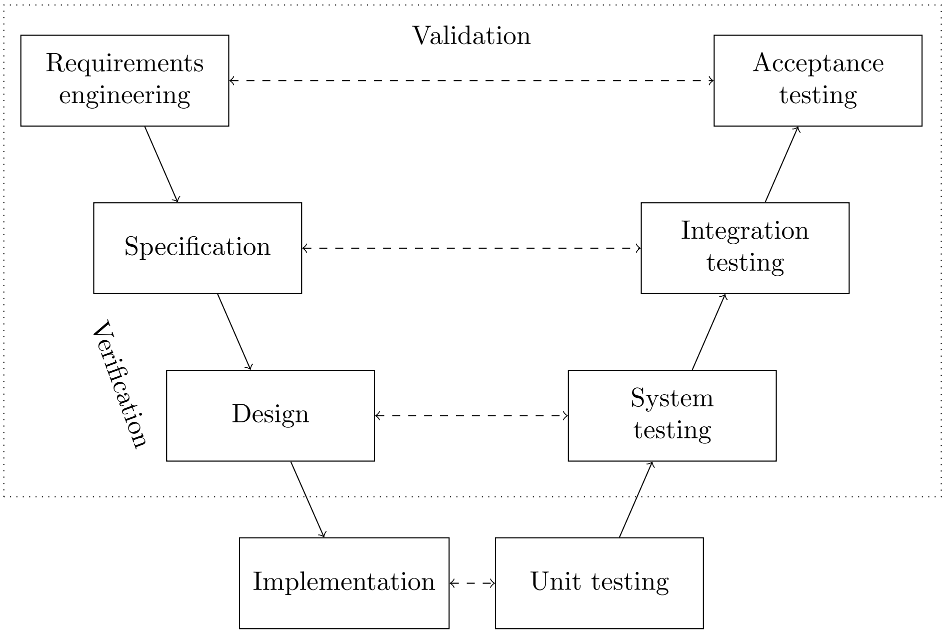


Figure 2.1: The V-model.

Chapter 3

Requirements

A *requirement* is a textual description of system behavior. A requirement describes in natural language, usually English, what a system is expected to do. This is a basic technique frequently used in practice. The content of this chapter is based on the ISO/IEC/IEEE standard 29148:2018 ISO (2018). You should be able to access the standard from the TU/e library, and a link to it is available from the Canvas page of this course.

A key section of the standard is "5.2 Requirements fundamentals". We do not assume you read the entire standard, but we do expect that you read the specific parts of the standard indicated in this chapter.

Note that requirements can be used all along the development life cycle. In this chapter, we consider the initial phase of a software development project. We consider requirements to form the first model of the system to be developed.

After reading this chapter, we expect:

- you can explain the place of requirements engineering in the software engineering process;
- you know the characteristics of good requirements;
- you can explain the semi-formal syntax used to write requirements;
- you know the characteristics of a good set of requirements;
- you can write a requirements document that satisfies the characteristics of a good set of requirements, and in which each of the individual requirements satisfies the characteristics of good requirements.

3.1 Definition and goal

A requirement is a *statement* which expresses a need and its associated constraints and conditions. It is a piece of text (written in natural language) describing a system, and does not require any deep technical knowledge. It is therefore understandable by most *stakeholders* involved in the project. Stakeholders generally are software providers, users, customers and suppliers.

The main objective for a set of requirements is as follows.

Requirements objective: to enable an agreed understanding between stakeholders.

Requirements are the initial step in specifying what is expected from the software system. By specifying them, the stakeholders reach an "agreed understanding" of the objectives of the system.

3.2 Requirements engineering

One of the challenging aspects of every software engineering project is to define the requirements. How do we reach an agreement on stakeholder objectives as described in the previous section? Requirements engineering is a challenge, and success depends on the degree to which we manage to properly describe the system that is desired by the customer, and communicate this description to the stakeholder.

The process of getting from the user needs to a requirements specification is called requirements engineering. This process consists of four steps:

1. Requirements elicitation
2. Requirements specification
3. Requirements verification and validation
4. Requirements management

3.2.1 Requirements elicitation

Requirements elicitation, sometimes also referred to as *requirements gathering* is the processes of getting the requirements of a system from the customers, users, and possibly other stakeholders. The key focus of this step is *understanding the problem*. The process is typically more involved than just asking the customer what the system should do. Therefore, in practice several techniques are used for requirements elicitation. Some common techniques are:

- **Interviewing** stakeholders. Representatives from each stakeholder group are selected for interviews to understand their expectations of the system and the tasks it should support. Interviews can either be open ended to understand the problem, or it can be structured. In a structured interview, questions are prepared to guide the interview. These can include, for example, the stakeholder's vision for the future, as well as alternative ideas. It is also important to ask for other sources of information. It may help to ask the interviewee to draw diagrams for easier communication.
- **Brainstorming** is a group technique intended to generate ideas. In brainstorming sessions, an experienced moderator and attendees are arranged around a table. During such a session, trigger questions are asked, and all participants write down their ideas related to the question. The main goal is to generate many new ideas. An example of a technique based on intensive brainstorming activities is Joint Application Development (JAD).
- **Observing**. The requirements engineer reads documents and discusses requirements with the user. Typically they shadow important potential users as they do their work, and ask the user to explain everything he or she is doing. It is important to record a video of a concrete working session for analysis.
- **Task analysis** is the process of analyzing the way people perform their jobs: the things they do, the things they act on and the things they need to know. Tasks are performed in order to achieve a goal. It is important to realize that the task analysis concentrates on the current situation, and is used as a starting point for a new system. Users will often refer to new elements of a system and its functionality. Scenario-based analysis can be used to exploit the new possibilities.
- **Scenario-based analysis** provides a more user-oriented view on the design and development of an interactive system. A scenario projects a concrete description of an activity that the user engages in when performing a specific task. It is important that the description is sufficiently detailed so that the design implications can be inferred and reasoned about. It is broader than task analysis.
- **Prototyping**. Here prototypes of the system are provided to stakeholders. This allows to effectively investigate possible solutions and get relevant information and feedback. The simplest kind of prototype is on paper (or digital sketches); in this form it is merely a set of pictures of the system shown to the user in sequence to explain what would happen. A more commonly used prototype is a mock-up of the system's UI, written in a rapid prototyping language. It does not normally perform any computations, access databases or interact with other systems. It may also be limited to just a particular aspect of a system.
- A **use case analysis** is done to determine the classes of users (actors) that will use the the system, and determine the tasks that each actor will need to do with the system. A use case is a typical sequence of actions that a user performs in order to complete a given task.

Note that the activities described above can be applied in different stages of the development process. For instance, interviewing and brainstorming are more suitable to early stages of the process, whereas prototyping and use cases analysis already require a good understanding of the user requirements, and are typically used at later stages where the requirements are refined. In this chapter we focus on the specification of user requirements in natural language. Use cases will be discussed in Chapter 4.

3.2.2 Requirements specification

Once the requirements are understood, the problem can be described. This is referred to as *requirements specification*. At this stage, all requirements (both functional and non-functional) are specified using software requirement models. Depending on the application domain, such models can be, for instance, ER-diagrams or (very high-level) class diagrams.

3.2.3 Requirements verification and validation

In the previous chapter we have already explained verification and validation in the software engineering process in general, and the V-model in particular. Requirements validation focuses on ensuring that the software correctly implements a specific function. Requirements verification ensures that the requirements specification itself satisfies its quality criteria. In particular, it should be ensured that requirements are consistent, complete and practically feasible.

3.2.4 Requirements management

In requirements engineering it is important to carefully keep track of the requirements, and analyze, document, track and prioritize them. Requirements are hardly ever stable, and a good requirements management process allows for changes in the requirements during the process. For this it is, among others, important to trace dependencies between requirements, and to keep track of the stakeholder objectives related to requirements.

3.3 Requirements Syntax

First, read section 5.2 of ISO (2018).

A requirement is written in natural language. Of course, natural language is inherently imprecise. Therefore, when defining requirements, we need to be very careful to, e.g., ensure that they are not ambiguous. One approach that helps to avoid common issues in requirements is to use a structured language with clearly defined syntax and vocabulary. Such requirements are said to be *semi-formal*. When it comes to writing requirements in natural language, we will only consider semi-formal requirements in this course. In Chapter 11 we will see how requirements can be formalized using temporal logic.

The ISO 29148 standard describes a clear syntax for writing requirements:

[**Condition**][**Subject**][**Action**][**Object**][**Constraint of Action**]

Requirements must at least have a Subject (what is the subject of the requirement) and an Action (what shall be done by the subject). The other parts are optional. The individual parts of the syntax have the following intention:

[**Condition**] condition under which the requirement is applicable. A condition generally starts with the word “When ...”. For instance, “When signal X is received” or “When the system is in mode Y”.

[**Subject**] describes the *actor*, for instance “the application” or “the system”.

[**Action**] describes the action or verb of the requirement. For instance “shall return to”, “shall send”.

[**Object**] the object of the action. This can for instance be a signal, a message or a particular state.

[**Constraint of Action**] a constraint restricts the action. For instance, giving a time limit can restrict the action. A typical word for such a constraint is "within".

We next consider a number of examples (taken from the standard).

Example 3.3.1. When signal x is received [**Condition**], the system [**Subject**] shall set [**Action**] the signal x received bit [**Object**] within 2 seconds [**Constraint of Action**].

This example illustrates the difference between [**Condition**] and [**Constraint of Action**]. A condition is a restriction on the subject or its environment. In the example, the condition that a particular signal is received. A constraint is a restriction on the action. In the example, that a change in a bit must take place within a given time limit. This difference is also illustrated by the following example:

Example 3.3.2. At sea state 1 [**Condition**], the Radar System [**Subject**] shall detect [**Action**] targets [**Object**] at ranges up to 100 nautical miles [**Constraint of Action**].

Finally, here is an example without a condition:

Example 3.3.3. The invoice system [**Subject**] shall display [**Action**] pending customer invoices [**Object**] in ascending order in which invoices are to be paid [**Constraint of Action**].

Exercise 3.1 Consider the requirements below. Are these requirements written according to the syntax described in the ISO 29148 standard? If not, rewrite the requirement according to the syntax. In both cases, clearly indicate the parts of the syntax ([**Subject**], etc.) in the requirement.

1. The system shall process at least 40 executing jobs at a time.
2. The system shall provide the means for the resource provider to see on which project this resource is working.
3. The system shall provide the means for the system admin to perform his actions on a computer with Windows XP, Mac OS X or Linux.
4. If one of the resources disappears while it was performing a job, the system should re-queue the job.

3.4 Characteristics

To help reach an agreed understanding between stakeholders, each individual requirement needs to possess specific characteristics. Before going into more details, we can already state that any requirement shall:

- solve a stakeholder objective,
- be about the (software) system,
- be verifiable.

Requirements must be related to the needs of stakeholders. That is, we should know why a given functionality of the system is needed. A common mistake is to express requirements about users. Furthermore, because requirements are used as specifications, we must have the possibility to actually check whether a system meets its requirements. In this section we describe characteristics that good requirements should have, both individually and within a set of requirements. The characteristics can be used to *validate* the requirements.

3.4.1 Characteristics of individual requirements

We here review the main characteristics provided to us by the ISO 29148 standard (ISO, 2018, Section 5.2.5) before discussing some examples.

Necessary Essential functionalities of the system should be included in the requirements. The system is not able to fulfil a stakeholder objective if this requirement is not implemented. Note that still, requirements may be prioritized in order of importance.

Appropriate Requirements need to specify *what* the system is expected to do without telling *how* the system will do it. Requirements must be independent of implementation details, such as algorithms or architectures.

For instance, a good requirement would be "The system shall sort packages" as opposed to "The system shall sort packages using the QuickSort algorithm".

Of course, one reason to be tempted to mention implementation details in a requirement might be to express that certain performance criteria need to be met. However, in such a case, an additional (non-functional) requirement should be defined addressing the expected performance, without enforcing the use of a particular algorithm. In that way, the requirements are robust against technical developments in the future, for instance if at some point, a more efficient algorithm has been developed.

Unambiguous A requirement should have exactly one *interpretation.* There should be only one way to understand a requirement. This is in general very hard to achieve using natural language (such as English), as natural language descriptions are often open to interpretation, depending, for instance, on the context. By keeping each requirement short and following the syntax proposed by the standard, it is feasible to write clear requirements.

Complete The requirement should contain all information that is necessary to understand it.

Singular To ensure clarity, a requirement must include no more than one statement. If more are needed, multiple requirements should be defined. This helps making requirements simple, easy to understand, and unambiguous.

Feasible A system satisfying the requirement can be implemented within set constraints. Constraints are typically time and money.

Verifiable We must have the possibility to prove (or disprove) that the system satisfies a given requirement. In other words, a requirement defines *what* a system should be able to do, and it is important that we can check the system.

Correct Requirements need to be upward traceable to stakeholder needs. We should keep in mind that the goal of requirements is to come to an agreement about what the system must do. A requirement therefore needs to be justifiable by linking it to a stakeholder goal or need.

Conforming The requirements should follow an agreed upon standard template and syntax. In this course, for conformance we expect our requirements to follow the syntax prescribed by the ISO standard.

Exercise 3.2 The following example requirements have been written by software engineers at CERN, European Laboratory for Particle Physics and pertain to the upgrade of an authorization management system database. Analyse these requirements according to the characteristics of individual requirements. If the requirements does not satisfy the characteristics, propose an alternative requirement, and also indicate how that requirement fits with the requirement syntax.

1. The information which is stored on the database can be accessed by any standard computer on the CERN network.
2. In order to obtain a CERN car sticker the person must have a valid CERN ID.
3. The opening of the software shall take less than 3-4 seconds under normal working conditions.
4. The user shall have access to a French - English dictionary. The user shall ask questions or propose suggestions for words translations by mailing to the administrators.
5. The software will be available 24hrs 365d/year.

3.4.2 Characteristics of a set of requirements

A *well-formed* set of requirements needs to possess the following characteristics:

Complete What is meant here by the standard is that the set of requirements contains everything pertinent to the system.

Consistent There should be no contradictory or overlapping requirements. Furthermore, the same terminology and measurements are used in the entire set of requirements.

Feasible The set of requirements is "realistic" in the sense that it can be achieved under life cycle constraints (costs, schedule, etc. . . .)

Comprehensible The set of requirements is written in such a way that the reader can understand what is expected of the system. In particular, remember that the requirements form an agreed understanding between stakeholders, and therefore, all stakeholders should be able to understand the set of requirements.

Able to be validated The set of requirements is such that it can be shown (proven) that an implementation that satisfies the requirements solves the stakeholder needs.

3.4.3 Requirement language criteria

The language used when writing requirements shall be as precise as possible in order to adhere to the requirements characteristics introduced in this section. In particular, vague terms shall be avoided. Here is a non-exhaustive list taken from (ISO, 2018, Section 5.2.7) of terms that *shall be avoided* in the requirements:

- **Superlatives:** such as best, most.
- **Subjective language:** user friendly, easy to use, cost effective.
- **Vague pronouns:** it, this, that.
- **Ambiguous adverbs and adjectives:** almost always, significant, minimal.
- **Open-ended, non-verifiable terms:** provide support, but not limited to, as a minimum.
- **Comparative phrases:** better than, higher quality
- **Loopholes:** if possible, as appropriate, as applicable.
- **Negative statements**
- **Passive voice:** shall be able

3.4.4 Attributes

Attributes are used to further explain and document a requirement. A requirement is a sentence following the proposed syntax. To review or analyse a requirement, it is important to know more about the context of this requirement. Here are some examples of attributes, partly taken from the ISO 29148 standard ISO (2018):

Identification A requirement should be uniquely identified.

Dependency It is important to identify dependencies between requirements. It should be the case that if a primary requirement is removed, the supporting requirement(s) can also be removed. When dealing with the initial specification, requirements can remain quite abstract and dependencies should be avoided.

Source Source denotes here the originator of the requirement. Knowing the source – and there may be multiple sources – is important to identify which stakeholder must be consulted for clarification, modification, or deletion.

Rationale The rationale behind each requirement should be captured. The rationale addresses the reasons why the requirement is required.

Priority The priority of each requirement should be identified.

On of the common ways to prioritize requirements is using the MoSCoW method. In this acronym, the letters stand for:

- Must have: these requirements are critical in order for the system to be a success.
- Should have: these requirements are important, but the system is usable if these requirements are not implemented. They can, e.g., be held back for a later release of the system.
- Could have: these requirements are desirable but not necessary. They will only be implemented when time/budget permit.
- Won't have: these requirements will probably not be addressed in the current project, but they could be interesting for future developments.

The standard proposes more examples. In practice, specific development teams or organization may have their own attributes.

3.5 Requirements document

The requirements are collected in a requirements document, such that the characteristics on individual requirements and those on sets of requirements are satisfied. With each of the requirements the indicated attributes are associated.

To clarify the actors that are involved with the system we separately include a list of actors with their goal in the requirements document. Note that actors can be humans, but also other systems that the system that we are developing interacts with.

3.5.1 Software systems requirements

Most of the traditional examples of requirements documents describe the requirements for a software system, where the actors involved with the system are human.

Example 3.5.1. Consider the following description:[1]

> The LIS is a software system that supports operations of a library. The LIS is capable of running on various commonly used operating systems (Windows, UNIX, Mac OS X, etc.) and is easy to maintain. The library lends books and magazines to borrowers, who are registered in the system, as are the books and magazines. The library handles the purchase of new titles for the library. Popular titles are bought in multiple copies. Old books, and magazines are removed when they are out of date or in poor condition. The LIS should support two kinds of users: librarians and borrowers. The librarian is an employee of the library who interacts with the LIS to manage library holdings. Librarians can create, update, and delete information about library titles and borrowers. They can also generate a report about borrowers and about titles in the LIS. A borrower is a library user who can make a loan of a copy of a title that is part of the library holdings. A borrower can also reserve a book or magazine that is not currently available in the library, so that when it is returned or acquired by the library, that person is notified. The reservation is cancelled when the borrower checks out the book or magazine or through an explicit cancelling procedure. Librarians and borrowers make requests and interact with the LIS through a library kiosk (a computer interface). Loans and returns of library material are processed using an ID scanner.

We write down requirements for this system systematically using three steps:

1. Identify the system
2. Identify the actors and their goal
3. Go through the text to try if each sentence gives rise to one or more requirements.

Our analysis is the following:

1. Identify the system: LIS software

[1] adapted from `http://www.swenet.org/Materials/86/req3-exercise.pdf`

2. Identify the actors and their goal:

   Librarian manage library holdings.
   Borrower reserve and borrow books.

3. We give some of the requirements for this system and explain how we obtained them. You are to identify the rest of the requirements in Exercise 3.3.

   The LIS is capable of running on various commonly used operating systems (Windows, UNIX, Mac OS X, etc.) and is easy to maintain.

   Here there are two possible requirements: (1) on which systems must be supported and (2) easy maintenance. If system compatibilities are an essential aspect, we have the following requirement:

   R1 The LIS system shall run on the following operating systems: Windows 8, Mac OS X from 10.11, and Ubuntu version 18.04.

   It is important to make the list of possible systems explicit to make the requirement verifiable. Easy maintenance is too vague a concept and cannot be a requirement. We can make this more concrete, e.g. in terms of costs.

   R2 The LIS system shall be maintainable at a yearly cost of at most €0.2 million.

   The library lends books and magazines to borrowers, who are registered in the system, as are the books and magazines.

   There are two types of "objects" in the system:

   R3 The LIS system shall support the registration of books, magazines, and borrowers.

   Note that at this stage, we do not know more specifically who can register books, magazines and borrowers.

   The library handles the purchase of new titles for the library.

   This does not say anything about the system. Should the purchase be done via the LIS system? We assume it does not, therefore, the bought items only need to be registered, which is already covered by R3.

As you can see in the example, although the description of the system is already reasonably precise, when writing down our requirements we still had to make many assumptions and add clarifications. During the software engineering process of a real system, of course, such assumptions and clarifications should be extracted from stakeholders, for instance using the requirements elicitation techniques we described before.

Exercise 3.3 Consider the description of the LIS in Example 3.5.1. In the example we already identified the system, the actors and their goal, and the first few requirements. Go through the rest of the text, and write down the requirements that correspond to it.

3.5.2 Requirements for embedded systems

Nowadays, systems do not only communicate with human users, but also with other systems or hardware. For example, in the domain of embedded systems or cyber-physical systems, we often have to receive inputs such as sensor values or commands, or send outputs to devices. In such situations, it is helpful to not only identify the actors involved with the system, but also identify the inputs and outputs. Here we use the interpretation that inputs are received from the environment, outputs are sent to the environment.

Example 3.5.2. Suppose we want to develop the control software of a hydraulic cylinder. A cylinder is used in many robotics or other automation equipments (like production plants) to move arms, objects, plates, etc. An example of a cylinder is shown in Figure 3.1.

We consider the following informal description of the system, from which we will extract requirements.

[2] Source: `https://en.wikipedia.org/wiki/File:Cutawayweldedcylinder544x123.jpg`, this image is in the public domain

Figure 3.1: Hydraulic cylinder[2]

The cylinder basically consists of a piston that is moving from two positions. The initial position will be called the zero position and the position reached when the piston has been pushed to the end will be called the end position. Below you will find a description of the control software of the cylinder. For brevity, we will simply call it the cylinder.

The cylinder moves from a zero position to an end position. To know its position, the cylinder has two sensors: one at the zero position and another one at the end position. The cylinder is moved by a valve that either pushes the cylinder from the zero position to the end position or pulls the cylinder from its end position to the zero position. LEDs display if the cylinder is at the zero position or the end position. The cylinder can be used by a number of commands. The cylinder produces a number of outputs that users can observe. The outputs tell users whether the cylinder is at the zero position or at the end position. The cylinder needs to first receive an "initialize" command after being powered on. When the cylinder is initialized, it will tell the user that it is at the zero position. The cylinder can then be moved by sending a move to end or a move to zero commands. To guarantee the safety of users and other equipments, the cylinder has an emergency stop button. Once this button is pressed, the cylinder will immediately stop moving. The emergency procedure will also re-initialize the cylinder to the zero position.

We first identify the actors with their goals from the informal description:

Operator Operate the cylinder.
Sensor0 Detect the cylinder's piston at the zero position.
Sensor1 Detect the cylinder's piston at the end position.
Valve Move the cylinder
LEDs Display the position of the cylinder.

Next we identify the inputs and outputs of the system.

- Inputs:

ID	Name	Rationale
IN000	inpInit	Receives initialization signals
IN001	inpAt0	Receives signals from Sensor0
IN002	inpAt1	Receives signals from Sensor1
IN003	inpMove0	Receives commands to move the piston to position 0
IN004	inpMove1	Receives commands to move the piston to position 1
IN005	inpStop	Receives emergency stop commands

- Outputs:

ID	Name	Rationale
OUT000	outShow0	Sends signals to LEDs to show that the piston is at position 0
OUT001	outShow1	Sends signals to LEDs to show that the piston is at position 1
OUT002	outMove0	Sends signals to the valve to move the piston to position 0
OUT003	outMove1	Sends signals to the valve to move the piston to position 1
OUT004	outReady	Makes observable if the system is initialized or not

Finally we can identify the requirements. Since we have already identified the inputs and outputs, we can refer to them directly in the requirements, making the requirements both more accurate and more concise. We identify the first two requirements. For the sake of brevity we do not explicitly present the sentence-by-sentence analysis of the informal description of the system. We do indicate how the requirements adhere to the syntax. Priorities are indicated after the identity of the requirement.

- The system needs to detect the piston's position and make it observable.

  R1 (S) When the inpAt0 signal is received from Sensor0 [**Condition**], the system [**Subject**] shall set [**Action**] the outShow0 bit [**Object**] within 1 ms [**Constraint of Action**].

  R2 (S) When inpAt1 signal is received [**Condition**], the system [**Subject**] shall set [**Action**] the outShow1 bit [**Object**] within 1 ms [**Constraint of Action**].

Exercise 3.4 Recall the description of the cylinder in Example 3.5.2. We already identified two requirements in the example. Write at least 5 additional requirements for this system.

3.6 Requirements in the Agile process

In modern development processes (like AGILE, SCRUM, XP, etc), the term "feature" is often used. Nevertheless, the goal of a feature is very similar to a requirement: document a user need.

In traditional software engineering processes such as the V-model, the requirements are first collected completely, before going on to the next phase of the software engineering process. However, in Agile development, a couple of user stories or scenarios are taken and fully developed in a short iteration. The system with the added functionality supporting the user story is released, and the process is repeated.

3.7 Conclusions

In this chapter, we presented a first specification method: requirements. Requirements are written in natural language using semi-formal syntax. They are very easy to use and are widely used in practice. At the very top level, requirements will specify the user needs. At that level of abstraction, requirements shall be understood by all stakeholders. The required technical background to read these requirements should be low. Later in the life cycle, requirements will be refined to software requirements, or sometimes even low level requirements. The goal of these refined requirements is to specify *to designers and programmers* what they need to implement. The level of detail is then much higher.

The main drawback of requirements is that they do not show any structure. It is a very long list (realistic systems will have thousands of requirements) of text. It also does not tell about what interactions the system should have with users. This is the purpose of *use cases*, also called user stories or scenarios, that we discuss in the next chapter.

Part I

Unified Modelling Language

The Unified Modelling Language (UML) is one of the most widespread graphical modelling languages used to model object-oriented systems. It offers a collection of 14 different diagrams that each model different aspects of the system to be modelled. Although it has been described in quite a lot of detail, the standard describing UML is (intentionally) vague in many places, allowing the user of the language to vary slightly in the precise semantics attributed to it. This is one of the reasons for its popularity, but it also makes it hard to exchange models between different tools.

In this course we focus on the most commonly used diagrams. We follow the description of these diagrams given in the book Seidl et al. (2015), which you should study in its entirety. A PDF version of the book is available for download through the TU/e digital library. A link to the book is available on the course's Canvas page.

The book is based on version 2.4.1 of the UML standard (Object Management Group, 2011). The latest version is 2.5.1 (Object Management Group, 2017), which has only limited differences with version 2.4.1.

In the next chapters of this reader we point out the chapters in the book that you should read, and offer exercises for the topics discussed in teh book.

If you want to create your own UML diagrams, for instance for assignments that are to be handed in, you should use the tool UMLet.[a]

Before we take a closer look at each of the UML diagrams, read Seidl et al. (2015, Chapters 1 and 2).

[a] https://www.umlet.com

Chapter 4

Use Case Diagrams

As discussed in Chapter 3, textual requirements are an easy specification method but fail at specifying interactions between a system and its users. A *use case* describes an interaction scenario and its possible alternatives. A *use case diagram* graphically depicts several use cases, their actors, and their relationships.

At the end of this chapter, you

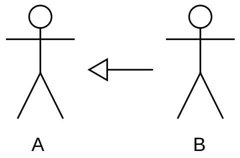
- can explain what a use case is;
- can explain what a use case diagram is;
- can reason about use cases and use case diagrams;
- can write detailed use case descriptions;
- can organize use cases in a use case diagram.

Study Seidl et al. (2015, Chapter 3).

The following exercises are selected from the e-learning environment that belongs to Seidl et al. (2015), and adapted to open questions. They are intended to improve your understanding of the basic concepts of use case diagrams.

Exercise 4.1 Model the following situation with a use case diagram: “An exam supervision is done by one professor and three tutors.”

Exercise 4.2 Consider the following use case diagram clipping.



For each of the following statements, indicate whether it is true or false.

1. *A* can execute the same use cases as *B*.
2. *A* inherits all of *B*’s associations.
3. *B* inherits all of *A*’s associations.
4. *B* can execute the same use cases as *A*.

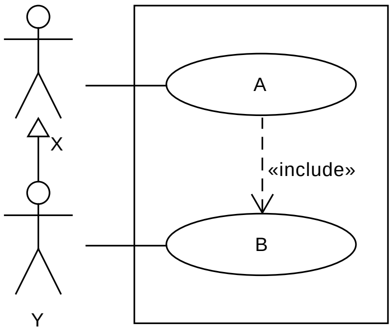
Exercise 4.3 Model the following situation with a use case diagram: “A travel booking is cancelled by a staff member or by the department head (who is also a staff member).”

Exercise 4.4 Model the following situation with a use case diagram: “A repair can be made by a master, a trainee or any other repair shop employee.”

Exercise 4.5 Model the following situation with a use case diagram: 'A teacher is conducting an interview with a student. In the course of that, the teacher always has to grade the student."

Exercise 4.6 Model the following situation with a use case diagram: 'A student edits their user profile. In the course of that they can also change their password if they like."

Exercise 4.7 Consider the following use case diagram.



1. Which are the valid combinations of actors that can communicate with use case A?
2. Which are the combinations for use case B?

Finally, we consider a number of situations that are described informally, for which we are to construct a use case diagram.

Exercise 4.8 Consider the following description of a transport company.

> The company owns a number of vehicles of different sizes which can transport goods. A client submits a request for transportation by specifying the size of the package to be transported, its source and destination. The distance between source and target determines the amount of time during which the vehicle will be en route. The company then sends an offer to the client by finding the first possible period during which a vehicle of an appropriate size is available. If the client agrees with the terms of the offer, it provides an account number and the authorization to withdraw the amount of the offer from the account. Upon a successful transaction with the bank (given the account information provided by the user), the amount of money will be transferred to the company's account and the company will schedule the transport as specified in the offer.

1. Identify the actors.
2. Create a use case diagram for the system.
3. Give a detailed description of at least one use case.

Exercise 4.9 Consider the following description of an ATM system.

> The ATM system can be started up or shut down by an operator. The operator can also register the ATM in the bank's central database. The customer can initiate a session with the ATM by inserting their bank card. Within a session, they perform a transaction, which can be withdrawing money from their account, checking the balance of the account with the bank, or depositing money into the account. Before completing the transaction, the customer needs to enter their PIN. The customer is only allowed to enter an invalid PIN at most three times.

1. Identify the actors.
2. Construct the use case diagram.
3. Give a detailed description of at least one use cases.

Exercise 4.10 Recall the description of the library information system (LIS) from Example 3.5.1. In this exercise, you have to create a use case diagram that describes the functionality of the LIS.

1. Identify the actors.
2. Identify the use cases (and their relations).
3. Identify the associations.
4. Give a detailed description of one use case.

Exercise 4.11 Recall the description of the hydraulic cylinder from Example 3.5.2.

1. Draw a use case diagram for the system.
2. Write a detailed description for all the use cases.

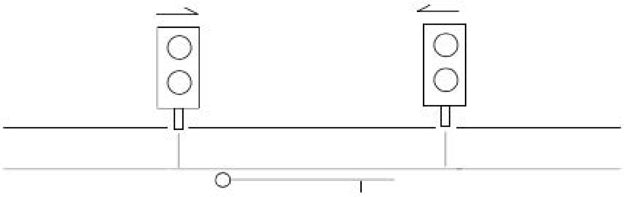


Figure 4.1: Overview of railway

Exercise 4.12 Consider the embedded system of a railway controller. The railway controller controls three tracks, two signals, and a crossing as depicted in Figure 4.1.

A signal shows either green or red. Each track is equipped with a sensor that shows the presence of a train and the direction of its movement. The supervisor of the control system may decide to change the status of the signal (from green to red or vice versa), or change the status of the crossing (from closed to opened or vice versa).

1. If the supervisor decides to change a signal to green, the controller should make sure that:
   (a) there is no train coming in the opposite direction towards the signal,
   (b) the crossing is closed, and
   (c) the signal in the opposite direction is red.

   If the above checks are successful, then the signal will be changed or otherwise the supervisor will be notified of the impossibility.
2. If the supervisor decides to change a signal to red, the controller should make sure that there is no train on the track before the signal. If this is the case, then the signal will be changed or otherwise, the supervisor will be notified of the impossibility.
3. If the supervisor decides to change a crossing to opened, the controller should make sure that:
   (a) there is no train on the track, where the crossing resides, and
   (b) the signals before and after the crossing are red.
4. Closing a crossing is always possible.

Create a use case diagram for this system.

Chapter 5

Class diagrams

Class diagrams support the specification of the structure of the system. They describe the elements of the system and the relations between them. In object-oriented programming, the elements of the class diagram are implemented as classes.

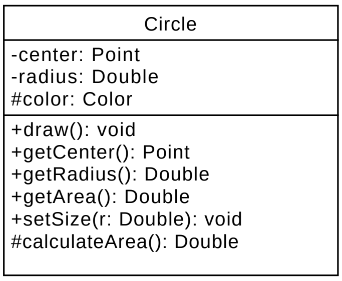
At the end of this chapter, you

- can explain what an object diagram is;
- can explain what a class diagram is;
- can reason about class diagrams;
- can relate object- and class diagrams;
- can construct class diagrams.

Study Seidl et al. (2015, Chapter 4).

Most of the following exercises are selected from the e-learning environment that belongs to Seidl et al. (2015), and adapted to open questions. They are intended to improve your understanding of the basic concepts of class diagrams.

Exercise 5.1 Consider the following snippet of a class diagram, that describes a *Circle* class.



1. What are the private attributes of the class (if any)?
2. What are the protected attributes of the class (if any)?
3. What are the public attributes of the class (if any)?
4. What are the private operations of the class (if any)?
5. What are the protected operations of the class (if any)?
6. What are the public operations of the class (if any)?

Exercise 5.2 Model the following situation with a class diagram: “An order is made with exactly one waiter, one waiter handles multiple orders”.

Exercise 5.3 Model the following situation with a class diagram: "We want to model that one tenant can rent multiple flats with different leasing contracts. Also, one flat can be rented by multiple tenants with different leasing contracts."

Exercise 5.4 Model the following situation with a class diagram: "Every restaurant has at least one kitchen, one kitchen is part of exactly one restaurant."

Exercise 5.5 Model the following situation with a class diagram: "An event may be part of another event."

Exercise 5.6 Model the following situation with a class diagram: "There are exactly two kinds of participants, namely members and guests. Guests are invited by members."

Finally, we consider a number of situations that are described informally, for which we are to construct a class diagram.

Exercise 5.7 Consider the following description of a singly linked list, taken from (Goodrich et al., 2015, p. 111)

> "A linked list, in its simplest form, is a collection of nodes that collectively form a linear sequence. In a singly linked list, each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list.
>
> A linked list's representation relies on the collaboration of many objects. Minimally, the linked list instance must keep a reference to the first node of the list, known as the head. Without an explicit reference to the head, there would be no way to locate that node. The last node of the list is known as the tail."

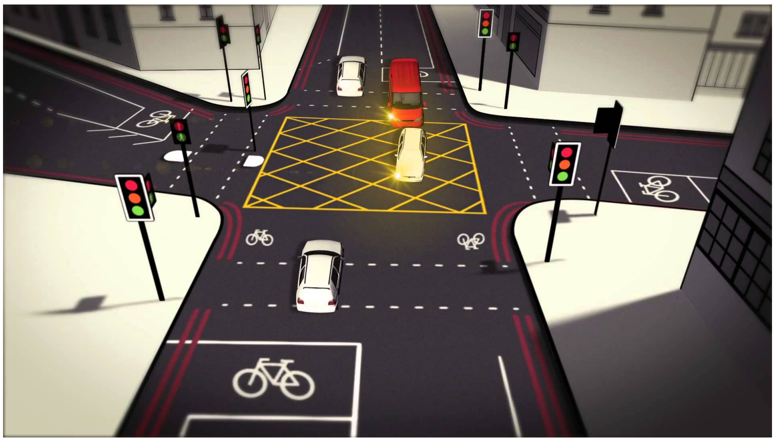
Give a class diagram that describes the singly linked list with elements of type Type. The class should allow to query the size of the list, i.e., the number of elements in the list, in constant time. It should be possible to check whether the list is empty. Also, the list provides access to the first and the last element in the list, elements can be added to the list at the beginning or at the end. The method removeFirst should remove and return the first element of the list.

Exercise 5.8 Recall the description of the transport company in Exercise 4.8. Identify classes, attributes, operations and relations in the description and the corresponding use-cases, and draw the class diagram for this system.

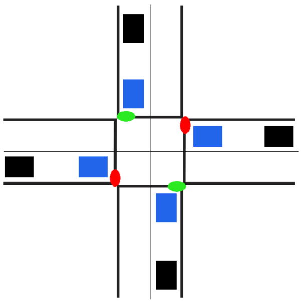
Exercise 5.9 Reconsider the description of the LIS from Example 3.5.1, of which you completed the requirements in Exercise 3.3, gave a use case diagram in Exercise 4.10. Identify classes, attributes, operations and relations in the description and the corresponding use-cases, and draw the class diagram for this system.

Exercise 5.10 Recall the railway crossing described in Exercise 4.12, and the corresponding use case diagram. Identify classes, attributes and operations and draw the class diagram for this system.

Exercise 5.11 Consider the embedded system of a traffic crossing controller. The controller controls two roads, one crossing, and traffic light signals on both sides of the crossing. An example of such a setup is illustrated below:



Each traffic light signal shows either green, yellow or red. Each road is equipped with two sensors in each driving direction: one in front of the corresponding traffic light to cross the crossing, and the other at a bigger distance from the traffic light. The first is used to detect whether a car is waiting to cross, the second is used to detect whether cars are currently approaching the crossing. This setup is sketched in the following picture, in which the first type of sensor is coloured blue, and the second type is black.



The control software should change the status of the signals under the right conditions, taking into account that one of the roads (the vertical one in the picture above) has priority over the other road, i.e., by default the signals for the vertical directions are green, and the other ones are red, as in the picture above. The control software should work as follows:

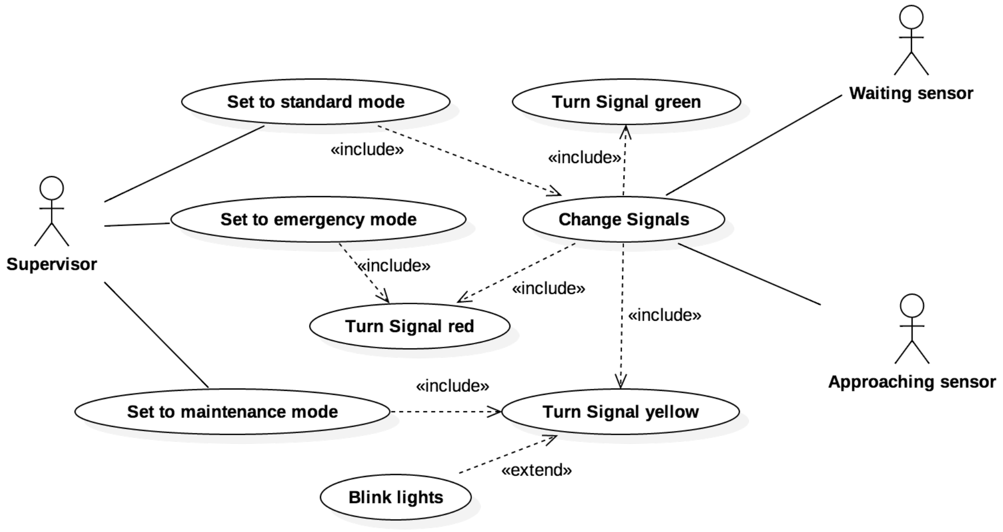
- If a car is approaching from the left or the right, the control software can start changing the signals, but only if no sensor on the vertical road is indicating that a car has (recently) passed.
- If a car is waiting on the left or the right, the control software should start changing the signals as soon as all sensors on the vertical road no longer indicate that a car has (recently) passed.
- As soon as all sensors on the horizontal road indicate that no car has (recently) passed, the control software should change the signals back to the default setup.

Changing the traffic lights involves performing the following steps in sequence:

- First change the signals that are currently green to yellow, and then to red;
- Then change the signals that were red before the changing procedure started to green.

Finally, the control software should be able to handle emergencies and maintenance sessions. For this purpose, a supervisor can change the mode of the control software. In emergency mode, all signals should be red. In maintenance mode, all signals should blink yellow. In standard mode, the control software behaves in the normal fashion as explained above. When switching to standard mode, the control software should switch the signals to the default setting.

To help you in the exercise, we provide the following use case diagram for the control software.



The use cases in the diagram above are abstract, we do not provide detailed use case descriptions, but this is enough to work on the exercise.

Identify classes, attributes, operations and associations and draw the class diagram for the control software system.

Chapter 6

State machine diagrams

State machine diagrams support the specification of the *behavior* of the system. They describe the states in which the system can be, and the way in which the system changes state.

At the end of this chapter, you

- can explain what an state machine diagram is;
- can reason about state machine diagrams;
- can construct state machine diagrams.

Study Seidl et al. (2015, Chapter 5).

The chapter in the book describes all different aspects of UML state machine diagrams and illustrates how (sequences of) events are executed. In Chapter 13 we will make the semantics of a subset of state machine diagrams precise. One thing the book does not clearly introduce is a configuration of a state machine. A configuration contains the set of states in which a state machine currently resides. Especially when a state machine diagram contains orthogonal states, it does not suffice to reason about a single state that is entered, as done in Table 5.1 in the book. Instead, we prefer the notation as in Table 6.1, which contains the same information as Table 5.1, using configurations instead of entered states.

Event	Configuration	x	y	z
Start	{A}	2		0
e2	{C, C1}		2	6
e1	{C, C1}	4		
e3	{C, C2}		0	3
e4	{E}	-1	2	
e1	{C, C2}		0	4
e5	{A}	-1	1	0

Table 6.1: Version of (Seidl et al., 2015, Table 5.1) using configurations instead of entered states. Only changes in variables are included in the table for readability, so, if a cell for a variable is empty, its value is the same as in the cell above.

In fact, it may sometimes be helpful to split the execution af the activities when an event is processed, to clearly see what happens. For the same example, we then obtain Table 6.2.

Some of the following exercises are selected from the e-learning environment that belongs to Seidl et al. (2015), and adapted to open questions. They are intended to improve your understanding of the basic concepts of state machine diagrams diagrams.

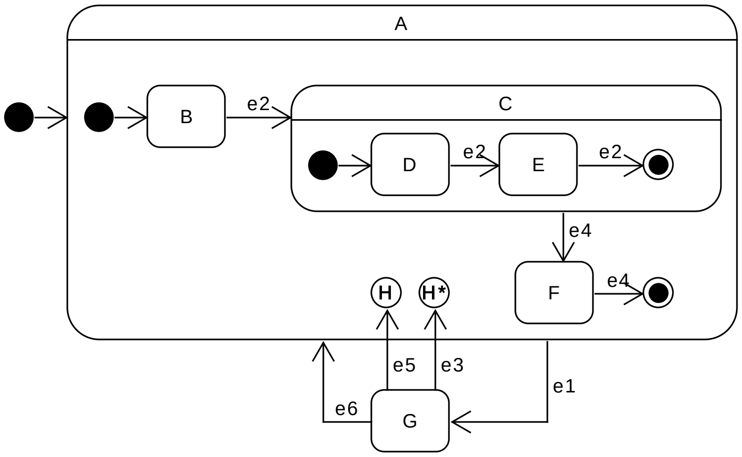
Exercise 6.1 Recall the state machine diagram from (Seidl et al., 2015, Figure 5.13), and assume the state machine is initially in configuration {S3}.

Event	Configuration	Comment	x	y	z
Start		start	2		
	$\{\mathsf{A}\}$	entry of A			0
e2		exit of A			1
		transition, $z = z * 2$			2
		entry of C		2	3
	$\{\mathsf{C}, \mathsf{C1}\}$	entry of $\mathsf{C1}$			6
e1	$\{\mathsf{C}, \mathsf{C1}\}$	$x = 4$ in $\mathsf{C1}$	4		
e3		exit of $\mathsf{C1}$			3
	$\{\mathsf{C}, \mathsf{C2}\}$	entry of $\mathsf{C2}$		0	
e4		exit of $\mathsf{C2}$	-1		
		exit of C		1	
	$\{\mathsf{E}\}$	entry of E		2	
e1		entry of C		2	4
	$\{\mathsf{C}, \mathsf{C2}\}$	entry of $\mathsf{C2}$		0	
e5		exit of $\mathsf{C2}$	-1		
		exit of C		1	
	$\{\mathsf{A}\}$	entry of A			0

Table 6.2: Version of (Seidl et al., 2015, Table 5.1) with individual rows for the individual activities. The new configuration is only shown on the line of the last activity.

1. What configuration is the state machine in after the occurrences of the events e2, e4, e4 (in that order)?
2. What configuration is the state machine in after the occurrences of the events e1, e3, e4?

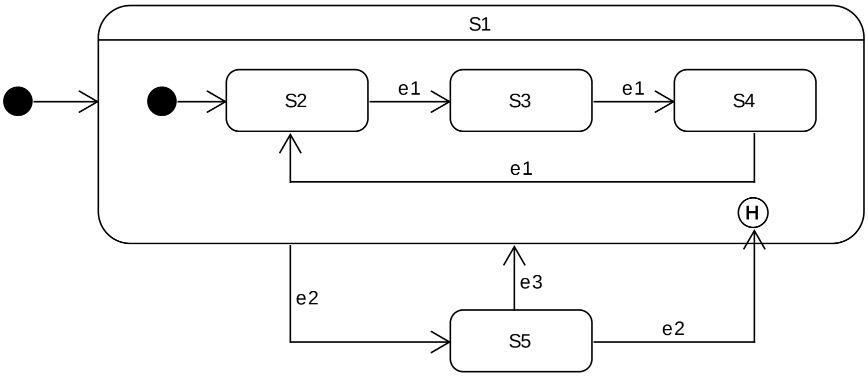
Exercise 6.2 Consider the following state machine diagram.



1. Suppose the state machine is in configuration $\{\mathsf{A}, \mathsf{C}, \mathsf{E}\}$. Explain what happens when first event e1 occurs, and next event e3 occurs; in what configuration is the state machine after executing these events?
2. Suppose the state machine is in configuration $\{\mathsf{A}, \mathsf{C}, \mathsf{E}\}$. Explain what happens when first event e1 occurs, and next event e5 occurs; in what configuration is the state machine after executing these events?
3. Suppose the state machine is in configuration $\{\mathsf{A}, \mathsf{C}, \mathsf{E}\}$. Explain what happens when first

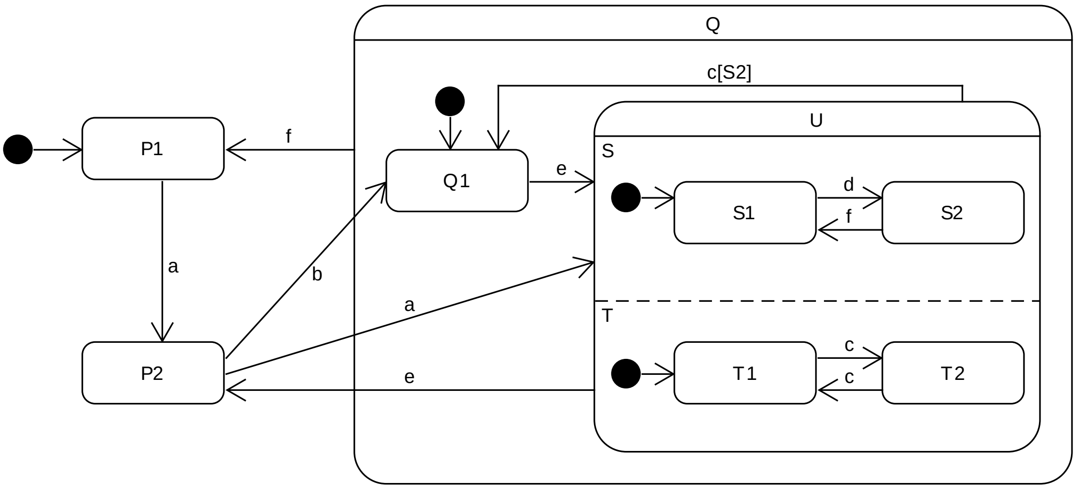
event e1 occurs, and next event e6 occurs; in what configuration is the state machine after executing these events?

Exercise 6.3 Consider the following state machine diagram.



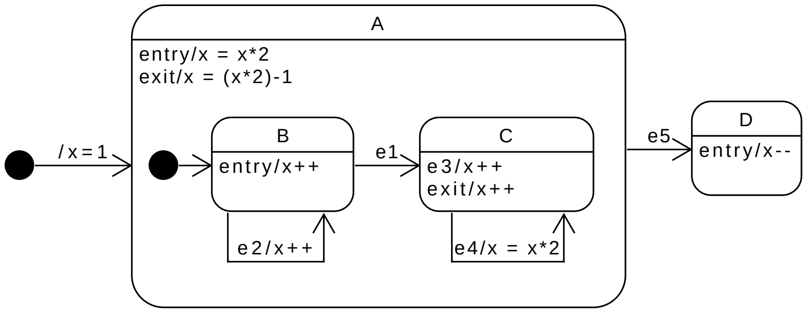
1. What configuration is the state machine in after the occurrences of the events e1, e2, e2, e1?
2. What configuration is the state machine in after the occurrences of the events e1, e2, e3, e1?
3. What configuration is the state machine in after the occurrences of the events e1, e1, e2, e2?
4. What configuration is the state machine in after the occurrences of the events e1, e1, e2, e2, e1, e1, e2, e2?

Exercise 6.4 Consider the following UML state machine diagram.



1. What are the types of the states P1, Q, U, and S.
2. Write down all configurations with respect to the root of the state diagram. (They do not necessarily have to be reachable).
3. What configuration is the state machine in after the occurrences of the events a, a, d?
4. What configuration is the state machine in after the occurrences of the events a, a, d, f?
5. What configuration is the state machine in after the occurrences of the events a, b, e, c?

Exercise 6.5 Consider the following state machine diagram.



1. What is the value of x after the occurrence of the event chain e2, e1, e4, e3, e5?
2. What is the value of x after the occurrence of the event chain e1, e3, e4, e4, e3, e5?
3. What is the value of x after the occurrence of the event chain e2, e2, e1, e3, e5?

Exercise 6.6 Recall the railway crossing described in Exercise 5.10, for which we constructed a class diagram.

The controller actually has a state. Draw a state machine diagram for the controller of the railway crossing.

Exercise 6.7 Recall the traffic crossing controller described in Exercise 5.11, for which we constructed a class diagram.

Draw a state machine diagram for the control software, by which you specify how it should react to environment triggers. Define suitable events to reason about triggers and activities. Try to create a diagram with at least two hierarchy levels.

Chapter 7

Sequence diagrams

Sequence diagrams are used to model the inter-object behavior, i.e., the communication between objects in the system. They can, for instance, be used to specify the link between a use case and the methods and classes defined in class diagrams.

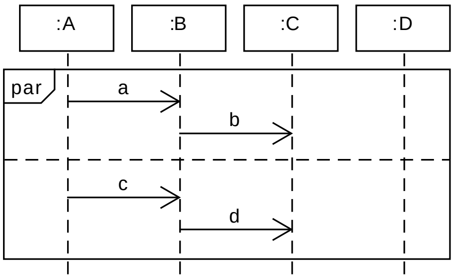
At the end of this chapter, you

- can explain what a sequence diagram is;
- can construct sequence diagrams;
- can relate sequence diagrams with class diagrams.

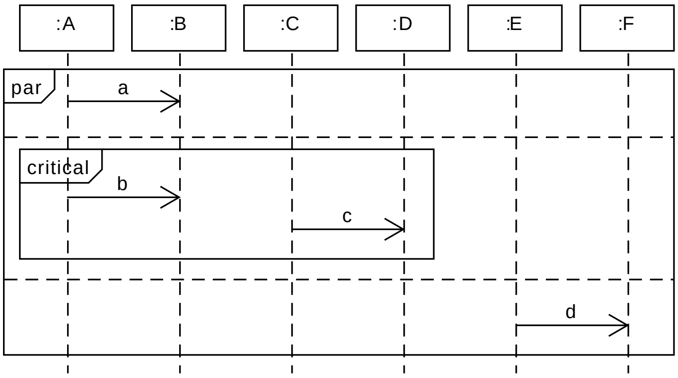
Study Seidl et al. (2015, Chapter 6).

Most of the following exercises are selected from the e-learning environment that belongs to Seidl et al. (2015), and adapted to open questions. They are intended to improve your understanding of the basic concepts of sequence diagrams.

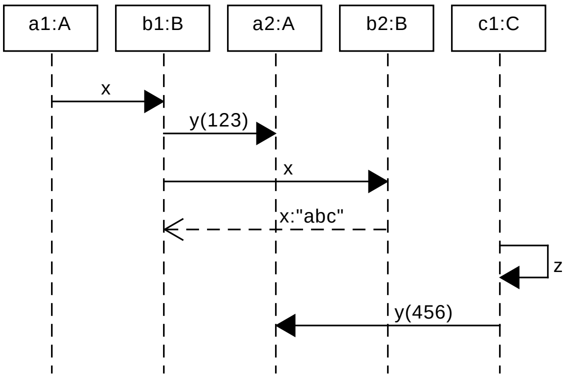
Exercise 7.1 Consider the following sequence diagram. List all traces that are possible.



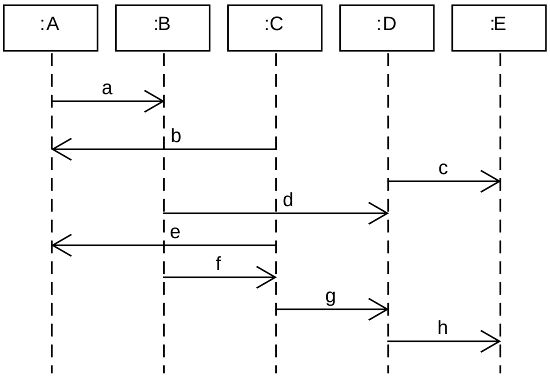
Exercise 7.2 Consider the following sequence diagram. List all traces that are possible.



Exercise 7.3 Consider the following sequence diagram. For each of the classes *A*, *B* and *C*, list the operations (with the types of the parameters and the type of the operation) that the class has according to the sequence diagram.



Exercise 7.4 You are given the following sequence diagram. Change the model such that *b* is always sent right after *a*.



The rest of the exercises are modeling exercises that continue on models that we have developed in previous chapters.

Exercise 7.5 Recall the railway crossing described in Exercise 4.12, for which you developed a class diagram and state machine diagram in Exercises 5.10, and 6.6.

Using the class diagram you produced and the use case diagram, draw a sequence diagram based on one of the use cases.

Exercise 7.6 Recall the traffic crossing controller described in Exercise 5.11, for which you developed a class diagram, and whose state machine diagram was modeled in Exercise 6.7.

Based on the use case diagram and the class diagram you constructed, draw a sequence diagram based on one of the use cases. Validate that the sequence diagram correctly relates to the class diagram.

Chapter 8

Activity diagrams

This chapter presents *activity diagrams*. An activity diagram is a kind of *behavioral specification*. It is quite suitable to specify workflows or complex algorithms.

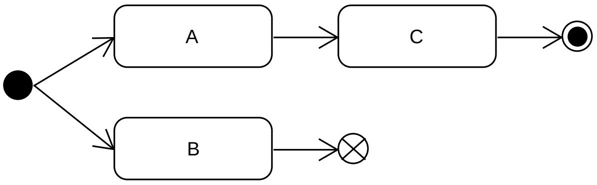
At the end of this chapter, you

- can explain what an activity diagram is;
- can construct activity diagrams;
- can reason about the flows described by activity diagrams.

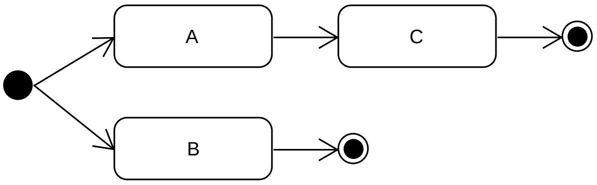
Study Seidl et al. (2015, Chapter 7).

Most of the following exercises are selected from the e-learning environment that belongs to Seidl et al. (2015), and adapted to open questions. They are intended to improve your understanding of the basic concepts of activity diagrams.

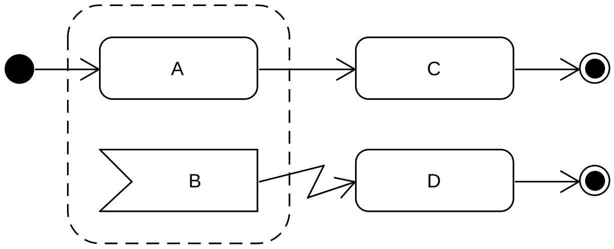
Exercise 8.1 You are given the following activity diagram. Give all action sequences (to completion) that are possible during one execution of the activity diagram.



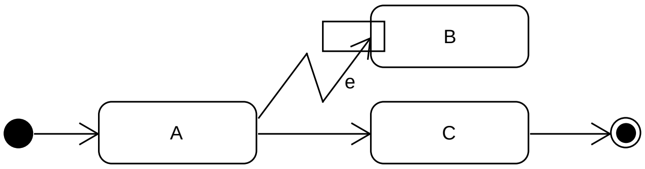
Exercise 8.2 You are given the following activity diagram. Give all action sequences (to completion) that are possible during one execution of the activity diagram.



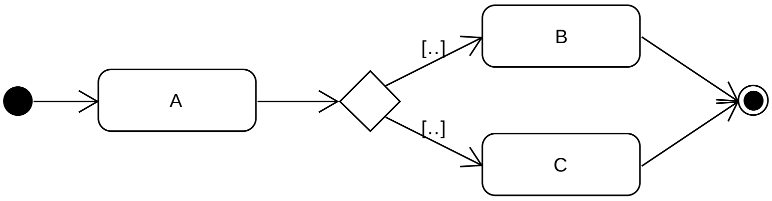
Exercise 8.3 You are given the following activity diagram. Give all action sequences (to completion) that are possible during one execution of the activity diagram. If an action is interrupted, include it in your sequence.



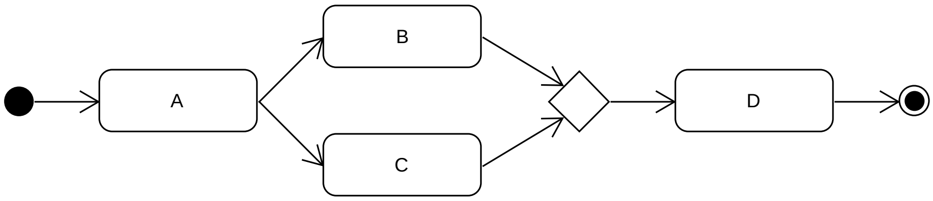
Exercise 8.4 You are given the following activity diagram. Give all action sequences (to completion) that are possible during one execution of the activity diagram. If an action is interrupted, include it in your sequence.



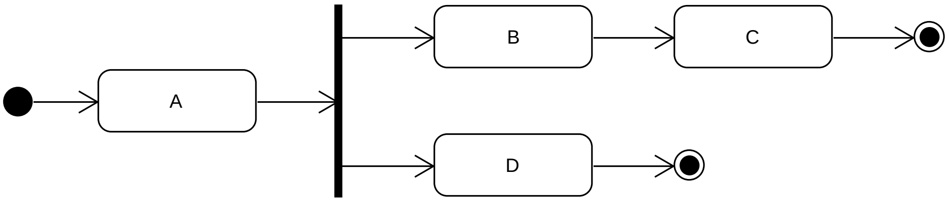
Exercise 8.5 You are given the following activity diagram. Give all action sequences (to completion) that are possible during one execution of the activity diagram.



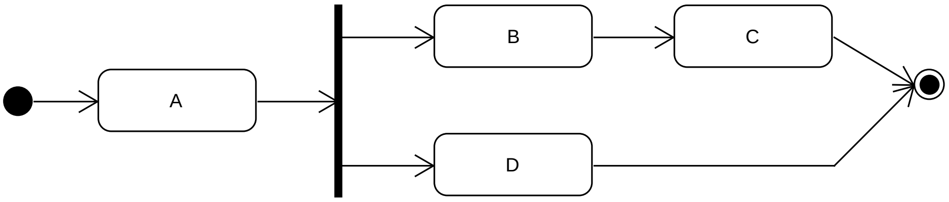
Exercise 8.6 You are given the following activity diagram. Give all action sequences (to completion) that are possible during one execution of the activity diagram.



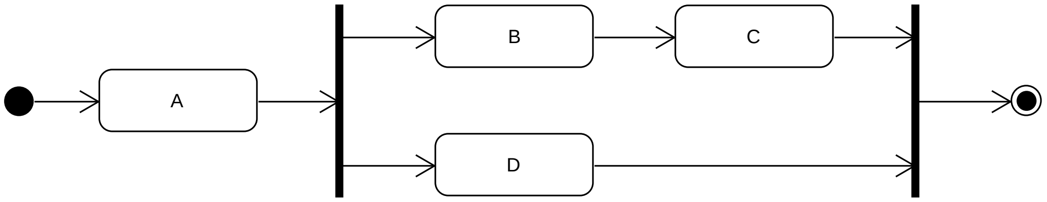
Exercise 8.7 You are given the following activity diagram. Give all action sequences (to completion) that are possible during one execution of the activity diagram.



Exercise 8.8 You are given the following activity diagram. Give all action sequences (to completion) that are possible during one execution of the activity diagram.



Exercise 8.9 You are given the following activity diagram. Give all action sequences (to completion) that are possible during one execution of the activity diagram.



The rest of the exercises are modeling exercises that continue on models that we have developed in previous chapters.

Exercise 8.10 Recall the railway crossing described in Exercise 4.12, for which you developed a class diagram, a state machine diagram and a sequence diagram in Exercises 5.10, 6.6, and 7.5.

Draw an activity diagram describing the complete algorithm of the controller. The controller takes a command from the supervisor. Commands are open, close, turn green and turn red. The activity diagram describes what the controller should do for each command.

Exercise 8.11 Recall the traffic crossing controller described in Exercise 5.11, for which you developed a class diagram, and whose state machine diagram and sequence diagram were modeled in Exercises 6.7 and 8.11.

Identify a scenario (say when the supervisor sets the software in “Maintenance Mode” or “Standard Mode”), then identify all the activities/actions that the control software should perform and create an Activity Diagram based on the chosen scenario and activities that are part of that scenario.

Part II

Formal specification and verification techniques

Before you start with this part of the course, make sure you read Seidl et al. (2015, Chapters 8 and 9). Chapter 8 gives a number of examples where the different models discussed in this course are integrated and used to model a system. You should carefully study this chapter. Chapter 9 discusses some further topics related to UML.

As we have seen in previous chapters, the Unified Modelling Language (UML), although very extensive, does not always have a clear meaning. If one wants to be able to precisely specify the behavior of systems, and either show that the (specification of) the system satisfies its requirements, or test whether the implementation of a system is consistent with its specification, we will need to have a more precise specification.

In this second part of the course, we focus on specification techniques that are precisely specified. In Chapter 10 we first describe a high-level language, Promela, that can be used to describe the behavior of (parallel) systems. In the same chapter we introduce the tool Spin that can be used to formally verify requirements of Promela models. The semantics of Promela models can be described using Kripke structures, that we formally describe in Chapter 9. Requirements can be formally specified using temporal logic. In Chapter 11 we introduce Linear Temporal Logic (LTL). We will explain when a Kripke structure satisfies an LTL property. Kripke structures focus on the properties that hold in a specific state. This model is well-suited for temporal logic model checking. For many other purposes, it is natural to focus on the interactions instead. In Chapter 12 we show how the behavior of systems can be described using labelled transition systems (LTSs), in which the transitions are labelled instead of the states. We next show that we can give a precise semantics of a subset of UML state machine diagrams using LTSs in Chapter 13, linking the informal specification techniques of UML to the formal techniques in this second part of the course.

Finally, in Chapter 14 we introduce a model-based testing technique based on input-output conformance that allows to automatically generate test cases for specifications described as labelled transition systems.

Chapter 9

Kripke Structures

In the previous chapters we introduced several UML diagrams. Even though the meaning of these diagrams is relatively well-defined, the UML standard leaves a lot of room for interpretation. Also, the standard defines some "variation points", where the user can actually assign their own meaning to a diagram. Furthermore, the models are *informal* in the sense that their notions and their semantics are only defined in plain English, and not *mathematically.* In the rest of the reader, we introduce models of software systems that are mathematically precisely defined. We will use these models to specify software systems. Thanks to their well-defined mathematical semantics, we will be able to *reason* about these models using mathematical logics. Logics are discussed in Chapter 11.

In this chapter we discuss Kripke structures. These are basic formalisms that can be used to precisely express the semantics of a model. In subsequent chapters, we will give a textual language that can be used to describe Kripke structures, and use these models and their associated tooling to reason about the behavior of systems.

At the end of this chapter, you

- can describe a system using Kripke structures;
- can identify states, transitions, and paths.

When defining precise semantics of a model (such as the Promela models described in Chapter 10) there are two main formalisms that are commonly used. They differ in their focus. Labelled transition systems describe the behavior of systems by focusing on the externally observable behavior. In this model, transitions are labelled by actions, and states are not labelled. We will study LTSs in detail in Chapter 12. In contrast, Kripke structures focus on the state of system, by labelling states with atomic propositions, and not labelling the transitions. In this chapter, our focus is on the latter.

9.1 Kripke structures

Kripke structures are state-transition systems in which the states are labelled. Atomic propositions are used to represent *state* information, that is, information about the current state of the system. This can be, for instance, the state of registers of a computer or the current value of program variables. For verification purposes, atomic propositions can help formulate requirements about a Kripke structure.

Definition 9.1.1 (Kripke structure)**.** A Kripke structure is a tuple $(S, s_0, \rightarrow, AP, L)$, where

- S is a set of states;
- $s_0 \in S$ is the initial state;
- $\rightarrow \subseteq S \times S$ is a transition relation;
- AP is a set of atomic propositions; and

- $L\colon S \to 2^{AP}$ is a labelling function that assigns a set of atomic propositions to every state.

Typically we assume that the transition relation is total, i.e., every state has at least one outgoing transition. This, in particular, will simplify the definition of the semantics of linear temporal logic in Chapter 11. A Kripke structure is finite if S, $\to$ and AP are finite. As before, we write $s \to s'$ for $(s, s') \in\to$.

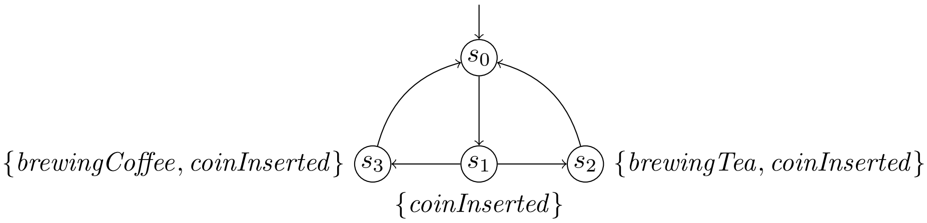

s_0
s_1
s_2
s_3
$\{brewingCoffee, coinInserted\}$
$\{brewingTea, coinInserted\}$
$\{coinInserted\}$

Figure 9.1: Kripke structure of a simple coffee machine producing either coffee or tea.

Example 9.1.1. Consider the example in Figure 9.1. This represents a system similar to the coffee machine in Figure 12.1, but as a Kripke structure. The set of states in this Kripke structure is $S = \{s_0, s_1, s_2, s_3\}$, with s_0 the initial state. The transitions relation $\to = \{(s_0, s_1), (s_1, s_2), (s_1, s_3), (s_2, s_0), (s_3, s_0)\}$. The atomic propositions $AP = \{coinInserted, brewingTea, brewingCoffee\}$. The labelling function is

- $L(s_0) = \emptyset$
- $L(s_1) = \{coinInserted\}$
- $L(s_2) = \{coinInserted, brewingTea\}$
- $L(s_3) = \{coinInserted, brewingCoffee\}$

A requirement about the state of this machine, that we could check using the atomic propositions assigned in the Kripke structure might be “The machine shall always have received a coin before brewing coffee”. In Chapter 11 we will explore the topic of formally defining properties about a system represented as a Kripke structure further.

Example 9.1.2. As another example, suppose we have two threads in a program that want to access some shared resource. The resource must be accessed under mutual exclusion, meaning that only one process can access it at the same time. Therefore, each of the processes can be in a non-critical section, where it does not need access to the resource, it can try to get access to the critical section, and eventually it can be in its critical section. We model this protocol using a Kripke structure, where states are labelled with atomic propositions $trying_i$ and in_i for $i \in \{0, 1\}$. The Kripke structure is shown in Figure 9.2.

Exercise 9.1 Consider an oven. This oven has:

- a button to start cooking
- a button to stop cooking
- a button to open the door
- a sensor to detect when the door is closed or opened
- when the oven stops cooking, it rings a bell

Create a Kripke structure that models the oven. Identify a set of atomic propositions such that you can express the following properties:

- The oven is never cooking with its door opened.
- The over starts cooking only after the start button is pressed.
- After starting to cook, the oven eventually stops cooking.

Exercise 9.2 The game of tic-tac-toe[1] is played on a 3-by-3 grid. The ×-player and the ○-player

[1] See `https://en.wikipedia.org/wiki/Tic-tac-toe`.

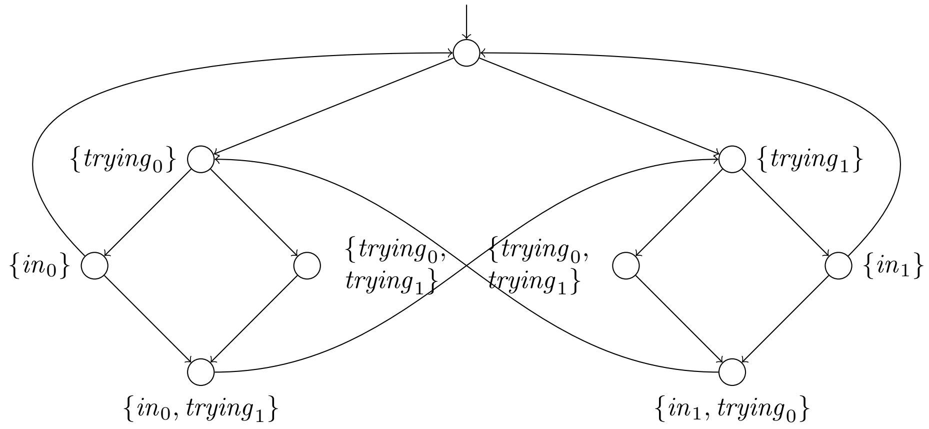


Figure 9.2: Kripke structure modelling mutual exclusion between two processes.

alternatingly place their respective symbol of $\{\times, \bigcirc\}$ on an empty cell in the grid. A player wins upon placing their symbol in such a way that their symbol appears thrice in the same row, column or diagonal. Model this game as a Kripke structure.

9.2 Paths

A Kripke structure describes all states of the system, and how the system can change between these states. Runs of system described with a Kripke structure are described using *paths*.

Definition 9.2.1 (Path)**.** Let $(S, s_0, \to, AP, L)$ be a Kripke structure. A *path* starting in a state $s \in S$ is a possibly infinite sequence of states $\pi = s, s_1, s_2, \ldots$, such that $s \to s_1$, and for all $i > 1$, $s_{i-1} \to s_i$. We write *paths*(s) to denote the set of all (finite and infinite) paths starting in s.

Since Kripke structures have a total transition relation, i.e., every state has an outgoing transition, there are no deadlocks in a Kripke structure. Therefore, we often only consider infinite paths. The set of infinite paths starting in state s is denoted *paths*$^\omega(s)$.

Definition 9.2.2 (Notation for paths)**.** Let $\pi = s_0, s_1, s_2, \ldots$ be path. We use the following notation:

- $\pi[i]$ denotes the state at position i, i.e., it is state s_i. In particular, $\pi[0]$ is the first state of the path.
- $\pi[i \ldots]$ is the suffix of the path starting at position i. For example, $\pi[1 \ldots]$ is the path π from which the first state has been removed. Note that $\pi[0 \ldots] = \pi$.

Example 9.2.1. Consider the Kripke structure in Figure 9.1. The following are examples of paths in this Kripke structures:

- s_0, s_1, s_3, s_0
- $(s_0, s_1, s_2)^\omega$ is the infinite path cyling through these three states infinitely often.
- $(s_0, s_1, s_2, s_0, s_1, s_3)^\omega$.

9.3 Applications

Kripke structures are used to verify (model check) systems. Kripke structures are traditionally verified using logics such as LTL, CTL and CTL*. In this course we restrict ourselves to introducing LTL, see Chapter 11, and we will only cover its syntax and semantics. We will not go into the details of algorithms used to verify LTL properties on a Kripke structure. Algorithmic details of some of these logics are also covered in the master's course algorithms for model checking.

Of course, describing a system directly using Kripke structures is tedious and error prone, because they tend to become large and unwieldy. Therefore, typically they are generated from high level languages such as Promela, which we introduce in Chapter 10.

Chapter 10

Promela and the SPIN model checker

In the first part of this course, we have used UML models to specify systems. In particular, we used UML state machine diagrams for describing behavior. However, the state machine diagram semantics is not clearly defined. In the previous chapter we have seen that the behavior of a system can be described using Kripke structures. However, descriptions using Kripke structures quickly become too complex to understand and maintain. It is therefore infeasible to specify the Kripke structure directly. This issue can be addressed by using a higher-level language that has a precisely defined semantics in terms of Kripke structures.

In this chapter we introduce the language PROMELA, which is a high-level language that can be used to describe parallel, communicating processes. A PROMELA model essentially has a Kripke structure semantics, but for the sake of synchronization and displaying the state space of a model, we typically include the statements on the transitions.

The SPIN model checker[1] supports the automatic verification of PROMELA models using assertions and LTL properties (see Chapter 11). Note that SPIN is primarily used as a command-line tool, and our instructions will focus on that. A graphical user interface that could be used is JSPIN,[2] but the tool has not seen updates in a long time. The book Ben-Ari (2008) serves as a nice introduction to SPIN as a language and tool, and it uses JSPIN for its explanations.

Model checking, the technique used by SPIN is an example of a formal method. Formal methods are the applied mathematics for modelling and analyzing computer systems. They offer a large potential for an early integration of verification in the design process, so we can detect fundamental issues in the design of a system before implementation. They provide more effective verification, i.e., they typically cover a larger part of the system than testing. Finally, studies show that the application of formal methods can actually reduce the time needed for software development, and the number of bugs found in production. The use of formal methods is highly recommended by the IEC, FAA and NASA for safety-critical software. The SPIN model checker has, for instance, been used to verify the Mars rover Curiosity, see Holzmann (2014).

At the end of this chapter, you

- can model simple, communicating systems using PROMELA;
- can detect deadlocks in PROMELA models using SPIN;
- can use assertions to formalize safety properties in PROMELA models;
- can verify whether a PROMELA model satisfies such properties;
- can explain a counterexample if the model contains a deadlock or violates an assertion.

For background material on SPIN you are referred to Ben-Ari (2008). Also, the SPIN manual[3] provides a wealth of information. Installation and usage instructions for SPIN on different platforms

[1] https://spinroot.com

[2] http://www.weizmann.ac.il/sci-tea/benari/software-and-learning-materials/jspin

[3] https://spinroot.com/spin/Man/Manual.html

are available on the Canvas page of the course. The examples in this chapter can by edited with any text editor.[4] All of the commands can be run from the command line, assuming that the `spin` executable is installed in the PATH. If you insist on using a graphical frontend, consider using the jSpin interface, which is also used in Ben-Ari (2008).

10.1 Specifying Promela models

A Promela model consists of type declarations, channel declarations, global variable declarations, process declarations, and the specification of an initial process. Since a Promela model represents a *finite* model, we need to make sure that data, channels, processes and process creation are all finite. In this section, we first introduce the primitives needed to model shared memory processes, that is, processes that do not communicate using explicitly defined synchronous or asynchronous channels. We will discuss channels in Section 10.4.

10.1.1 A simple example

Consider the following example definition of a process `Alpha` in Promela. The process has formal parameters `x` and `y`, which have type **`int`**. It declares local variable `z` of type **`int`** and initializes it to `1`. Keyword **`skip`** denotes the empty statement. Subsequently, it assigns `x` to `2`. Finally, the expression `x>2 && y>z && z==1` is executable if it evaluates to true.

```
proctype Alpha(int x; int y) {
    int z = 1;
    skip;
    x=2;
    x>2 && y>z && z==1;
    skip;
}

init {
    run Alpha(5,3);
}
```

Using the **`init`** keyword, we can specify the behavior of a process that is active in the initial system state. An **`init`** process has no parameters. The **`init`** process is typically used to initialize global variables, and to instantiate other processes through the use of the **`run`** operator. In this case, the process `Alpha` is initialized with value `5` for its parameter `x` and `3` for parameter `y`. The **`run`** operator creates a new process which runs asynchronously with the process that are already active at the point where **`run`** is called. Since **`run`** is called from the initial process **`init`**, in this example we end up with two running processed.

We can run a random simulation of a Promela model using Spin as follows.

```
    spin alpha.pml
```

If we run this simulation, we get the following fairly cryptic output.

```
timeout
#processes: 2
    3:    proc  1 (Alpha:1) alpha.pml:5 (state 3)
    3:    proc  0 (:init::1) alpha.pml:11 (state 2) <valid end state>
2 processes created
```

This does not give us a lot of information. It only tells us that two processes were created (**`:init:`** and `Alpha`), and that one of the two processes triggers a timeout. In order to understand why one of the processes times out, let us get some more information.

There are three options that can be passed to Spin in order to get more information in the output that are relevant at this point:

[4] We suggest to use a text editor with syntax highlighting for Promela, e.g. Visual Studio Code.

- `-p` prints all statements that are executed.
- `-g` prints all global variables and their current value.
- `-l` prints all local variables and their current value.

Our example does not have global variables. We show the output with the other two options enabled.

```
spin -p -l alpha.pml
```

This produces the following output, that shows that first the initial process **`init`** with process id 0 (`proc 0`) is created. This process in turn creates the process `Alpha` with process id 1. `Alpha` initializes variable `y` to `1`. From this, it executes a **`skip`** to state `1`, shown on line `2:` in the output. Subsequently it sets `x` to `2`, after which a **`timeout`** appears.

```
  0:    proc  - (:root:) creates proc  0 (:init:)
Starting Alpha with pid 1
  1:    proc  0 (:init::1) creates proc  1 (Alpha)
  1:    proc  0 (:init::1) alpha.pml:10 (state 1)       [(run Alpha(5,3))]
  2:    proc  1 (Alpha:1) alpha.pml:3 (state 1) [(1)]
  3:    proc  1 (Alpha:1) alpha.pml:4 (state 2) [x = 2]
                Alpha(1):x = 2
      timeout
#processes: 2
  3:    proc  1 (Alpha:1) alpha.pml:5 (state 3)
                Alpha(1):z = 1
                Alpha(1):y = 3
                Alpha(1):x = 2
  3:    proc  0 (:init::1) alpha.pml:11 (state 2) <valid end state>
2 processes created
```

The output now does provide us with enough information to understand why the timeout is triggered: After setting `x` to `2`, process `Alpha` indicates the timeout at condition `x>2 && y>z && z==1`, as, at that point in the execution, it holds that `x==2`, so the condition cannot be executed, and execution halts at that point in the process.

10.1.2 Processes

The example from the previous section already introduces a lot of Promela's syntax. Let us first look at the definition of processes. A process in Promela is defined as a **`proctype`** that consists of a name, a list of formal parameters, local variable declarations and a body. The body consists of a sequence of statements, in which global variables can be accessed.

In Promela, processes execute concurrently with all other processes, independent of their relative speed. Communication between processes can be done using shared variables (global variables) or using channels. When executing processes in parallel, there may be several processes of the same type. In that case, each process has its own local state, that consists of the process counter (or program counter), which indicates the location of the process within the **`proctype`** body, as well as the values of the local variables.

Processes can be created in two ways. First of all, a process can be created using the **`run`** statement, which returns a process id, at any point in the execution. Processes start executing after the run statement has created them. Alternatively, processes can be created by adding **`active`** in front of the **`proctype`** declaration. Finally, there is a special process **`init`**. Both **`init`** and **`active`** are created at the start of the execution of the process.

Every process has a process id, which can be accessed from the process using `_pid`. The creation of processes and the use of `_pid` is illustrated in the following example.

Example 10.1.1. Consider the following process.

```
active proctype Hello() {
    printf("Hello,␣my␣pid␣is:␣%d\n", _pid);
}
init {
    int lastpid;
    printf("init,␣my␣pid␣is:␣%d\n", _pid);
    lastpid = run Hello();
    printf("last␣pid␣was:␣%d\n", lastpid);
}
```

During an execution, three processes are created: one for the initial process **init**, one for the instance of `Hello` that is created because it is marked **active**, and another one because of the call to `Hello` from the initial process.

If we perform a random simulation of this process (stored in `hello.pml`), using `spin -p hello.pml`, we get, e.g., the following output.

```
  0:    proc  - (:root:) creates proc  0 (Hello)
  0:    proc  - (:root:) creates proc  1 (:init:)
init, my pid is: 1
1 :init ini printf('init, my pid is: %d\\n',_pid
Starting Hello with pid 2
  2:    proc  1 (:init::1) creates proc  2 (Hello)
1 :init ini lastpid = run Hello()
last pid was: 2
1 :init ini printf('last pid was: %d\\n',lastpid 2
Hello, my pid is: 2
2 Hello 1)  printf('Hello, my pid is: %d\\n',_pi 2
Hello, my pid is: 0
0 Hello 1)  printf('Hello, my pid is: %d\\n',_pi 2
  5:    proc  2 (Hello:1) terminates
  5:    proc  1 (:init::1) terminates
  5:    proc  0 (Hello:1) terminates
3 processes created
```

This confirms that, indeed, three processes were created. In this particular process, the **init** process has `_pid` 1, the `Hello` instance created using the **run** statement in `Init` has `_pid` 2, and the `Hello` instance that was created due to the **active** marking has `_pid` 0.

Note that this also shows that the **init** process does not necessarily get the lowest `_pid`. In fact, the **init** and **active** processes are instantiated in declaration order, and **run** processes are created when the instruction is processed.

The previous example also illustrates how the C-style format string **printf** can be used to output information from a Promela model. Note that this is just diagnostic information, and does not affect the state space that is generated.

Using the **active** keyword, also multiple processes can be created. For instance, one can write **active** `[N]` **proctype** `P()` to create `N` copies of process `P`.

Example 10.1.2. We illustrate this using an adaptation of our previous example, creating 2 copies of `Hello`.

```
active[2] proctype Hello() {
    printf("Hello,␣my␣pid␣is:␣%d\n", _pid);
}
init {
    int lastpid;
    printf("init,␣my␣pid␣is:␣%d\n", _pid);
    lastpid = run Hello();
    printf("last␣pid␣was:␣%d\n", lastpid);
}
```

During execution, we now get four processes: two copies of `Hello` due to the **active** keyword, the **init** process, and the copy of `Hello` created by the init process. The output of `spin -p hello-N.pml` is as follows:

```
  0:    proc  - (:root:) creates proc  0 (Hello)
  0:    proc  - (:root:) creates proc  1 (Hello)
  0:    proc  - (:root:) creates proc  2 (:init:)
      Hello, my pid is: 0
  1:    proc  0 (Hello:1) hello-N.pml:2 (state 1)       [printf('Hello, my pid is: %d\\n
     ',_pid)]
              init, my pid is: 2
  2:    proc  2 (:init::1) hello-N.pml:6 (state 1)      [printf('init, my pid is: %d\\n
     ',_pid)]
          Hello, my pid is: 1
  3:    proc  1 (Hello:1) hello-N.pml:2 (state 1)       [printf('Hello, my pid is: %d\\n
     ',_pid)]
Starting Hello with pid 3
  4:    proc  2 (:init::1) creates proc  3 (Hello)
  4:    proc  2 (:init::1) hello-N.pml:7 (state 2)      [lastpid = run Hello()]
                  Hello, my pid is: 3
  5:    proc  3 (Hello:1) hello-N.pml:2 (state 1)       [printf('Hello, my pid is: %d\\n
     ',_pid)]
              last pid was: 3
  6:    proc  2 (:init::1) hello-N.pml:8 (state 3)      [printf('last pid was: %d\\n',
     lastpid)]
  6:    proc  3 (Hello:1)                   terminates
  6:    proc  2 (:init::1)               terminates
  6:    proc  1 (Hello:1)           terminates
  6:    proc  0 (Hello:1)       terminates
4 processes created
```

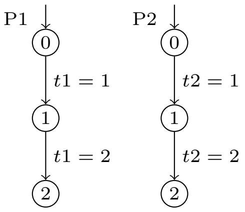
10.1.3 Interleaving semantics

Processes in Promela execute concurrently, i.e., all active processes are allowed to take a step from their current state. Individual statements are atomic, so each statement is executed without any interleaving with other processes. Processes are scheduled non-deterministically, so we cannot predict which process takes the next step. Processes are interleaved, so statements of different processes do not occur at the same time. This means that at any time only one process takes a step. The only exception to processes being interleaved is when synchronous communication is used. We defer its discussion to Section 10.4, and for the time being we assume there is no synchronous communication. When multiple possible actions are enabled at any point during execution, then a choice is made non-deterministically.

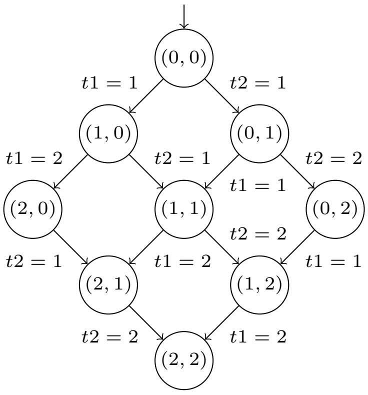
Example 10.1.3. Consider the following PROMELA model.

```
active proctype P1() { byte t1; t1=1; t1=2; }
active proctype P2() { byte t2; t2=1; t2=2; }
```

These processes individually have the following semantics, described as a Kripke structure where the transitions are labelled with the statements that are executed.



Since these processes do not synchronize, the execution is fully interleaved. The resulting semantics is the following.



10.1.4 Variables and types

In the example in Section 10.1.1 we already saw that PROMELA models can contain integer variables. The basic types supported are **bit**, **bool**, which represents bits and Boolean values; **byte**, which represents bytes, i.e. 8-bit values; **short**, which is a 16 bit signed integer; **int**, which represents a 32-bit signed integer. They can be used, e.g., as in the following.

```
bit      turn=1;  /* 0..1 */
bool     flag;    /* false, true (or 0..1) */
byte     counter; /* 8-bits per byte, so 0..2^8 (=255) */
short    s;       /* -(2^16)-1 .. (2^16)-1 */
int      msg;     /* -(2^32)-1 .. (2^32)-1 */
```

The type `pid` is used to keep track of the identifiers of processes; this is, effectively, an alias for the **byte** type.

More complicated types that can be used are arrays and records (that are similar to structs in C). Examples of those are the following:

```
byte     a[27];       /* Byte-array of length 27 */
bit      b[4];        /* Bit-array of length 4 */
typedef Record {      /* Record is a type with two fields, f1 and f2 */
    short f1;
    byte f2;
}
Record  rr;           /* rr is a variable of type Record */
rr.f1 = ..            /* We can assign to the individual fields */
```

Message types, declared using **mtype**, are used to model enumerations. For example, `mtype = {left, right}` is an enumeration with two values, `left` and `right`. Its use is similar to an **enum** in C. Internally, each value will also be associated with an integer in the range 1..255; value `0` is reserved for uninitialized **mtype** variables. Multiple **mtype**s can be used in a single model, by adding the name of the type as follows.

```
mtype:fruit = {apple, pear};
mtype:size = {small, medium, large};
```

To use variables, they should be declared. Variables can get a value in different ways. They can either be assigned a value, they can be given a value through argument passing, or they can get a value through message passing, see Section 10.4.

Consider the following fragment. Here we declare two variables, `ii` and `bb`, and subsequently assign a value.

```
/* We declare two variables, and subsequently assign values */
int ii;
bit bb;
bb = 1;
ii = 2;
```

What is the value of these variables after declaration? The default initial value of basic variables is `0`. So, after both declarations, but before the assignments, variables `ii` and `bb` both have value `0`. In terms of verification times, later, this is not efficient, as it unnecessarily introduces additional states. It is therefore more efficient to assign an initial value at the time of declaration if possible. An example of this is the following.

```
/* Declaration and assignment, avoiding introduction of intermediate state */
short s=-1;
```

Variables can be used in expressions. Most arithmetic, relational and logical operators supported by C are also supported in Promela. Consider the following example.

```
Record rr;
rr.f1 = 0;
rr.f2 = 2;
rr.f2*s+27 == 23; /* Test with arithmetic expression as left-hand side */
printf("value:␣%d\n", rr.f2*rr.f2); /* Print value of numeric expression */
```

Using **`#define`**, macros can be defined (that can be used similar to macros in, e.g., C). This is useful, for instance, if you want to define a constant on which your model depends, or a predicate.

Example 10.1.4. Suppose we want to parameterize our model with a value N that determines, e.g., the size of an array. We can then write the following.

```
#define N 3
```

Now, we can use the generic value N instead of 3 in our model, and we only need to change the value in one place if we want tot try the model for larger instances.

If we want to model a predicate, for instance, we want to model a predicate that is true if its argument is equal to 42, than we can also use macros.

```
#define ISGOOD(x) (x == 42)
```

We can similarly write macros with multiple parameters.

```
#define ISBETWEEN(x, low, high) \
  ( (x > low) && \
    (x < high) )
```

This last example defines a predicate `ISBETWEEN` that determines whether `x` is in the open interval `(low, high)`. Note that we here use `\` at the end of the line to denote the continuation of the line.

Macros are simply expanded using the preprocessor at the places in the model where they are used.

Exercise 10.1 In Exercise 9.2 we modelled the game of tic-tac-toe as a Kripke structure. Now, suppose that we, instead, want to model it using Promela. In this exercise we define some of the *data types* needed to model this game.

1. Define a data type that is able to capture whether a position on the grid is covered by ×, ○, or no player.
2. For the grid, define a global array `grid`, which initially represents the empty grid. You may assume that the layout of the grid is as follows:

6	7	8
3	4	5
0	1	2

3. Define a macro `WINS(grid, p)` that is true if and only if `grid` is in a configuration that is won by player `p`.
4. The game ends in a draw if the entire grid is covered, but neither of the players wins. Define a macro `DRAW(grid)` that encodes that the game ends in a draw.

Note that PROMELA does not allow the straightforward definition of multi-dimensional arrays. Where, e.g., in C you could define an integer matrix with N rows and M columns using a two dimensional array as `int matrix[N][M];` this notation is not supported by PROMELA. However, we can work around this using a trick. We first define a type for the rows.

```
typedef rows {
    int row[M];
};
```

So, basically rows now is a record with as only field an array `row` of length `M`, indicating each of the values on this row. We can now use this to define the matrix itself.

```
rows matrix[N];
```

Hence, `matrix` is now an array containing `N` rows.

We can now access an element at row i column j using the following notation:

```
matrix[i].row[j];
```

Exercise 10.2 Reconsider the grid of the tic-tac-toe game. Suppose we have positions (i, j) with i the row, and j the column as follows.

(2,0)	(2,1)	(2,2)
(1,0)	(1,1)	(1,2)
(0,0)	(0,1)	(0,2)

Define the grid using a two-dimensional array of which the indices correspond to the positions shown above. Initialize the grid to be empty.

10.1.5 Statements

The body of a process consists of a sequence of statements. Each statement is either *executable*, which means the statement can be executed immediately, or it is *blocked*, which means the statement cannot be executed. The executability of a statement depends on the global state of the system.

Some statements are always executable. Examples of this are print statements, assignments, **`skip`**, **`assert`**, and **`break`**. Others are not always executable. In particular, the **`run`** statement is executable if a new process can be created; the number of processes is bounded (by 255), so if the bound is reached, no new process can be started. A **`timeout`** can only be executed if there are no other executable processes. An expression (which is used as a test) is executable if and only if it evaluates to true (i.e. a non-zero value). So, for example, `2 < 3` is executable, `x < 27` is executable if the value of `x` is smaller than `27`, otherwise it blocks, and `3 + x` is executable if `x` is not equal to `-3`, if it is equal to `-3` it blocks because the expression evaluates to `0`.

Example 10.1.5. Using blocking expressions we can very compactly model things such as busy-waiting. Consider the following loop in C-code: `while (a != b)skip;` This can be modelled in PROMELA using just the expression `a == b;`.

Conditionals

To model more complicated processes, we can use conditionals and loops in PROMELA. Conditionals are modelled using an if-statement of the following structure:

```
if
:: c_0 -> s_0_1; s_0_2; ...
:: c_1 -> s_1_1; s_1_2; ...
:: ...
:: c_n -> s_n_1; s_n_2; ...
:: else -> s_e_1; s_e_2; ...
fi
```

Here the `c_i` are conditions. The **`else`** becomes executable if none of the other guards are executable. The operator `->` is equivalent to `;`, but is used to clearly separated guards from subsequent statements. The sequence of statements `s_i_1; s_i_2; ...` is only executed if `c_i` is executable. If there is at least one `c_i` that is executable, the if-statement is executable, and SPIN non-deterministically chooses one of the executable `c_i` and executes the corresponding sequence of statements.

Example 10.1.6. We can use this pattern to give variable `n` a random value in the range $[0, 3]$ as follows.

```
if
:: n=0;
:: n=1;
:: n=2;
:: n=3;
fi
```

Loops

In a similar way, repetition can be modelled using the **`do`**-statement.

```
do
:: c_0 -> s_0_1; s_0_2; ...
:: c_1 -> s_1_1; s_1_2; ...
:: ...
:: c_n -> s_n_1; s_n_2; ...
do
```

With respect to the conditions, a **`do`**-statement behaves like an **`if`**-statement. However, instead of ending the statement at the end of the chosen option, the **`do`**-statement repeats the choice selection. To exit from the loop, the **`break`** statement can be used. This transfers control to the end of the **`do`**-statement.

Example 10.1.7. We can now model a simple process of a traffic light.

```
mtype = {RED, YELLOW, GREEN};

active proctype TrafficLight() {
  mtype state = GREEN;
  do
  :: (state == GREEN)    -> state = YELLOW;
  :: (state == YELLOW)   -> state = RED;
  :: (state == RED)      -> state = GREEN;
  od;
}
```

The color of the light is modelled using the **`mtype`**. The light starts by showing green, and it then cycles through the colors, without ever terminating. Note that the **`do`**-statement in this example does not introduce non-determinism.

Exercise 10.3 In Exercise 10.1 we defined the grid for the game of tic-tac-toe using a global variable. The core of the behaviour of a player in the game is that it repeatedly puts a token on an empty position in the grid.

Model a process `PlayerX` in PROMELA that repeatedly puts the token for player × on an empty position in the grid.

Atomic execution

As we observed before, statements in different processes are executed in an interleaved way. Sometimes, however, we want to execute multiple statements as an indivisible block of actions. In

PROMELA this is facilitated by the **atomic** keyword. All statements inside an **atomic** block are executed without being interrupted by statements from other processes.[5]

Example 10.1.8. Consider the following excerpt, which assumes a previous declaration of integer variables x and y.

```
atomic {
    int tmp = x;
    x = y;
    y = tmp;
}
```

This fragment swaps the values of variables x and y in way that cannot be interrupted by other statements.

Note that the **atomic** construct is often used to initialize groups of processes, ensuring that the execution of all process begins simultaneously.

Example 10.1.9. Consider the following PROMELA fragement.

```
proctype P(byte id) {
    ...
}

init {
    byte proc;
    atomic {
        proc = 0;
        do
        :: proc < N ->
            run P(proc);
            proc++;
        :: proc == N -> break;
        od;
    }
}
```

This fragment declares a process P with parameter id. In the initialization, N copies of this process are created, and each process is passed a unique identifier. The initialization is performed atomically.

10.2 Example: Peterson's mutual exclusion algorithm

Now that we have seen the basic syntax of PROMELA we show how it can be used to model a realistic example.x Mutual exclusion is the problem where multiple processes require access to shared resource in such a way that only one of the processes has access to the shared resource at any time. The access to the shared resource happens in a so-called *cricital section.* To this end, mutual exclusion algorithms execute in cycles, where, at the beginning of the cycle, a process can indicate it wants to access its critical section. The task of the mutual exclusion algorithm is to guarantee that at most one process is in its critical section at any time.

Peterson's algorithm Peterson is a classic algorithm for a shared memory systems that implements mutual exclusion between two processes. As global (shared) variables it uses a Boolean array flag[2], where, whenever flag[i] is **true**, process i wants to enter its critical section. It also uses an integer variable turn, which is used for tie-breaking to ensure that one of the processes is allowed to enter its critical section, when both processes have indicated they want to access the critical section. Initially flag[0] = flag[1] = **false** and turn = 0.

The two processes, with id 0 and 1, are symmetric. We here give the code for process i.

[5] There is one exception to this: if a statement in an **atomic** block is blocked, atomicity is lost and other processes can execute statements.

```
flag[i] := true;
turn := 1-i;
while (flag[1-1] && turn == 1-i) do
{
    /* busy waiting */
}
/* Critical section */
flag[i] := false;
```

Note that this code is repeated in a (non-terminating) loop, so that each process can repeatedly request access to its critical section. Using `1-i` in the algorithm, we indicate the id of the other process.

We can translate this process directly to a small PROMELA model, using the `pid` type to keep track of the turn.

```
/* Peterson's mutual exclusion algorithm for 2 processes. */

/* shared variables */
bool flag[2];
pid turn;

active [2] proctype P() /* two processes; */
{
        /* Since we only have two processes, and we do not have the separate
           init process the _pid values of these processes are 0 and 1. */
        pid i = _pid;
        do :: true ->
                flag[i] = true;
                turn = 1-i
                do
                        :: (flag[1-i] && (turn == 1-i)) -> skip; /* continue looping */
                        :: else -> break;                        /* stop the loop */
                od;
                skip; /* Here the critical section is executed */
                flag[i] = false;
        od;
}
```

Note that if you run a random simulation of this model, the simulation does not terminate. This is because each of the processes contains an infinite loop. You can still run a random simulation, and manually abort it after a while. In the next section we will see how, alternatively, SPIN can be used to generate the entire state space of the process, and check properties exhaustively.

10.3 Verifying system properties using assertions

SPIN is a model checking tool. Model checking tools automatically verify whether the Kripke structure underlying a model satisfies a property φ. SPIN supports checking the following types of properties: *deadlocks* (or bad endstates), this is the default when verifying models; *assertions* which can be used to check whether properties hold at a particular place in the model; *unreachable code*; and *linear temporal logic* (LTL) properties. In this chapter, we focus on the first three types of properties. Linear temporal logic, and verification of such properties using SPIN, will be discussed in Chapter 11.

In general, we distinguish two kinds of properties. Safety properties and liveness properties. Intuitively, a safety property expresses that *"nothing bad ever happens"*. Examples of safety properties are:

- The value of x is always less than 42 (this is an invariant).
- The car never crashes.
- The system never deadlocks.

When a model satisfies a safety property, the tool will report this. Otherwise, Spin will find a trace to the "bad thing", i.e. a state violating the property. Safety properties can often be expressed in Spinby adding assertions.

Liveness properties express that *"something good will eventually happen"*. Examples of such properties are:

- The system eventually terminates.
- If X occurs then eventually Y occurs.

Liveness properties can typically not be expressed using assertions, and require us to use LTL. In this chapter we therefore focus on safety properties.

10.3.1 Checking for deadlocks

A deadlock is a state in the system in which no further steps are possible. Recall the `Alpha` process from Section 10.1.1. We observed during simulation that a timeout appears, but, in fact, the state that we identified based on the simulation is a deadlock.

In Spin deadlock states are referred to as *invalid end-states*. The only valid end-states are those states at the closing curly brace ({) of a **`proctype`** declaration. Addionally, if it is by design that a process can terminate earlier in the process, we can explicitly mark this by adding an `end:` label at the place where we want to allow termination. This will mark the given state as a valid end state.

Now, let us see how we can use the exhausive verification performed by Spin to identify the deadlock state in the `Alpha` process. If we want to perform exhaustive verification using Spin, we need to perform two steps. First, we need to generate a protocol specific analyzer, and subsequently we need to run the verification. We next describe both steps.

Generating protocol specific analyzer

A protocol specific analyzer is a set of C files that must be compiled in order to generate an executable for the analyzer. An analyzer can be generated for a particular Promela model by executing `spin -a` on the model.

Example 10.3.1. Recall the model in `alpha.pml` described in Section 10.1.1. We can generate the protocol specific analyzer as follows.

```
spin -a alpha.pml
```

Executing this generated the following files:

```
pan.b
pan.c
pan.h
pan.m
pan.t
```

The key files to note here are `pan.c` and `pan.h`, that can be compiled using the C compiler.

We next compile a *verifier* by building these C files using the compiler. Here we assume that the GCC compiler is installed. Compilation can be done using the following command:

```
gcc -DNOREDUCE -o pan pan.c
```

Here, `gcc` is the name of the compiler that is used, `-DNOREDUCE` indicates that the verifier that is generated should not use partial order reduction. Finally, `-o pan pan.c` indicates that the executable that is generated is calles `pan`, and that the file that we should compile is `pan.c`.

If we are sure we only want to perform verification of safety properties, i.e., no cycle detection is needed during the verification process, we can optimize for this by compiling `pan` using the `-DSAFETY` flag. Whenever we are only checking for deadlocks or verifying assertions, this flag is safe to use.

Running the verification

Once the protocol specific analyzer has been generated and compiled, we can perform the corresponding verification by executing `pan` as follows.

```
./pan
```

If we assume the analyzer was generated following the example in the previous section, this command outputs the following.

```
pan:1: invalid end state (at depth 1)
pan: wrote alpha.pml.trail

(Spin Version 6.5.2 -- 6 December 2019)
Warning: Search not completed

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 32 byte, depth reached 2, errors: 1
        3 states, stored
        0 states, matched
        3 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.000       equivalent memory usage for states (stored*(State-vector + overhead))
    0.292       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.458       memory used for DFS stack (-m10000)
  128.653       total actual memory usage



pan: elapsed time 0 seconds
```

In particular, on the first two lines, there is an indication that an invalid end-state was reached, and that a trail file was written. Besides that, the output warns that the search was not completed: because an error was detected, the tool reported the error, and stopped the verification. Furthermore, the options that were used and the time and memory that were required are reported. From the output we see that we checked for assertion violations and invalid end states, and other checks were not performed.

Trail hunting

In the previous section we observed that, when verification fails, a trail file is written. Such a trail file is very useful to get additional information from the model, in order to understand the cause of the failure. If a trail file has been written for `spec.pml`, we can print information from it using

```
spin -t spec.pml
```

Furthermore, the same options `-p`, `-g`, and `-l` that were discussed for random simulation can be used to change the information that is printed.

Example 10.3.2. Reconsider the deadlock detected for the `Alpha` process, corresponding to `alpha .pml`. We can print the counterexample as follows.

```
spin -t -p -l alpha.pml
```

It produces the following output:

```
Starting Alpha with pid 1
  1:    proc  0 (:init::1) alpha.pml:10 (state 1)       [(run Alpha(5,3))]
  2:    proc  1 (Alpha:1) alpha.pml:3 (state 1) [(1)]
  2:    proc  1 (Alpha:1) alpha.pml:4 (state 2) [x = 2]
                Alpha(1):x = 2
spin: trail ends after 2 steps
#processes: 2
  2:    proc  1 (Alpha:1) alpha.pml:5 (state 3)
                Alpha(1):z = 1
                Alpha(1):y = 3
                Alpha(1):x = 2
  2:    proc  0 (:init::1) alpha.pml:11 (state 2) <valid end state>
2 processes created
```

The output indicates the trail ends after 2 steps. The last step on this trail (labelled `2:`) is where `Alpha` has performed `x = 2`, and at the end of the trail, the values of the local variables are `z = 1, y = 3, x = 2`. From this we can immediately see that the next statement in `Alpha`, which is a boolean expression, evaluates to false.

In the case of this example, the trail actually contains the shortest path to a counterexample. However, in general this is not guaranteed by the verifier. If during debugging your model using a counterexample, you find that the counterexample is long, in some cases, it helps to search for a shorter counterexample. This can be achieved by executing `pan` with the `-i` option.

Exercise 10.4 Use the verifier in SPIN to show that the model of Peterson's mutual exclusion algorithm from Section 10.2 is deadlock free.

Exercise 10.5 Instead of explicitly modelling Peterson's using a busy-waiting loop, we can also rely on the fact that processes block on a boolean expression as long as the expression evaluates to false. The resulting model is the following:

```
/* Peterson's mutual exclusion algorithm for 2 processes. */

/* shared variables */
bool flag[2];
pid turn;

active [2] proctype P() /* two processes; */
{
        /* Since we only have two processes, and we do not have the separate
           init process the _pid values of these processes are 0 and 1. */
        pid i = _pid;
        do :: true ->
                flag[i] = true;
                turn = 1-i
                !(flag[1-i] && (turn == 1-i)); /* Block until guard of busy-wait loop is
                     false */
                flag[i] = false;
        od;
}
```

Use the verifier in SPIN to show that this alternative model is also deadlock free.

Exercise 10.6 Consider the following attempt at giving a mutual exclusion algorithm, still for two processes, that does not use the `turn` variable, other than that the variables and initialization are the same as in Peterson's algorithm.

```
    flag[i] := true;
    while (flag[1-1]) do
    {
        /* busy waiting */
    }
    /* Critical section */
    flag[i] := false;
```

1. Create a Promela model for this mutual exclusion algorithm for two processes, using the trick from Exercise 10.5 to model the busy-waiting loop.
2. Use Spin to show that the resulting model has a deadlock.
3. Using the counterexample, explain the deadlock.

10.3.2 Checking safety properties

Safety properties can be checked for a Promela model by adding assertions to the model. These assertions, like in programming, are boolean expressions about local and global variables that should hold at the place where they are evaluated.

Example 10.3.3. Recall the model of Peterson's mutual exclusion algorithm from Section 10.2. To check mutual exclusion, we add an additional global variable that is incremented when a process enters its critical section, and decremented when a process leaves its critical section. In the critical section, we then assert that the number of process in the critical section is exactly 1. If mutual exclusion does not hold, the assertion will be violated, as there is some state in the model where both processes have incremented this global variable, neither of the process have decremented it. We have also added an explicit check on our assumption of the value of `_pid`.

The process that results is the following:

```
/* Peterson's mutual exclusion algorithm for 2 processes. */

/* shared variables */
bool flag[2];
pid turn;
byte ncrit;

active [2] proctype P() /* two processes; */
{
        /* Since we only have two processes, and we do not have the separate
           init process the _pid values of these processes are 0 and 1. */
        assert((_pid == 0) || (_pid == 1))
        pid i = _pid;
        do :: true ->
                flag[i] = true;
                turn = 1-i
                do
                        :: (flag[1-i] && (turn == 1-i)) -> skip; /* continue looping */
                        :: else -> break;                        /* stop the loop */
                od;
                ncrit++;
                assert(ncrit == 1) /* Here the critical section is executed */
                ncrit--;
                flag[i] = false;
        od;
}
```

Verification can be done as before using:

```
spin -a peterson-assert.pml
gcc -DSAFETY -DNOREDUCE -o pan pan.c
./pan
```

This reports no errors, hence mutual exclusion is satisfied.

Exercise 10.7 Consider the following attempts at a mutual exclusion algorithm, that is similar to the version in Exercise 10.6, but in which the flag is set only after the process has determined that the other process has not set its flag.

```
    while (flag[1-1]) do
    {
        /* busy waiting */
    }
    flag[i] := true;
    /* Critical section */
    flag[i] := false;
```

1. Create a Promela model for this mutual exclusion algorithm for two processes. You can choose whether to model busy-waiting using a loop, or using the trick from Exercise 10.5. Include a counter and an assertion to determine whether mutual exclusion is violated.
2. Use Spin to show that the resulting model violates mutual exclusion.
3. Using the counterexample, explain the deadlock.

10.4 Communication

So far we have focused on shared memory processes. If we want to model distributed systems that, e.g., communicate via the network, we need no explicitly model communication channels, and the communication that happens along these channels.

Communication via *channels* is supported in Promela. Communication can be either synchronous (also referred to as rendezvous communication) or asynchronous (message passing). Both types of communication are modelled in a similar way.

```
    chan <name> = [<dim>] of {<type1>, <type2>, .., <typen>};
```

where

- `<name>` is the name of the channel.
- `<dim>` is the capacity of the channel, i.e., the number of messages that can be contained in the channel. If `<dim>` is `0`, the channel is synchronous.
- `{<type1>, <type2>, .., <typen>}` is the type of the messages contained in the channel.

Channels essentially operate in a FIFO (first-in-first-out) manner, meaning that the message that was first inserted into the channel by the sender also needs to be consumed from the channel first by the receiver. In case the channel dimension is `0`, no message can be stored in the channel, and sending and receiving need to happen simultaneously.

Example 10.4.1. Consider the following declarations.

```
mtype { MSG, ACK };
chan c = [0] of {bit};
chan toR = [2] of {mtype, bit};
chan line[2] = [1] of {mtype, Record};
```

We have a message type, denoted using **`mtype`** consisting of elements `MSG` and `ACK`, for message and acknowledgement. Channel `c` is a synchronous channel to exchange bits; `toR` is an asynchronous channel of dimension `2` along which messages of the form **`mtype, bit`** can be exchanged. Finally, `line` is an array of channels of capacity `1`.

Note that in the example we have shown that, if we want to use multiple channels of the same type, we can use an array of channels. This can, e.g., be used to compactly model the situation where every process has its own incoming channel, indexed with the identifier of the process.

Promela's syntax distinguishes between sending and receiving messages. Sending a message really means putting a message in a channel buffer (for *dim* > 0), and is written as `ch!<expr1>,<expr2>,...,<exprn>;`, where `ch` is the channel along which the message is sent. The `<expri>` are the values that are sent. Their type must correspond to the types of the channel declaration. A send statement is executable if the channel buffer is not full. If the channel buffer is full, the statement blocks.

Example 10.4.2. In the following fragment, we use the declarations from the previous example. We first send message `1` on channel `c`, then `0` along `toR`. Finally, we send a record `m`, with values `5` and `10` for the fields along the first `line` channel. (Recall that we have an array of channels `line`.)

```
c!1;
toR!ACK, 0;
Record m;
m.f1=5;
m.f2=10;
line[1]!MSG, m;
```

Receiving messages from a channel buffer is done in a similar way. The basic way of receiving messages using *message passing* is `ch?<var1>,<var2>,...,<varn>;`, where, if the channel buffer is not empty, the first message is read from the channel, the values are stored in the `<vari>` variables and the message is removed from the channel. Instead of accepting messages containing any value, can use the pattern `ch?<const1>,<const2>,...,<constn>;` to only accept the message if the first message in the channel matches als the `<consti>` conditions, i.e., the values of the first message in the channel are equal to the constants. We call this *message testing*. Now, to be of use, `<var>` and `<const>` entries can be mixed.

Example 10.4.3. In the following fragment, a bit is received from channel `c` and stored to `b`, and along `line[1]` we receive a message provided that the first element is `MSG`; the record in the second element is stored to `m`.

```
bit b;
c?b;
Record m;
line[1]?MSG,m;
```

Note that, when receiving from channel `line[1]`, if the message is of the form `ACK,m`, receiving on the last line will not succeed, as `ACK` does not match `MSG`.

Instead of testing whether an element of a message is equal to a constant, we can also compare with the current value of a particular variable. To this end, `eval(x)` represents the constant with value equal to `x`.

Example 10.4.4. In the following fragment, we only receive a message provided the first element is `MSG`, the current value of variable `g`.

```
mtype g = MSG;
Record m;
line[1]?eval(g),m;
```

When doing rendez-vous (synchronous) communication, i.e. the dimension of a channel is 0, the channel cannot store messages. So, if one process wants to execute a send statement `ch!` and simultaneously another process wants to execute a corresponding receive statement `ch?` with matching constants, then both statements are executable. Both statements will handshake, and *together* result in a transition of the system.

Example 10.4.5. Consider the following model.

```
chan ch = [0] of {bit, byte};

active proctype P {
  ch!1,3+7;
}

active proctype Q {
  byte x;
  ch?1,x;
}
```

In this, `P` wants to do `ch!1,3+7`, and `Q` wants to do `ch?1,x`, so `P` and `Q` can synchronize. After doing so, `x` will have value `10`.

Exercise 10.8 In Exercises 10.1 and 10.3 we developed the basics of a model of a tic-tac-toe game. However, so far, we only modelled an independent player, and did not yet model the turns that players take.

1. Define a message type **mtype**:message that can be used to indicate the beginning and end of a turn.
2. Using this message type, declare two synchronous channels, one for each player, that are used to indicate the beginning and end of the turn of these players.
3. Use these channels to extend the process of player × you defined in Exercise 10.3 such that the player only places a token on the grid after it has got the turn, and after putting a token on the grid, it ends the turn.
4. Extend your model with the process for player ○.
5. Finally, we have to model the game itself. For this, model a process Game that keeps track of the turn and the winner (if one of the players has already won). This process passes the turn to each of the player in an alternating fashion, and each time checks whether a player has won the game already.

Exercise 10.9 Reconsider the tic-tac-toe model that you constructed in previous exercises. The game terminates when the loop of the Game process terminates.

1. First check whether this model contains deadlocks. If you get an error message, can you explain why you get an error message? Resolve the error.
2. In the updated model, verify using an assertion whether, when the game terminates, we have a winner.
3. If you find that the previous property does not hold, think about how the assertion should be changed to obtain a property that does hold when the game terminates.

So far we have only been interested in the values of local and global variables in counterexamples. When dealing with channels, there are some additional options that can be passed to spin during random simulation, and when printing counterexamples:

- -r shows all message receive events,
- -s shows all message send events.

Furthermore, when debugging communicating processes, it can be useful to print the trace in the form of a (textual version of a) sequence diagram. This can be achieved using the option -c.

10.5 Alternating Bit Protocol

We now switch our attention to modelling a real communication protocol in Promela. Consider the situation where a sender process S and a receiver process R want to reliably exchange messages along unreliable channels. That is, there are lossy channels toR and toS. Now the question is, how can we ensure that a message sent by the sender is received by the receiver. A standard solution to this problem is to add a bit to every message sent by the sender. The receiver acknowledges a message by sending back the received bit. If the sender is certain that the receiver correctly received the previous message, it sends a new message, in which it flips the bit, otherwise it resends the previous message. This standard solution is typically referred to as the *alternating bit protocol.*

10.5.1 Reliable channels

Let us first assume that the channels are reliable, so no messages are lost. The sender and receiver processes can be modelled in Promela using the model shown in Listing 1.

In this model, we see the following. First, we have two types of messages, an actual message, denoted MSG, and an acknowledgement, denoted ACK. We have two channels that can each contain N messages. The sender outputs sendbit along its output channel, and subsequently receives an

```
#define N       2 /* dimension of channels */

mtype = { MSG, ACK };
chan toR = [N] of { mtype, bit };   /* Channel from S to R */
chan toS = [N] of { mtype, bit };   /* Channel from R to S */

proctype Sender(chan in,out) {
  bit sendbit;  /* alternation bit transmitted */
  bit recvbit;  /* alternation bit received */

  do
  :: out ! MSG(sendbit) ->     /* Send current message */
     in ? ACK(recvbit);        /* Await response */
     if
     :: recvbit == sendbit -> /* Successful transmission */
        sendbit = 1-sendbit;  /* Toggle bit */
     :: else -> skip;        /* Transmission error, don't get new msg */
     fi;
  od;
}

proctype Receiver(chan in,out) {
  bit recvbit;  /* alternation bit received */
  do
  :: in ? MSG(recvbit) -> /* Message received successfully */
     out ! ACK(recvbit);  /* Send acknowledgement with received bit */
  od;
}

init {
  run Sender(toS, toR);
  run Receiver(toR, toS);
}
```

Listing 1: Alternating Bit Protocol with reliable channels.

acknowledgement on its input channel. It subsequently checks if the acknowledgement contains the bit that was just sent. If so, the message was transferred correctly, and we send a message with a new bit; if not, we resend the message that was sent previously.

The receiver operates in a similar way. It receives a message along its input channel, and sends the same bit as acknowledgement.

Note that this model introduces a bit of new notation as well: we can parameterize processes with channels, so that we can more conveniently refer to them with their local names. For instance, `toS` is an outgoing channel for `R`, and an incoming channel for `S`. Furthermore, we can use **`#define`** to give a name to an expression, this is basically just a macro that is substituted at all places where it appears. Finally, instead of writing `out ! MSG, sendbit`, we write `out ! MSG(sendbit)`. Whenever the first argument of a channel represents the message type, this tends to be more readable.

10.5.2 Message content

In the previous model, the message only contains the bit, but no content. We extend the model such that a message also contains a value. For convenience, we let the sender send numbers modulo 8, so it first sends 0, then 1, etc, until it sends 7. It then wraps back to 0. We need to calculate this next value, and extend the sender and receiver processes to deal with it. The new model is shown in Listing 2.

```
#define N       2   /* dimension of channels */
#define MAX     8 /* MAX value to send */

mtype = { MSG, ACK };
chan toR = [N] of { mtype, byte, bit }; /* Channel from S to R */
chan toS = [N] of { mtype, bit };       /* Channel from R to S */

proctype Sender(chan in,out) {
  bit sendbit;  /* alternation bit transmitted */
  bit recvbit;  /* alternation bit received */
  byte m;             /* message data */

  do
  :: out ! MSG(m,sendbit) ->    /* Send current message */
     in ? ACK(recvbit);             /* Await response */
     if
     :: recvbit == sendbit ->   /* Successful transmission */
        sendbit = 1-sendbit;    /* Toggle bit */
        m = (m+1)%MAX                 /* Get new message */
     :: else -> skip;                 /* Transmission error, don't get new msg */
     fi;
  od;
}

proctype Receiver(chan in,out) {
  byte m;            /* message data received */
  bit recvbit;  /* alternation bit received */
  do
  :: in ? MSG(m, recvbit) ->    /* Message received successfully */
     out ! ACK(recvbit);                   /* Send acknowledgement with received bit */
  od;
}

init {
  run Sender(toS, toR);
  run Receiver(toR, toS);
}
```

Listing 2: Alternating Bit Protocol with reliable channels and data.

10.5.3 Unreliable channels

So far, we still did not deal with unreliability. To model an unreliable channel, we now, essentially, split the channels, and introduce an intermediate process. Such an unreliable channel can receive a message from the sender, and either forward it to the receiver, or send an `error` to the receiver to indicate that the message was corrupted.

The unreliable channels used in the alternating bit protocol are shown in Listing 3.

```
proctype unreliable_channel_bit(chan in, out) {
  bit b;  /* Bit received from input */

  do
  :: in ? ACK(b) ->     /* Receive ack along input channel */
     if
     ::out ! ACK(b);    /* Correct transmission */
     ::out ! ERROR(0);  /* Corrupted */
     fi
  od
}

proctype unreliable_channel_data(chan in, out) {
  byte d;               /* Data received from input */
  bit b;                /* Bit received from input */

  do
  :: in ? MSG(d,b) ->   /* Receive transmission along input channel */
     if
     ::out ! MSG(d,b);  /* Correct transmission */
     ::out ! ERROR(0,0);         /* Corrupted */
     fi
  od
}
```

Listing 3: Unreliable channels for the alternating bit protocol.

These channels can now be used in the alternating bit protocol. Note that we also change the initialization, and extend the sender and receiver processes to deal with errors that occur. In particular, if the sender receives an ERROR, it will resend the previous message. If it receives an acknowledgement, it will use the bit in the acknowledgement to determine if it needs to resend the previous message or not. If not, it will determine the next message. To get a more concise model, we have unrolled the loop in the sender one step, so the first message is sent outside the loop, and inside the loop, a message is sent after receiving an acknowledgement.

Similarly, if the receiver receives an ERROR it sends an acknowledgement with the old bit. If it receives a message with the same bit as the message that was last successfully received, it knows it is dealing with a retransmission, and the message is ignored. If a message with a different bit as the message that was last successfully received, the message is new, and it should be accepted. The receive bit and the last received message are updated.

Note that, to fit the appropriate channel formats, the parameters of the error messages are filled up with zeroes. In a random simulation this could be omitted, but verification will fail because too few parameters are provided with the message.

In the model, we have added some aspects to the receiver that help in the verification and debugging of the protocol. For instance, when a message is accepted, it is printed using **`printf`**. Additionally, we store the last message that was accepted, so that we can test whether a new message that is accepted is the correct one. To test this, we use the **`assert`** statement.

Exercise 10.10 Consider the model of the alternating bit protocol as we have constructed it in this section. Observe that the model has an assertion that checks that we only accept the correct messages.

```
#define N       2 /* dimension of channels */
#define MAX     2 /* Send integers modulo MAX */

mtype = { MSG, ACK, ERROR };

chan fromS = [N] of { mtype, byte, bit };       /* From S to unreliable channel */
chan toR = [N] of { mtype, byte, bit };         /* From unreliable channel to R */
chan fromR = [N] of { mtype, bit };             /* From R to unreliable channel */
chan toS = [N] of { mtype, bit };               /* From unreliable channel to s */

proctype Sender(chan in,out) {
  bit sendbit;  /* alternation bit transmitted */
  bit recvbit;  /* alternation bit received */
  byte m;       /* message data */

  out ! MSG(m,sendbit) ->       /* Send current message */
  do
  :: in ? ACK(recvbit);         /* Await response */
     if
     :: recvbit == sendbit ->   /* Successful transmission */
        sendbit = 1-sendbit;    /* Toggle bit */
        m = (m+1)%MAX           /* Get new message */
     :: else ->
        skip                    /* Transmission error, don't get new msg */
     fi;
     out ! MSG(m,sendbit) ->    /* Send message (either old or new) */
  :: in ? ERROR(recvbit) ->     /* Receive error */
     out ! MSG(m,sendbit)       /* Send again */
  od;
}

proctype Receiver(chan in,out) {
  byte m;                       /* message data received */
  byte last_m = MAX-1;          /* data of last error-free message, for verification */
  bit recvbit;                  /* alternation bit received */
  bit last_recvbit = 0;         /* receive bit of last error-free message */

  do
  :: in ? MSG(m, recvbit) ->            /* Message received successfully */
     out ! ACK(recvbit);                        /* Send acknowledgement with received
         bit */
     if
     :: (recvbit == last_recvbit) ->    /* bit is not alternating, old message */
        skip                            /* don't accept message */
     :: (recvbit != last_recvbit) ->    /* bit is alternating; accept message */
        printf("ACCEPT␣%d\n", m);
        assert(m == (last_m+1)%MAX);    /* check that we accept only correct messages */
        last_recvbit = recvbit;         /* store alternating bit */
        last_m = m                      /* store m */
     fi
  :: in ? ERROR(m, recvbit) ->          /* Receive error */
     out ! ACK(last_recvbit)            /* Send ack with old bit */
     fi
  od
}

init {
  run Sender(toS, fromS);
  run Receiver(toR, fromR);
  run unreliable_channel_bit(fromR, toS);
  run unreliable_channel_data(fromS, toR);
}
```

Listing 4: Alternating Bit Protocol with unreliable channels and data.

1. Use Spin to check that this version of the alternating bit protocol is deadlock free and satisfies all assertions.
2. In the model of the `Receiver`, we initialize `last_recvbit` to `1`. Change this to initialize to `0`, and check safety of this version of the protocol again. What do you observe? Give an explanation of the message you see.

Chapter 11

Linear Temporal Logic (LTL)

In previous courses such as logic and set theory you have learned to formally reason using propositional and predicate logic. Such logics only allow you to reason about static properties. You could, for instance, formally specify a sorting routine by relating its inputs to its post-condition. However, when reasoning about the behavior of system, it is often desirable to reason about (changes in) the system during its execution.

In Chapter 10 we have seen how safety properties in Promela models can be formulated using assertions, and checked using Spin. In this chapter, we instead formulate properties using temporal logics. Temporal logics allow reasoning about propositions in terms of time, for instance, "I am always studying", "I will eventually graduate", or "I will study until I graduate".

There are many different temporal logics that are used to formally describe requirements, and that allow (automatically) verifying whether a system described using LTSs or Kripke structures satisfy the requirements. The most commonly used temporal logics are LTL, CTL, CTL* and the modal mu-calculus. In this chapter, we introduce Linear Temporal Logic (LTL) to reason about state properties of Kripke structures defined in Chapter 9. The other logics will be introduced in other courses in the computer science bachelor or master's programmes.

We base this chapter on standard textbook descriptions such as (Baier and Katoen, 2008, Chapter 5).

At the end of this chapter, you

- can explain the syntax and semantics of LTL;
- can informally explain the meaning of LTL formulas;
- can translate requirements into LTL formulas;
- can determine whether a given Kripke structure satisfies a given LTL formula.
- can express LTL formulas in the explicit syntax used by Spin.
- can use Spin to verify whether a Promela model satisfies an LTL formula.

11.1 Syntax

We first describe the syntax according to which LTL formulas are constructed. LTL formulas are constructed using *atomic propositions*, denoted a, standard Boolean connectives $\neg$, $\wedge$ and $\vee$, and the temporal modalities X (next), U (until), F (eventually), and G (globally/always).

Definition 11.1.1 (LTL syntax)**.** Let AP be a set of atomic propositions. The set of LTL formulas over AP is described using the following syntax:

$$\varphi ::= a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \mathsf{X}\,\varphi \mid \varphi_1 \,\mathsf{U}\, \varphi_2 \mid \mathsf{F}\,\varphi \mid \mathsf{G}\,\varphi$$

One should read this definition as follows. The standard boolean connectives have their traditional meaning.

- a is an atomic proposition,
- φ, φ_1 and φ_2 are LTL formulas,
- X denotes the "next" operator,
- U denotes the "until" operator,
- F denotes the "eventually" operator, and
- G denotes the "globally" operator.

Note that for instance the constants true and false can be obtained by taking an atomic proposition a, and using $a \wedge \neg a$ for false, and $a \vee \neg a$ for true. We also use the standard rules for propositional logic, and write $\varphi \rightarrow \psi$ instead of $\neg\varphi \vee \psi$, and $\varphi \leftrightarrow \psi$ for $(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$. Note that $\rightarrow$ is implication and $\leftrightarrow$ is bi-implication. We here use the single arrow $\rightarrow$ instead of the standard double arrow $\Longrightarrow$ to clearly distinguish between the implication within the syntax of LTL, and the implication when we are reasoning *about* LTL.

Operator precedence The unary operators bind stronger than the binary ones. Negation ($\neg$) and X, F and G bind equally strong. Temporal operator until (U) takes precedence over the boolean connective ($\wedge$). Operator U is right associative, that is, $\varphi_1 \,\mathsf{U}\, \varphi_2 \,\mathsf{U}\, \varphi_3$ stands for $\varphi_1 \,\mathsf{U}\, (\varphi_2 \,\mathsf{U}\, \varphi_3)$.

So, for example formula $\mathsf{F}\, a \,\mathsf{U}\, b$ should be read as $(\mathsf{F}\, a) \,\mathsf{U}\, b$, and $\mathsf{X}\, a \,\mathsf{U}\, \neg\, \mathsf{G}\, b$ is $(\mathsf{X}\, a) \,\mathsf{U}(\neg(\mathsf{G}\, b))$.

11.2 Semantics of LTL

Ultimately, we want to establish what it means for an LTL property to hold for (the initial state of) a Kripke structure. For the sake of simplicity of the definitions, in this chapter we assume that Kripke structures have a total transition relation. We therefore do not have to deal with finite paths in the definition of the semantics. The semantics of LTL follows a two step approach. The property holds for a Kripke structure if it holds for its initial state. An LTL property holds for a state in a Kripke structure if it holds for all *infinite paths* starting in that state. On the lower level, therefore, the semantics of an LTL formula is defined on such paths. In this chapter, we sometimes talk about paths instead of infinite paths, especially when it is clear from the context that we mean infinite paths.

Definition 11.2.1 (Validity of an LTL formula over a path)**.** Let $(S, s_0, \rightarrow, AP, L)$ be a Kripke structure, π an infinite path starting in some state in the Kripke structure. Validity of an LTL formula over an infinite path, denoted $\pi \models \varphi$, is defined inductively as follows.

- $\pi \models a$ if and only if $a \in L(\pi[0])$;
- $\pi \models \neg\varphi$ if and only if $\pi \not\models \varphi$;
- $\pi \models \varphi \wedge \psi$ if and only if $\pi \models \varphi$ and $\pi \models \psi$;
- $\pi \models \varphi \vee \psi$ if and only if $\pi \models \varphi$ or $\pi \models \psi$;
- $\pi \models \mathsf{X}\,\varphi$ if and only if $\pi[1\ldots] \models \varphi$;
- $\pi \models \varphi \,\mathsf{U}\, \psi$ if and only if $\exists j \geq 0. \pi[j\ldots] \models \psi$ and for all $0 \leq i < j$, $\pi[i\ldots] \models \varphi$;
- $\pi \models \mathsf{F}\,\varphi$ if and only if $\exists i \geq 0. \pi[i\ldots] \models \varphi$; and
- $\pi \models \mathsf{G}\,\varphi$ if and only if $\forall i \geq 0. \pi[i\ldots] \models \varphi$.

Figure 11.1 illustrates the intuition behind most of the LTL operators. The informal explanation of the semantics is as follows.

Atomic Proposition An atomic proposition is true on an infinite path, if it holds on the first state of the path. In the top line of Figure 11.1, proposition a holds at the first position, so it holds for that path.

Negation of a formula The negation of an LTL formula holds on an infinite path π if and only if the atomic proposition does not hold on that same path.

Conjunction Given two LTL formulas φ and ψ, the conjunction holds on an infinite path π if and only if both formulas hold on the path.

Disjuction Given two LTL formulas φ and ψ, the disjunction holds on an infinite path π if and only if at least one of the formulas holds on the path.

Next The next operator is used to specify that a formula holds *at the next* position on an infinite path. Given proposition a, formula $\mathsf{X}\, a$ holds on an infinite path if it holds at the next state, that is, state at position 1 on the path. At the top of Figure 11.1, $\mathsf{X}\, \neg a$ holds as $a \notin L(\pi[1 \ldots])$. In the second line in the figure, you can see a path where $\mathsf{X}\, a$ holds.

Until The until operator specifies that a formula is true until another one is true. There are two parts in the definition of $\varphi \,\mathsf{U}\, \psi$: (1) formula ψ must hold at some position on the path; and (2) at all previous positions, formula φ must hold. It is important to notice that $\varphi \,\mathsf{U}\, \psi$ enforces ψ to hold but it does not enforce φ to be true. If ψ holds at all positions, $\varphi \,\mathsf{U}\, \psi$ also holds. In the third row in Figure 11.1, we see an example where the property $a \,\mathsf{U}\, b$ holds; a holds in the first two states on the path, whereas b does not hold at those states, but b dus hold in the third state.

Eventually Intuitively (see the fourth row in Figure 11.1) $\mathsf{F}\, \varphi$ means that φ must hold somewhere in the future. This operator is sometimes denoted $\diamond$.

Globally This is the dual of the eventually operator. Intuitively (see the last line in Figure 11.1), $\mathsf{G}\, \varphi$ means that φ always holds. This operator is sometimes denoted $\square$.

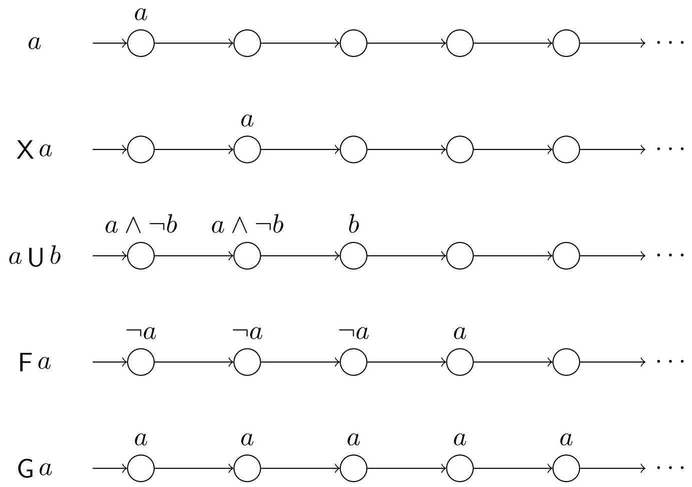


Figure 11.1: Intuition for the main LTL operators, based on (Baier and Katoen, 2008, Figure 5.1).

So far, we have only discussed the semantics of LTL formulas over infinite paths. However, as pointed out at the beginning of this section, we are interested in whether the Kripke structure satisfies an LTL formula. An LTL formula φ is valid in state s, denoted $s \models \varphi$ if and only if φ holds for all infinite paths starting in s. Formally, this is defined as follows.

Definition 11.2.2 (Validity of an LTL formula)**.** Let $K = (S, s_0, \rightarrow, AP, L)$ be a Kripke structure, and $s \in S$.

$$s \models \varphi \text{ if and only if } \forall \pi \in \mathit{paths}^{\omega}(s). \pi \models \varphi$$

An LTL formula φ is valid for Kripke structure K if and only φ holds in the initial state of K. Formally,

$$K \models \varphi \text{ if and only if } s_0 \models \varphi$$

Example 11.2.1. Let's have a look at an example. Consider the model of a simple beverage vending machine in Figure 11.2. On this machine, we can express the following requirement:

> If one coin has been inserted and the start button has been pressed, the system shall eventually serve tea.

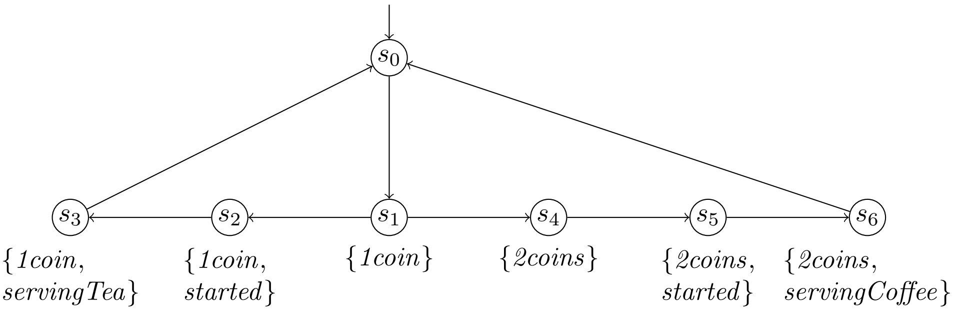


Figure 11.2: A beverage vending machine.

This requirement states that if one coin has been inserted and the start button has been pressed, users should get tea. It also states that users should get tea when one coin has been inserted and the start button has been pressed.

To express this formula, we defined (among others) the following atomic propositions:

- *1coin* is true if and only if one coin has been inserted. Coins are consumed after a beverage has been produced.
- *started* is true if and only if the start button has been pressed. This proposition is reset when the machine starts producing a beverage.
- *servingTea* is true if and only if the system is about to serve tea.

We can then formalize our requirement as follows:

$$\mathsf{G}\,((\mathit{1coin} \wedge \mathit{started}) \rightarrow \mathsf{F}\,\mathit{servingTea})$$

Exercise 11.1 The coffee machine in Figure 11.2 contains the following additional atomic propositions:

- *2coins* is true if and only if two coins have been inserted. Coins are consumed after a beverage has been produced.
- *servingCoffee* is true if and only if the system is about to serve coffee.

Express the following properties as LTL formulas:

1. Eventually, a coin can be inserted into the system.
2. The system infinitely often serves tea.
3. The system infinitely often serves a beverage.
4. Whenever two coins are inserted into the system, the are not consumed before the machine serves coffee.
5. Whenever one coin is inserted into the system, coins are not consumed before the machine serves a beverage.

Exercise 11.2 Assume a system – say a circuit – with one input bit i and one output bit o. We assume atomic propositions $AP = \{o, i\}$. Atomic propositions i and o are true if and only if the corresponding bit has value 1.

Write an LTL formula for each one of the following statements:

1. It is impossible that the system outputs two successive 1's.
2. Whenever the input bit is 1, in at most two steps the output bit will be 1.
3. Whenever the input is 1, the output bit does not change in the next step.
4. The output infinitely often has the value 1.

Exercise 11.3 Consider a traffic light. Write an LTL formula that formalizes the requirement that whenever the light is green, it must remain green until it becomes yellow, and ultimately becomes red.

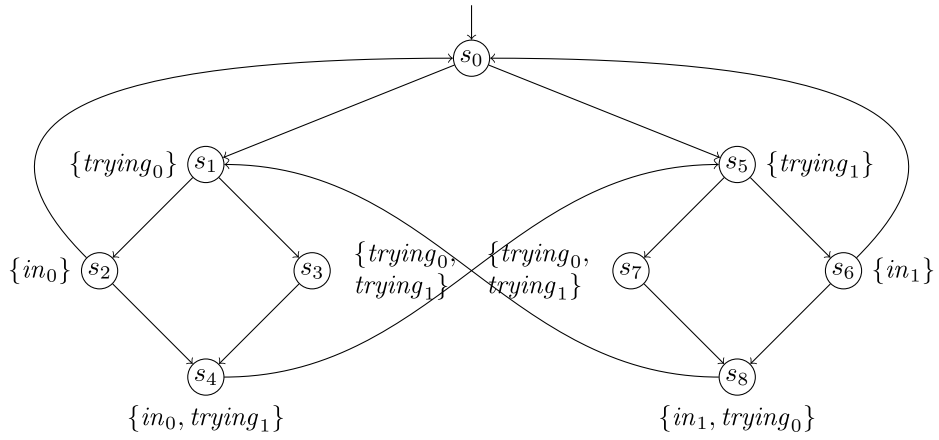


Figure 11.3: Kripke structure modelling mutual exclusion between two processes.

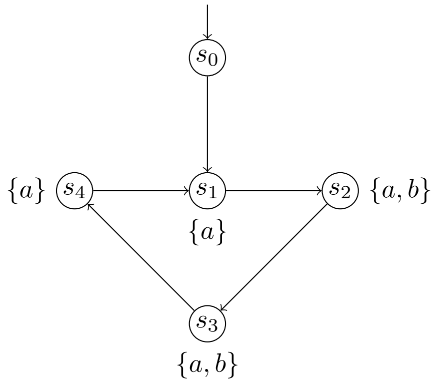
Example 11.2.2. Recall the example of a mutual exclusion protocol from Figure 9.2. We repeat the Kripke structure in Figure 11.3.

We formalize a number of properties using LTL.

- The resource must be accessed under mutual exclusion, meaning that only one process can access it at the same time.
$$\mathsf{G}(\neg(in_0 \wedge in_1))$$
Note that this property holds in the Kripke structure in Figure 11.3.
- Whenever process 0 is trying to enter its critical section, it will eventually be in its critical section. This property is also called starvation freedom.
$$\mathsf{G}(trying_0 \rightarrow F in_0)$$
This property also holds in the Kripke structure in Figure 11.3.
- Whenever process 0 is trying to enter its critical section, process 1 may access its critical section at most once, before process 0 is in its critical section. An attempt to formalize this property is the following.
$$\mathsf{G}(trying_0 \rightarrow (trying_1 \,\mathsf{U}(in_1 \,\mathsf{U}\, in_0))$$
Let us take a closer look at this property. It is an invariant property due to the globally operator. Now, when process 0 is trying to enter its critical section, it can be the case that process 1 is also trying. If that is the case, $trying_1$ holds until the property $in_1 \,\mathsf{U}\, in_0$ holds. We can now have two cases. Either process 0 is the first to enter the critical section, so in_0 holds, the second until is true immediately (and at this point, process 1 is still trying). In the other case, process 1 is the first to enter the critical section, so in_1 becomes true, and to validate the property, in_1 must remain true until process 0 also enters its critical section. If mutual exclusion is satisfied, this hence means that the once process 1 leaves its critical section, process 0 must immediately be granted access to its critical section.
Note that this property does not holds in the Kripke structure in Figure 11.3. A counterexample to this property is the path (prefix) $s_0 s_5 s_7 s_8 s_1 s_2 \ldots$, where in state s_7 $trying_0$ holds, so it must be the case that $s_7 s_8 s_1 s_2 \ldots \models trying_1 \,\mathsf{U}(in_1 \,\mathsf{U}\, in_0)$ then process 1 enters its critical section, so $s_8 s_1 s_2 \ldots \models in_1 \,\mathsf{U}\, in_0$, however, since in s_1 neither in_1 nor in_0 hold, the property is not satisfied.

In the example we have seen that, if a property does not hold, we can give a path that does not satisfy the property. For properties of the form $\mathsf{G}\,\varphi$ it suffices to give a finite prefix of the path to a state in which φ is violated. With a slight abuse of terminology, we also refer to such a path prefix as path in the rest of this chapter.

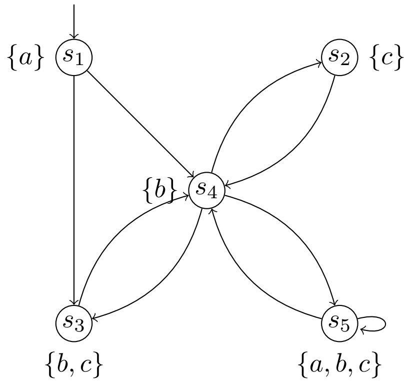
Exercise 11.4 Consider the following Kripke structure K over the set of atomic propositions $\{a, b\}$.



Indicate for each of the following LTL formulas the set of states for which these formulas are satisfied.

1. $\mathsf{X}\,a$
2. $\mathsf{X}(\mathsf{X}\,a)$
3. $\mathsf{G}\,b$
4. $\mathsf{G}\,\mathsf{F}\,b$
5. $\mathsf{F}\,\mathsf{G}\,a$
6. $\mathsf{G}(a\,\mathsf{U}\,b)$
7. $\mathsf{G}(b\,\mathsf{U}\,a)$
8. $\mathsf{F}(b\,\mathsf{U}\,a)$

Exercise 11.5 Consider the following Kripke structure K with initial state s_1 over the set of atomic propositions $AP = \{a, b, c\}$:



Indicate for each of the LTL formulas φ_i below whether $K \models \varphi_i$. Justify your answers! If $K \not\models \varphi_i$ give a path π of K such that $\pi \not\models \varphi_i$.

1. $\varphi_1 = \mathsf{F}\,\mathsf{G}\,c$

2. $\varphi_2 = \mathsf{G}\,\mathsf{F}\,c$
3. $\varphi_3 = \mathsf{X}\,\neg c \rightarrow \mathsf{X}\,\mathsf{X}\,c$
4. $\varphi_4 = \mathsf{G}\,a$
5. $\varphi_5 = a\,\mathsf{U}\,\mathsf{G}(b \vee c)$
6. $\varphi_6 = (\mathsf{X}\,\mathsf{X}\,b)\,\mathsf{U}(b \vee c)$

Exercise 11.6 Suppose we have two users, *Peter* and *Betsy*, and a single printer device *Printer*. Both users perform several tasks, and every now and then they want to print their results on the *Printer*. Since there is only a single printer, only one user can print a job at a time. Suppose we have the following atomic propositions for *Peter* at our disposal:

- *Peter.request*: indicates that *Peter* has requested usage of the printer;
- *Peter.use:* indicates that *Peter* is using the printer;

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

1. Mutual exclusion, that is, only one user at a time can use the printer.
2. Finite time of usage, that is, a user can print only for a finite amount of time.
3. Absence of individual starvation, that is, if a user requests to print something, they are eventually able to do so.
4. Absence of blocking, that is, a user can always request to use the printer.

Exercise 11.7 In this exercise we extend LTL with some new operators. Let φ and ψ be LTL formula. Make the definitions of the following informally explained operators precise by providing LTL formulas that formalize their intuitive meanings.

1. "At next", denoted $\varphi\,\mathsf{N}\,\psi$ which means at the next time where φ holds, ψ also holds.
2. "While", denoted $\varphi\,\mathsf{W}\,\psi$ which means φ holds at least as long as ψ does.
3. "Before", denoted $\varphi\,\mathsf{B}\,\psi$ which means that if ψ holds sometime, φ does so before.

Exercise 11.8 Recall the description of tic-tac-toe for which you created a Kripke structure in Exercise 9.2. Recall the set of atomic propositions

$$AP = \{\mathsf{marked}_y(i,j) \mid y \in \{\times, \bigcirc\} \wedge 1 \leq i,j \leq 3\} \cup \{\mathsf{turn}_\times, \mathsf{turn}_\bigcirc, \mathsf{win}_\times, \mathsf{win}_\bigcirc, \mathsf{draw}\}\,.$$

Give LTL formulas for the following statements:

1. The game always ends
2. There is at most one winner
3. No player can take two turns in a row
4. A cell can contain at most one marking
5. A marked cell can never be unmarked

11.3 Equivalence of LTL formulas

In propositional logic and predicate logic we commonly use equivalences between formulas to reason about logical statements. For instance, $\neg\neg a \equiv a$, and $\neg(a \wedge b) = \neg a \vee \neg b$. In LTL there are similar rules of equivalence. We first define what it means when one LTL formula implies another. Let φ and ψ be LTL formulas, then

$$\varphi \implies \psi \text{ if and only if } \forall \pi\colon \pi \models \varphi \implies \pi \models \psi$$

We extend this to equivalence in the standard way. Let φ and ψ be LTL formulas, then we have that

$$\varphi \equiv \psi \text{ if and only if } \varphi \implies \psi \text{ and } \psi \implies \varphi$$

Note that LTL subsumes propositional logic. Therefore, all equivalences from propositional logic are also valid for LTL. In this section we introduce additional rules that are valid for LTL.

11.3.1 A remark on negation

Before we go on and define some equivalences on LTL formulas, we first remark on negation of LTL formulas in general. For an infinite path, it holds that a property is true if and only if its negation is not true, that is,

$$\pi \models \varphi \text{ if and only if } \pi \not\models \neg\varphi$$

However, **for a Kripke structure, this equivalence does not hold!** A Kripke structure in general may not satisfy a property, while it also does not satisfy the negation of this property. This is due to the fact that both the property and its negation are evaluated over *all infinite paths* starting in the initial state of the Kripke structure.

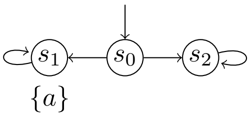


Figure 11.4: Negation example.

Example 11.3.1. Consider the Kripke structure in Figure 11.4. This Kripke structure does not satisfy $\mathsf{F}\, a$ nor does it satisfy $\neg\,\mathsf{F}\, a$. Counter-examples to the first formula are (possibly finite) paths going to state s_2 from the initial state. Counter-examples to the second formula are (possibly finite) paths going to state s_1 from the initial state.

11.3.2 Several laws of equivalence

From the semantics of LTL, it directly follows that F and G can be derived once we have the operator U. In particular, we have the following rules.

$$\begin{array}{ll} \text{(F-U)} & \mathsf{F}\,\varphi \equiv \text{true}\,\mathsf{U}\,\varphi \\ \text{(G-F)} & \mathsf{G}\,\varphi \equiv \neg\,\mathsf{F}\,\neg\varphi \end{array}$$

Because of this, some presentations of LTL will not introduce F and G explicitly as part of the definition, but instead introduce them as derived operators.

Duality From these previous equalities, we immediately get the following dualities between operators in LTL.

$$\begin{array}{ll} \text{(Du-X)} & \neg\,\mathsf{X}\,\varphi \equiv \mathsf{X}\,\neg\varphi \\ \text{(Du-G)} & \neg\,\mathsf{G}\,\varphi \equiv \mathsf{F}\,\neg\varphi \\ \text{(Du-F)} & \neg\,\mathsf{F}\,\varphi \equiv \mathsf{G}\,\neg\varphi \end{array}$$

Note that the last two dualities are similar to the duality between $\forall$ and $\exists$ is predicate logic.

For X, we can for instance see that the duality holds using the following reasoning. Let π be an arbitrary infinite path such that $\pi \models \neg\,\mathsf{X}\,\varphi$. Then $\pi \not\models \mathsf{X}\,\varphi$. From the semantics it follows that $\pi[1\ldots] \not\models \varphi$, hence $\pi[1\ldots] \models \neg\varphi$, so $\pi \models \mathsf{X}\,\neg\varphi$.

Idempotency Operators are idempotent if multiple applications of the same operator do not change the outcome. The temporal operators of LTL satisfy idempotency laws that can be derived from the semantics of LTL. In particular, we have the following laws.

$$\begin{array}{ll}
\text{(Id-G)} & \mathsf{G}\,\mathsf{G}\,\varphi \equiv \mathsf{G}\,\varphi \\
\text{(Id-F)} & \mathsf{F}\,\mathsf{F}\,\varphi \equiv \mathsf{F}\,\varphi \\
\text{(Id-U1)} & \varphi\,\mathsf{U}(\varphi\,\mathsf{U}\,\psi) \equiv \varphi\,\mathsf{U}\,\psi \\
\text{(Id-U2)} & (\varphi\,\mathsf{U}\,\psi)\,\mathsf{U}\,\psi \equiv \varphi\,\mathsf{U}\,\psi
\end{array}$$

As an example, we sketch the proof of the idempotency law for G. Consider an arbitrary infinite path π such that $\pi \models \mathsf{G}\,\mathsf{G}\,\varphi$. According to the semantics, this is equivalent to $\forall i \geq 0.\pi[i\ldots] \models \mathsf{G}\,\varphi$. This is equivalent to (again using the semantics) $\forall i \geq 0.\forall j \geq 0.\pi[i\ldots][j\ldots] \models \varphi$. Note that $\pi[i\ldots][j\ldots] = \pi[(i+j)\ldots]$, so this is equivalent to $\forall i \geq 0.\forall j \geq 0.\pi[(i+j)\ldots] \models \varphi$. Using predicate logic, this is easily shown equivalent to $\forall i \geq 0.\pi[i\ldots] \models \varphi$, which is equivalent to $\mathsf{G}\,\varphi$.

The other laws can be proven using the semantics in a similar way.

Absorption The absorption laws allow to eliminate an occurrence of the F or G operators, depending on the law. The first law observes that from some point on, infinitely often φ, is equivalent to saying that φ holds infinitely often. The second law expresses that when always ultimately φ is the same as saying ultimately φ.

$$\begin{array}{ll}
\text{(Ab-FGF)} & \mathsf{F}\,\mathsf{G}\,\mathsf{F}\,\varphi \equiv \mathsf{G}\,\mathsf{F}\,\varphi \\
\text{(Ab-GFG)} & \mathsf{G}\,\mathsf{F}\,\mathsf{G}\,\varphi \equiv \mathsf{F}\,\mathsf{G}\,\varphi
\end{array}$$

Distributivity We also have a number of distributivity laws. In particular, X distributes over conjunction and disjunction, eventually distributes over disjunction and globally distributes over conjunction.

$$\begin{array}{ll}
\text{(Di-X}\wedge\text{)} & \mathsf{X}(\varphi \wedge \psi) \equiv \mathsf{X}\,\varphi \wedge \mathsf{X}\,\psi \\
\text{(Di-X}\vee\text{)} & \mathsf{X}(\varphi \vee \psi) \equiv \mathsf{X}\,\varphi \vee \mathsf{X}\,\psi \\
\text{(Di-F}\vee\text{)} & \mathsf{F}(\varphi \vee \psi) \equiv (\mathsf{F}\,\varphi) \vee (\mathsf{F}\,\psi) \\
\text{(Di-G}\wedge\text{)} & \mathsf{G}(\varphi \wedge \psi) \equiv (\mathsf{G}\,\varphi) \wedge (\mathsf{G}\,\psi)
\end{array}$$

Be careful however! Even though it might be tempting to distribute F over $\wedge$ and G over $\vee$, these laws do not hold in general.

$$\mathsf{F}(\varphi \wedge \psi) \not\equiv (\mathsf{F}\,\varphi) \wedge (\mathsf{F}\,\psi)$$
$$\mathsf{G}(\varphi \vee \psi) \not\equiv (\mathsf{G}\,\varphi) \vee (\mathsf{G}\,\psi)$$

Next also distributes over all other temporal operators:

$$\begin{array}{ll}
\text{(Di-XF)} & \mathsf{X}\,\mathsf{F}\,\varphi \equiv \mathsf{F}\,\mathsf{X}\,\varphi \\
\text{(Di-XG)} & \mathsf{X}\,\mathsf{G}\,\varphi \equiv \mathsf{G}\,\mathsf{X}\,\varphi \\
\text{(Di-XU)} & \mathsf{X}(\varphi\,\mathsf{U}\,\psi) \equiv (\mathsf{X}\,\varphi)\,\mathsf{U}(\mathsf{X}\,\psi)
\end{array}$$

For some proofs it may be useful to use monotonicity of the temporal operators. This means that if we can prove that $\varphi \implies \psi$ holds, and $\mathsf{X}\,\varphi$ holds, then also $\mathsf{X}\,\psi$ holds. Note that these rules are implications instead of equivalences.

$$\begin{array}{ll}
\text{(M-X)} & (\varphi \implies \psi) \implies \mathsf{X}\,\varphi \implies \mathsf{X}\,\psi \\
\text{(M-F)} & (\varphi \implies \psi) \implies \mathsf{F}\,\varphi \implies \mathsf{F}\,\psi \\
\text{(M-G)} & (\varphi \implies \psi) \implies \mathsf{G}\,\varphi \implies \mathsf{G}\,\psi \\
\text{(M-U1)} & (\varphi \implies \varphi') \implies (\varphi\,\mathsf{U}\,\psi) \implies (\varphi'\,\mathsf{U}\,\psi) \\
\text{(M-U2)} & (\psi \implies \psi') \implies (\varphi\,\mathsf{U}\,\psi) \implies (\varphi\,\mathsf{U}\,\psi')
\end{array}$$

To illustrate why these last results hold, consider the proof of M-X, other results can be proven in a similar way.

Lemma 11.3.1. *Let φ and ψ be arbitrary LTL formulas. Then*

$$(\varphi \implies \psi) \implies \mathsf{X}\,\varphi \implies \mathsf{X}\,\psi$$

Proof. Fix arbitrary LTL formulas φ and ψ, and assume that $\varphi \implies \psi$, i.e., for all infinite paths π, if $\pi \models \varphi$ then $\pi \models \psi$. We have to prove that $\mathsf{X}\,\varphi \implies \mathsf{X}\,\psi$, i.e., for all infinite paths π, $\pi \models \mathsf{X}\,\varphi \implies \mathsf{X}\,\psi$. Fix arbitrary infinite path π, and assume that $\pi \models \mathsf{X}\,\varphi$. By definition of X, then $\pi[1\ldots] \models \varphi$. Then, according to our assumption, $\pi[1\ldots] \models \psi$, hence by definition of X, $\pi \models \mathsf{X}\,\psi$. □

Exercise 11.9 Consider the formulas $\varphi = a\,\mathsf{U}(b\,\mathsf{U}\,c)$ and $\psi = (a\,\mathsf{U}\,b)\,\mathsf{U}\,c$. Determine whether these formulas are equivalent. If they are, give a proof, if not, give a counterexample (that is, a path such that one formula holds on the path and the other does not).

Exercise 11.10 For each of the following equivalences, state if the equivalence holds. If it holds, prove the equivalence. If not, provide a counter-example that illustrates that the formula on the left and the formula on the right are not equivalent.

1. $\mathsf{G}\,\varphi \rightarrow \mathsf{F}\,\psi \equiv \varphi\,\mathsf{U}(\psi \vee \neg\varphi)$
2. $\mathsf{F}\,\mathsf{G}\,\varphi \rightarrow \mathsf{G}\,\mathsf{F}\,\psi \equiv \mathsf{G}(\varphi\,\mathsf{U}(\psi \vee \neg\varphi))$
3. $\mathsf{G}\,\mathsf{G}(\varphi \vee \neg\psi) \equiv \neg F(\neg\varphi \wedge \psi)$ (easy)
4. $\mathsf{F}(\varphi \wedge \psi) \equiv \mathsf{F}\,\varphi \wedge \mathsf{F}\,\psi$
5. $\mathsf{G}\,\varphi \wedge \mathsf{X}\,\mathsf{F}\,\varphi \equiv \mathsf{G}\,\varphi$
6. $\mathsf{F}\,\varphi \wedge \mathsf{X}\,\mathsf{G}\,\varphi \equiv \mathsf{F}\,\varphi$ (easy)
7. $\mathsf{G}\,\mathsf{F}\,\varphi \rightarrow \mathsf{G}\,\mathsf{F}\,\psi \equiv \mathsf{G}(\varphi \rightarrow \mathsf{F}\,\psi)$
8. $\mathsf{X}\,\mathsf{F}\,\varphi \equiv \mathsf{F}\,\mathsf{X}\,\varphi$

11.4 Verifying LTL properties using Spin

In the previous chapter, we have verifies safety properties using assertions. Recall that safety properties express that *"nothing bad ever happens"*. For instance, there is no violation of mutual exclusion. However, if we want to express liveness properties, that is, properties expressing that *"something good will eventually happen"*, using assertions does not suffice. In such cases, LTL is a useful tool. In the remainder of this chapter we illustrate how LTL properties can be verified using Spin. One key difference between safety and liveness properties is that, when a model violates a liveness property, Spin will find a lasso, i.e., a loop on which "something good" never happens, and a prefix that leads to a state on this loop. This in contrast to the traces that show assertion violations.

11.4.1 Spin LTL syntax

When using LTL in Spin we need to be careful about the syntax used. Spinsupports the following operators (where we show both the temporal operators and the standard boolean operators).

- `[]` the temporal operator always, i.e. G
- `<>` the temporal operator eventually, i.e., F
- `!` the boolean operator for negation, i.e., $\neg$
- `U` the temporal operator strong until, i.e., U
- `W` the temporal operator weak until; `p W q` is equivalent to `[]p || (p U q)`, that is, it is like until, but is is also satisfied if `p` always remains true,and `q` never becomes true.
- `V` the dual of `U`: `(p V q)` means `!(!p U !q))`
- `&&` the boolean operator for logical and, i.e. $\wedge$
- `||` the boolean operator for logical or, i.e, $\vee$,
- `/\` alternative form of `&&`
- `\/` alternative form of `||`
- `->` the boolean operator for logical implication, i.e., $\rightarrow$

- `<->` the boolean operator for logical equivalence, i.e., ↔.

This overview was taken from the Spin manual.[1]

Example 11.4.1. Consider the model of the alternating bit protocol with unreliable channels introduced in Chapter 10. Since the messages are accepted in a cyclic way, each of the messages should be accepted eventually.

We can check this by determining that local variable `last_m` in the `Receiver` process can take both values eventually.

This can be expressed in Spin's LTL version as follows.

```
  ltl msg_zero { <>(Receiver:last_m == 0)};
  ltl msg_one { <>(Receiver:last_m == 1)};
```

Remark 11.4.1. Note that we use a pretty ugly trick to express the property in this example. We use remote references, that allow us to peek into the scope of the receiver process.[2]

In particular, the use of remote references is not compatible with some of the techniques that Spin uses internally. If you want to do verification that refers to local variables, it is therefore better to make those variable global.

In what follows, we always ensure that variables used in LTL formulas are global variables.

11.4.2 Checking liveness properties

Liveness properties, typically in the form of LTL formulas, can be checked using Spin as follows. First, LTL formulas are added to a model. When compiling the `pan` executable, the option `-DSAFETY` *should not be passed to the compiler*. Additionally, we need to pass the `-a` (find acceptance cycles) option to `pan`.

When Spin finds a counter-example of a liveness property, it stores the counterexample in a trail-file as before. Its content can be viewed in the exact same way as the earlier counterexample traces. Note that for LTL properties, typically, the counterexample has the shape of a lasso: it gives a path to a state that is on a cycle which shows why the property does not hold.

Example 11.4.2. Consider the version of the alternating bit protocol with unreliable channels, that we adapted for verification by making `last_m` a global variable, and to which the two LTL properties have been added. We show the relevant updates in Listing 5. Assume this is stored in `abp-unreliable-ltl.pml`.

Note we ensure that only one LTL formula is not commented out. Verification can now be run using the following commaands.

```
spin -a abp-unreliable-ltl.pml
gcc -DNOREDUCE -o pan pan.c
./pan -a
```

If the property does not hold, the tool will report, e.g., `pan:1: acceptance cyle (at depth 24).` It also indicates that a counterexample has been stored.

Exercise 11.11 Using Spin, verify both properties that we formulated in the previous example. You will find that one property is true, and the other property is false. Explain why this is the case.

Exercise 11.12 In Exercise 10.9 we verified that when the game of tic-tac-toe terminates either we have a winner or we have a draw. This gives us partial correctness in the sense that our result holds if the process terminates. To obtain correctness, we however need to guarantee that the property holds eventually.

Extend your model of tic-tac-toe with an LTL formula that checks that eventually we either have a winner or a draw. Verify using Spin that the property is satisfied.

[1] See `http://spinroot.com/spin/Man/ltl.html`.

[2] See `http://spinroot.com/spin/Man/remoterefs.html` for an explanation

```
#define N       2 /* dimension of channels */
#define MAX     2 /* Send integers modulo MAX */

mtype = { MSG, ACK, ERROR };

chan fromS = [N] of { mtype, byte, bit };       /* From S to unreliable channel */
chan toR = [N] of { mtype, byte, bit };         /* From unreliable channel to R */
chan fromR = [N] of { mtype, bit };             /* From R to unreliable channel */
chan toS = [N] of { mtype, bit };               /* From unreliable channel to s */

byte last_m = MAX-1;            /* data of last error-free message, for verification */

ltl msg_zero { <>(last_m == 0) };
//ltl msg_one { <>(last_m == 1) };
proctype Receiver(chan in,out) {
  byte m;                       /* message data received */
  bit recvbit;                  /* alternation bit received */
  bit last_recvbit = 1;         /* receive bit of last error-free message */
```

Listing 5: Updates to alternating bit protocol for LTL verification.

Chapter 12

Labelled transition systems

The formal model we discussed so far, Kripke structures, focuses on the atomic propositions that hold in a particular state. PROMELA models allow to give a high-level description of Kripke structures. However, if we want to talk about communication between components, we need to label the transitions with the input and output actions that happen on these transitions. We silently used this when we used channels in PROMELA models.

Labelled transition systems (LTSs) provide an alternative formalization of state-based systems where the focus is exactly on the actions that happen when we take a transition, and we cannot see any information about states. LTSs can, for example, be used to automatically test implementations, and to give a precise formal semantics to UML state machine diagrams.

At the end of this chapter, you

- can describe a system using labelled transition systems;
- can explain the difference between labelled transition systems and Kripke structures;
- can explain non-determinism and its uses;
- can identify states, transitions, and traces.

12.1 Labelled transition systems

One approach to modelling the behavior of a system is to focus on the way it interacts with its environment. In doing so, we use basic *actions* to describe elementary communications as the key building block of a model. A labelled transition system (LTS) captures the *states* of the system, and has *transitions* labelled by actions to describe the evolution of the system when a particular action happens. An LTS can be visualized as a graph where vertices are the states and edges represent transitions between the states. Edges are labelled by actions. Depending on the properties that we are interested in, some system behavior may be irrelevant. Such behavior can be considered *internal*, i.e., we abstract away from the details of exactly what the system is doing when moving between states via an internal transition. Internal transitions are transitions labelled with the *internal action* τ (sometimes also referred to as the *silent action*).

Formally, labelled transition systems are defined as follows.

Definition 12.1.1. A *Labelled Transition System (LTS)* is a tuple $(S, s_0, Act, \rightarrow)$, where

- S is a set of states;
- $s_0 \in S$ is the initial state;
- Act is a set of actions (not including the internal action τ); and
- $\rightarrow \subseteq S \times Act \cup \{\tau\} \times S$ is a transition relation.

An LTS is *finite* if S and $\rightarrow$ are finite. With $s \xrightarrow{a} s'$, we denote that $(s, a, s') \in \rightarrow$.

Example 12.1.1. Figure 12.1 shows an example of an LTS modelling a simple beverage vending machine. The set of states of this LTS is $S = \{s_0, s_1, s_2, s_3\}$, with initial state s_0. The set of actions is

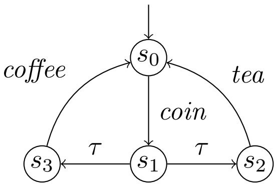


Figure 12.1: Labelled transition system of a simple machine producing either coffee or tea.

$\mathit{Act} = \{\mathit{coin}, \mathit{coffee}, \mathit{tea}\}$. Transition relation $\rightarrow = \{(s_0, \mathit{coin}, s_1), (s_1, \tau, s_2), (s_1, \tau, s_3), (s_2, \mathit{tea}, s_0), (s_3, \mathit{coffee}, s_0)\}$.

Depending on the context in which labelled transition systems are used, we sometimes see small variations. For instance, sometimes LTSs are described using sets of initial states, instead of a single initial state. In Chapter 14 we will see variations of LTSs in which we split the actions into input and output actions.

Exercise 12.1 Consider an oven that follows the same description as the oven in Exercise 9.1. Now, instead of a Kripke structure, create an LTS modelling the oven behavior. Again make sure that it satisfies the following requirements:

- The oven is never cooking with its door opened.
- The oven starts cooking only after the start button is pressed.
- After starting to cook, the oven eventually stops cooking.

The set of *all successors* of a state s is the set containing all states *reachable* from state s by performing a single step, not regarding the action.

$$\mathit{Succ}(s) = \{s' \in S \mid \exists \alpha \in \mathit{Act} \cup \{\tau\} \colon s \xrightarrow{\alpha} s'\}$$

In a similar way as for the successors, we can define the set of *predecessor* states of s that can reach s by performing a single α-step.

We refer to the set of *all predecessors* of state s as $\mathit{Pred}(s)$:

$$\mathit{Pred}(s) = \{s' \in S \mid \exists \alpha \in \mathit{Act} \cup \{\tau\} \colon s' \xrightarrow{\alpha} s\}$$

In other words, $\mathit{Pred}(s)$ contains all states that can reach state s by performing any action.

States that do not have outgoing transitions are referred to as *deadlock* states, or simply deadlocks. When modelling real systems they are typically undesired, and constitute a design error in the model.

Definition 12.1.2 (Deadlock state)**.** A state $s \in S$ is called *deadlock* if and only if $\mathit{Succ}(s) = \emptyset$.

Exercise 12.2 Give an LTS for a FIFO (first-in-first-out) queue of capacity 2 that can handle data items of value 0 and 1. Values are read by executing the action in_0 or in_1, reading 0 and 1 respectively. Values are output using the actions out_0 and out_1.

12.2 Traces

An LTS describes all possible behaviors of a system. All valid behaviors start from an initial state of the LTS. We describe the behaviors using *traces*. A trace is a (possibly empty) sequence of actions performed by the system. Given a set of actions Act, formally a trace is an element of $(\mathit{Act} \cup \{\tau\})^*$, where ϵ denotes the empty trace. We also sometimes use notation borrowed from regular expressions to describe (sets of) traces. For example, $a \cdot b$ is used to denote the trace consisting of action a followed by action b. We usually omit the $\cdot$ and simply write $a\ b$. Also, a^* denotes all traces that consist of zero or more repetitions of a.

To formalize the traces of an LTS, we first generalize the transition relation to allow for sequences of actions.

Definition 12.2.1 ((Strong) traces)**.** Let $(S, s_0, Act, \rightarrow)$ be an LTS. We define the generalized transition relation $\twoheadrightarrow$. Let $s, t \in S$, $a \in Act \cup \{\tau\}$, and $\sigma \in (Act \cup \{\tau\})^*$. Then

$$s \xrightarrow{\epsilon}\!\!\!\!\twoheadrightarrow s \text{ for all } s \in S$$

$$s \overset{a\sigma}{\twoheadrightarrow} t \text{ if } s \xrightarrow{a} s' \text{ and } s' \overset{\sigma}{\twoheadrightarrow} t \text{ for some } s' \in S$$

We write $s \overset{\sigma}{\twoheadrightarrow}$ if there is some state $t \in S$ such that $s \overset{\sigma}{\twoheadrightarrow} t$. The set of *strong traces* (sometimes referred to as traces) starting in state s is defined as

$$\mathit{traces}(s) = \{\sigma \mid s \overset{\sigma}{\twoheadrightarrow}\}$$

The traces of an LTS are the traces of its initial state.

Note that every state can reach itself by executing the empty sequence of actions ϵ, and there is a trace $a\sigma$ from one state s to a state t if from s we can do a transition labelled a to some state s', from which there is trace σ from s' to t.

Example 12.2.1. Reconsider Figure 12.1. All valid traces of the LTS in Figure 12.1 consist of zero or more repetitions of the following two sequences:

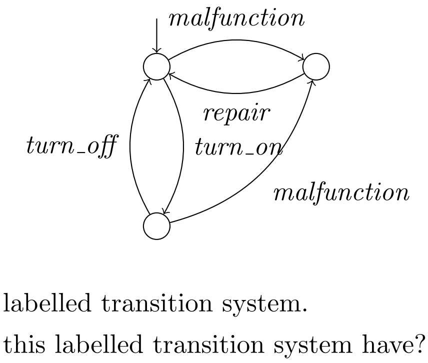
1. *coin* τ *coffee*
2. *coin* τ *tea*

Followed by a prefix of these traces. Concretely, the prefixes are the following:

1. ϵ
2. *coin*
3. *coin* τ

So, for instance, *coin* τ *tea coin* τ *coffee coin* τ is a sequence, as is *coin*.

Exercise 12.3 At a high level of abstraction, the engine of a car can be seen as something that can turn on and off. Also, the engine can malfunction, in which case it can be repaired. A labelled transition system modelling the behavior of such an engine is depicted below. It consists of 3 states, 4 distinct actions, and 5 labelled transitions.



1. Give five traces of this labelled transition system.
2. How many traces does this labelled transition system have?

Observe that traces are very closely related to paths. Paths are sequences of the *states* that can be reached from a given state, but do not include the transitions. Traces instead give the *actions* that appear on the transitions, and do not include the states.

12.3 Weak traces

Internal transitions abstract away from some internal behavior of a system. It is assumed that an external observer cannot see the internal actions. Also, the observer cannot see in which state the system currently is. So far, our definition of traces does not consider only observable actions, since it does not treat the internal action τ specially.

We can restrict traces to consider sequences of observable actions by abstracting from the internal action τ. If s can do the trace *coin* τ *coffee*, i.e. $s \xrightarrow{coin\ \tau\ coffee}\!\!\!\rightarrow t$, then we write $s \xRightarrow{coin\ coffee} t$ for the sequence of observable actions. We say that s is able to perform *weak trace coin coffee.* A weak trace is an element of Act^*. Note that we write Act^+ for the weak traces containing at least one observable action.

Formally, the weak traces of a (state in an) LTS are defined as follows.

Definition 12.3.1 (Weak traces)**.** Let $(S, s_0, Act, \rightarrow)$ be an LTS with $s, t \in S$. We first define the weak generalized transition relation $\Longrightarrow$ as follows. Let $a \in Act$, $s, t \in S$, and $\sigma \in Act^+$.

$$s \overset{\epsilon}{\Longrightarrow} t \text{ if } s \overset{\tau^*}{\twoheadrightarrow} t$$

$$s \overset{a}{\Longrightarrow} t \text{ if } \exists s_1, s_2 : s \overset{\epsilon}{\Longrightarrow} s_1 \xrightarrow{a} s_2 \overset{\epsilon}{\Longrightarrow} t$$

$$s \overset{a\sigma}{\Longrightarrow} t \text{ if } \exists s' \in S. s \overset{a}{\Longrightarrow} s' \wedge s' \overset{\sigma}{\Longrightarrow} t$$

We write $s \overset{\sigma}{\Longrightarrow}$ if there is some state $t \in S$ such that $s \overset{\sigma}{\Longrightarrow} t$. The set of weak traces starting in state s is defined as

$$wtraces(s) = \{\sigma \mid s \overset{\sigma}{\Longrightarrow}\}$$

The weak traces of an LTS are the weak traces of its initial state.

Example 12.3.1. Consider the labelled transition system from Figure 12.1. This LTS has infinitely many weak traces. Some concrete examples are:

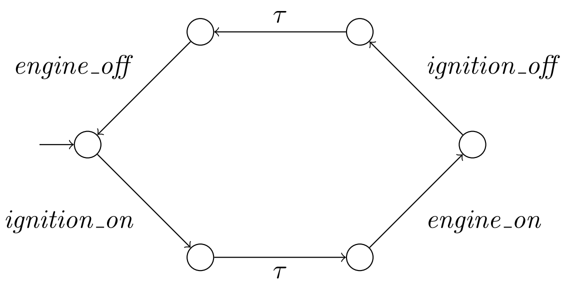
- ϵ
- *coin*
- *coin coffee*
- *coin tea*

All other weak traces of this LTS can be built by repeating the last two weak traces zero or more times, followed by one of the first two weak traces. Note that, in particular, none of the weak traces can contain the internal action τ.

Exercise 12.4 Consider the ignition of a car. Initially, the ignition is off. When the ignition is turned on, a signal is given to the engine that it must start, and eventually the engine turns on. Likewise, when the ignition is on, the ignition can be turned off, the engine is signalled to stop, and the engine shuts off.

From the point of view of the driver, we could consider the start and stop signals internal to the system, and we can only observe whether the ignition is turned on or off, and whether the engine is running, we therefore do not model start and stop explicitly, but consider them internal actions (τ).

This situation is depicted in the following LTS.



1. Give one weak trace of this LTS that contains at least two different actions.
2. Give one weak trace of this LTS that contains at least four different actions.

We also define the set of states that is reached after executing a weak trace σ.

Definition 12.3.2 (after). Let $(S, s_0, Act, \rightarrow)$ be an LTS with $s \in S$. Given a a trace $\sigma \in Act^*$, the set of states reached from s **after** executing σ is defined as follows:

$$s \textbf{ after } \sigma = \{t \in S \mid s \overset{\sigma}{\Longrightarrow} t\}$$

In other words, s **after** σ contains all states that can be reached from state s by performing weak trace σ. We generalize this to sets of states $S' \subseteq S$:

$$S' \textbf{ after } \sigma = \bigcup_{s \in S'} s \textbf{ after } \sigma$$

In a similar way, we can define the set of all states that can be reached from a given state s is denoted $\mathit{Reach}(s)$.

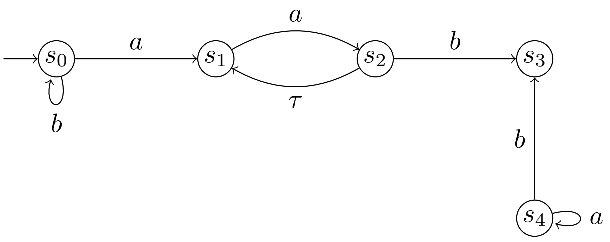
Definition 12.3.3 (Reachable states)**.** Let $(S, s_0, Act, \rightarrow)$ be an LTS with $s \in S$. The set of states reachable from s is defined as follows:

$$\mathit{Reach}(s) = \{t \in S \mid \exists \sigma \in Act^*. s \overset{\sigma}{\Longrightarrow} t\}$$

We refer to $\mathit{Reach}(s_0)$ as the reachable states of the LTS.

In the rest of the reader we typically assume that all states $s \in S$ are reachable from the initial state, i.e., $S = \mathit{Reach}(s_0)$.

Exercise 12.5 Consider the following labelled transition system.



Determine each of the following sets.

1. $\mathit{Reach}(s_0)$
2. s **after** $b\ a$
3. s **after** $a\ a$
4. s **after** $a\ a\ b$
5. s **after** $a\ b$

12.4 Non-determinism

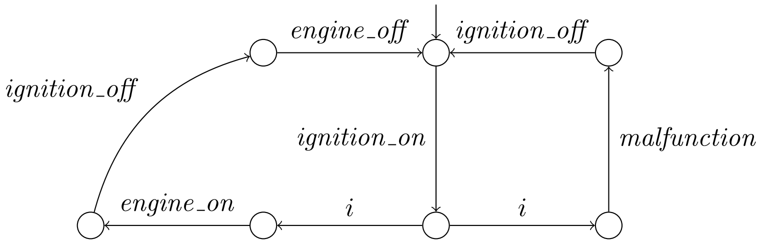
The close reader may have already observed that from a particular state s in an LTS, there can be multiple outgoing transitions with the same label. In such a case, we call the LTS non-deterministic. More precisely, an LTS is *deterministic* if it defines for every state $s \in S$ and action $a \in Act$ at most one target state.

For an LTS without internal transitions, determinism is defined as follows.

Definition 12.4.1 (Deterministic LTS (without internal transitions))**.** Let $A = (S, s_0, Act, \rightarrow)$ be an LTS such that for all $s \xrightarrow{a} t$, $a \neq \tau$. LTS A is *deterministic* if and only if for all $s, t, t' \in S$ and $a \in Act$, $s \xrightarrow{a} t \wedge s \xrightarrow{a} t' \implies t = t'$.

If an LTS is not deterministic, we call it *non-deterministic*.

Example 12.4.1. One could argue that the LTS modelling the engine in Exercise 12.3 is unrealistic. For instance, why does the engine always have to be repaired when it malfunctions? Why can the engine malfunction if it is turned off? We therefore, instead, consider only a very specific type of malfunction. The engine can be turned on and off. If the ignition is turned on, then non-deterministically the engine turns on, or it does not. An LTS modelling such an engine is shown below.



In this LTS we use the action i to model the nondeterministic option to malfunction. Essentially, the action i here models some internal behavior of the engine.

When considering LTSs with internal transitions, we need to be a bit more careful when defining determinism. Several alternatives have been described in the literature. For instance, one could simply apply our previous definition also to LTSs with internal transitions. Here, however, we will follow the definition as it is given in the article on which we base Chapter 14.

Definition 12.4.2 (Deterministic LTS (with internal transitions))**.** Let $A = (S, s_0, Act, \rightarrow)$ be an LTS. We say A is *deterministic* iff we have for all $s \in S$ and $a \in Act$ that $|s \textbf{ after } a| \leq 1$.

Note that as an immediate consequence of this definition, any LTS with states s, t, t' such that $s \xrightarrow{a} t \xrightarrow{\tau} t'$ is non-deterministic by definition, since $s \textbf{ after } a = \{t, t'\}$.

For LTSs without internal actions, the two definitions coincide.

Example 12.4.2. Recall the LTS shown in Figure 12.1. This LTS is non-deterministic, since from state s_0, two states are reachable by following the *coin* transition followed by a τ-transition. More formally, $s_0 \textbf{ after } \mathit{coin} = \{s_1, s_2, s_3\}$.

Exercise 12.6

1. Draw a labelled transition system modelling a light switch with actions *on* and *off*.
2. Draw a labelled transition system that extends this light switch such that switching on/off and their effect are both modelled explicitly. Use actions *switch_on*, *switch_off* as well as *on* and *off*.
3. Now, draw a labelled transition system that extends this light switch with a nondeterministic option to break when switching on the light. Use an action i to represent the choice. When broken, no action is possible anymore.

12.5 Parallel composition

To model complex systems, a good approach is to first build simple blocks. Second, these basic blocks are composed to form a more complex system. We already saw some examples of this in Chapter 10. We here consider the composition of labelled transition systems using handshaking communication. The idea is to define a set of handshaking actions, called H. Two labelled transition systems communicate via H by performing actions in H together. That is, the two LTSs need to synchronize on all actions in H. For actions outside H, each LTS evolves independently of the other. Formally, this composition is defined as follows.

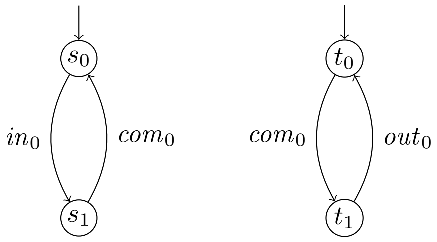
Definition 12.5.1 (Handshaking parallel composition of LTSs)**.** Let $L_1 = (S_1, s_{1,0}, Act_1, \to_1)$ and $L_2 = (S_2, s_{2,0}, Act_2, \to_2)$ be two LTSs, and assume a set of handshaking actions $H \subseteq Act_1 \cap Act_2$. We define the parallel composition $L_1 \parallel_H L_2$ as the LTS $(S, s_0, Act, \to)$, where

- $S = S_1 \times S_2$,
- $s_0 = (s_{1,0}, s_{2,0})$,
- $Act = Act_1 \cup Act_2$,
- Transition relation $\to$ is the least set of transitions satisfying the following:

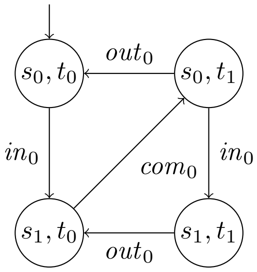
$$\frac{s_1 \xrightarrow{a} s_2 \quad t_1 \xrightarrow{a} t_2}{(s_1, t_1) \xrightarrow{a} (s_2, t_2)}\ a \in H \qquad \frac{s_1 \xrightarrow{a} s_2}{(s_1, t_1) \xrightarrow{a} (s_2, t_1)}\ a \notin H \qquad \frac{t_1 \xrightarrow{a} t_2}{(s_1, t_1) \xrightarrow{a} (s_1, t_2)}\ a \notin H$$

Note that when $a \notin H$, it may be the case that $a = \tau$, i.e., the internal transition does not communicate, and is executed locally.

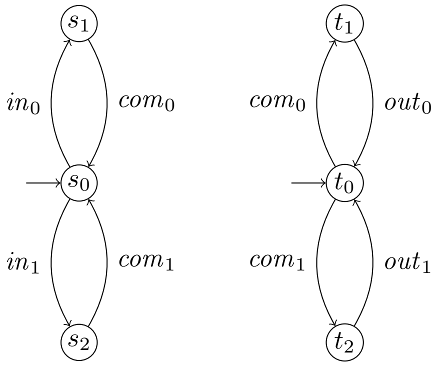
Example 12.5.1. A one-place buffer can read an element along its input, and output that element along its output. Consider the following pair of labelled transition systems that can both input and output the element 0. The first buffer inputs using the action in_0, and outputs using com_0, and the second buffer inputs using com_0 and outputs using out_0. The LTSs are as follows:



Let $H = \{com_0\}$, so the LTSs synchronize on com_0, such that the element output by the first buffer is moved to the second buffer. The resulting parallel composition is the following.



Exercise 12.7 We extend the buffers from the previous example, such that the buffers can read the values 0 and 1, instead of just 0. So, the labelled transition systems are the following. The LTSs are as follows:



Let $H = \{com_0, com_1\}$. Give the labelled transition system that corresponds to the handshaking parallel composition of these labelled transition systems.

12.6 Applications

Similar to Kripke structures, labelled transition systems can also be used to verify (model check) systems. For labelled transition systems, typical logics that are used are Hennessy-Milner logic and the modal mu-calculus (instead of LTL, CTL or CTL*). These logics are out of the scope of this course, but they are for instance covered in the bachelor course *process theory*, and master's courses such as *system validation* and *algorithms for model checking*. Labelled transition systems can also be used to automatically generate test cases in model based testing. This is covered in Chapter 14. Labelled transition systems also form the basis for formalizing the semantics of other, higher-level specification formalisms. We show how (a subset of) UML state machine diagrams can be formalized in terms of LTSs in the next chapter.

Chapter 13

Formal state machine diagrams

In Chapter 6, we introduced UML state machine diagrams. In Chapters 9 and 12, we discussed Kripke structures and labelled transition systems. In all of these formalisms, we reason about systems in terms of states and transitions between states. However, state machine diagrams offer a more high-level description of the behavior of a system, which allows for a more concise specification. The downside of this is that it is rather difficult to reason about the behavior described by a state machine diagram. To enable formal reasoning about the behavior described by UML state machine diagramså we need to give their semantics.

In this chapter, we present such a semantics using labelled transition systems. To keep things simple, we restrict ourselves to a small subset of UML state machine diagrams. The formalization in this chapter is inspired by a formalization of Harel's state charts by Mousavi in an earlier version of this course. A similar formalization is given in Lilius and Paltor (1999).

At the end of this chapter, you

- can formally describe the syntax of state machine diagrams;
- can explain the semantics of state machine diagrams;
- can translate simple state machine diagrams into labelled transition systems;
- can formally reason about state machine diagrams.

13.1 State machine diagram syntax

In Chapter 6 we have seen that state machine diagrams contain hierarchies of (pseudo-) states that are connected by transitions. The transitions are triggered by events, enabled by guards, and they result in activities. Also, states have enter, exit and do activities. It is important to observe that events can also appear as activities; in this case, occurrence of an event means that it is either sent to the environment (to be processed by another state machine), or to the state machine itself. In this chapter the only activities we consider are events.

To formally define the translation of (a subset of) state machine diagrams into LTSs, we first need to formally capture the syntax (and structure) of state machine diagrams. This is done in the rest of this section.

13.1.1 States

We first define the state structure of a finite state machine. The state structure records for all states their type, the children, and whether they are initial or not. Recall from Chapter 6 that a state can either be simple or composite. If a state is composite, it can be orthogonal. The UML standard prescribes three Booleans for every state, recording whether it is simple, composite and orthogonal. In theory, this would give 8 possibilities, whereas only three options are valid: a state is either simple, composite and non-orthogonal, or composite and orthogonal. Here we therefore we opt to define these three types of states directly, referring to a simple state as *basic*, a composite, non-orthogonal state as an *or-state*, and a composite, orthogonal state as an *and-state*. In the formalization, the regions within an orthogonal state are themselves treated as or-states.

Definition 13.1.1 (State structure)**.** The *state structure* of a state machine diagram is a 4-tuple $\mathcal{S} = (S, \mathit{type}, \mathit{chld}, \mathit{initial})$, where

- S is a (finite) set of states;
- $\mathit{type} : S \to \{\mathit{Basic}, \mathit{Or}, \mathit{And}\}$ is a total function indicating the type of each state. A state can be a *simple*-state, a *composite*-state, or an *orthogonal*-state;
- $\mathit{chld} : S \to 2^S$ is a total function defining for each state the set of children that is structurally contained within that state;
- $\mathit{initial} : S \to \mathbb{B}$ is a total function indicating whether a state is initial or not.

The state types described above correspond directly to the types in the state machine diagrams. We do take the initial pseudo-states into account, but do not model them explicitly as a state in the definition above, instead we mark all states connected to an outgoing transition from an initial state as initial. Note that regions of an orthogonal state are always initial.

We assume that a defined state structure is *well-formed.* In particular, this requires that basic states have no children, and that or-states have no more that one child which is initial.

For convenience, we always consider a state diagram D to be completely contained inside one composite state at the highest hierarchical level, which we call root.

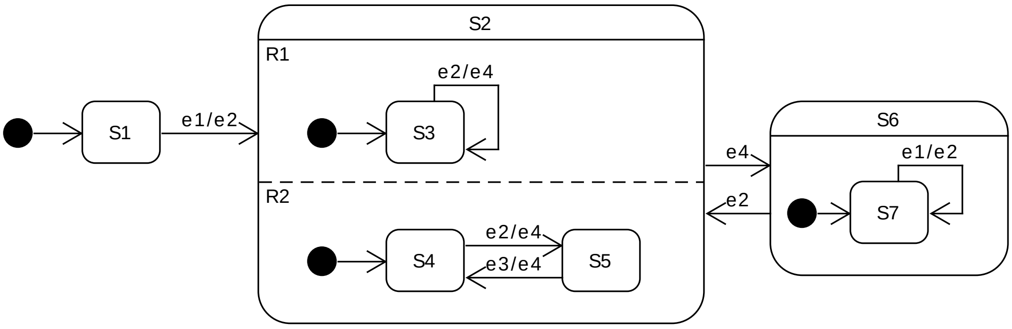


Figure 13.1: A hierarchical state diagram with sub-machines and regions

Example 13.1.1. Figure 13.1 shows an example state machine diagram with all three kinds of states, and two regions within the orthogonal state. For this diagram, we have:

- $S = \{\mathsf{root}, \mathsf{S1}, \ldots, \mathsf{S7}, \mathsf{R1}, \mathsf{R2}\}$,
- $\mathit{type}(\mathsf{S1}) = \mathit{type}(\mathsf{S3}) = \mathit{type}(\mathsf{S4}) = \mathit{type}(\mathsf{S5}) = \mathit{type}(\mathsf{S7}) = \mathit{Basic}$, $\mathit{type}(\mathsf{S2}) = \mathit{And}$, and $\mathit{type}(\mathsf{root}) = \mathit{type}(\mathsf{S6}) = \mathit{type}(\mathsf{R1}) = \mathit{type}(\mathsf{R2}) = \mathit{Or}$,
- $\mathit{chld}(\mathsf{root}) = \{\mathsf{S1}, \mathsf{S2}, \mathsf{S6}\}$, $\mathit{chld}(\mathsf{S2}) = \{\mathsf{R1}, \mathsf{R2}\}$, $\mathit{chld}(\mathsf{R1}) = \{\mathsf{S3}\}$, $\mathit{chld}(\mathsf{R2}) = \{\mathsf{S4}, \mathsf{S5}\}$ and $\mathit{chld}(\mathsf{S6}) = \{\mathsf{S7}\}$. All other states have an empty set of children.
- Finally, for $s \in \{\mathsf{root}, \mathsf{S1}, \mathsf{S3}, \mathsf{S4}, \mathsf{S7}, \mathsf{R1}, \mathsf{R2}\}$, we have $\mathit{initial}(s) = \text{true}$. For all other states, this value is false.

In the rest of this section, we fix a state machine diagram with state structure $\mathcal{S} = (S, \mathit{type}, \mathit{chld}, \mathit{initial})$. To further characterize the state structure, we define the (strict) descendants, containing all states contained in a state, and the (strict) ancestors, capturing all states in which a particular state is contained.

For a state $s \in S$, its *strict descendants*, denoted $\mathit{sdesc}(s)$, are all its children, as well as their strict descendants.

Definition 13.1.2 ((Strict) descendants)**.** For state $s \in S$, the set of *strict descendants*, denoted $\mathit{sdesc}(s)$, is the smallest set of states satisfying:

- $\mathit{chld}(s) \subseteq \mathit{sdesc}(s)$; and
- $\forall s, s' \in S . s' \in \mathit{chld}(s) \wedge s'' \in \mathit{sdesc}(s') \implies s'' \in \mathit{sdesc}(s)$.

The *descendants* of s, denoted $\mathit{desc}(s)$ are defined as $\mathit{desc}(s) = \{s\} \cup \mathit{sdesc}(s)$.

Example 13.1.2. Recall the example in Figure 13.1. We have, for instance,

$$\mathit{sdesc}(\mathsf{S2}) = \{\mathsf{R1}, \mathsf{S3}, \mathsf{R2}, \mathsf{S4}, \mathsf{S5}\} \qquad \mathit{desc}(\mathsf{S2}) = \{\mathsf{S2}, \mathsf{R1}, \mathsf{S3}, \mathsf{R2}, \mathsf{S4}, \mathsf{S5}\}$$
$$\mathit{sdesc}(\mathsf{S6}) = \{\mathsf{S7}\} \qquad \mathit{desc}(\mathsf{S6}) = \{\mathsf{S6}, \mathsf{S7}\}$$

For a state $s \in S$, its *(strict) ancestors* are all states in which s is (strictly) contained.

Definition 13.1.3 ((Strict) ancestors)**.** For state $s \in S$, the set of *strict ancestors*, denoted $\mathit{sanc}(s)$ and the *ancestors*, denoted $\mathit{anc}(s)$ are defined as follows:

$$\mathit{sanc}(s) = \{s' \in S \mid s \in \mathit{sdesc}(s')\}$$
$$\mathit{anc}(s) = \{s' \in S \mid s \in \mathit{desc}(s')\}$$

Example 13.1.3. Recall the example in Figure 13.1. We have, for instance,

$$\mathit{sanc}(\mathsf{S2}) = \{\mathsf{root}\} \qquad \mathit{anc}(\mathsf{S4}) = \{\mathsf{root}, \mathsf{S2}, \mathsf{R2}, \mathsf{S4}\}$$

For a non-empty set of states S', we define their *least common or-ancestor*. The least common or-ancestor of a non-empty set of states is the or-state having all the states in the set as descendants, while being closest (in hierarchy distance) to these states compared to any other or-state that has the states as descendants. Note that the least common strict or-ancestor $\mathit{lcsoa}(S')$ exists for $\emptyset \subset S' \subseteq S$, and is uniquely.

Definition 13.1.4 (Least common strict or-ancestor)**.** The *least common strict or-ancestor*, lcsoa: $2^S \rightarrow S$ is defined as follows. For $\emptyset \subset S' \subseteq S$, $\mathit{lcsoa}(S')$ is the (unique) state satisfying each of the following conditions:

- $\forall s \in S' . \mathit{lcsoa}(S') \in \mathit{sanc}(s)$, i.e., it is a strict ancestor of each of the states in S',
- $\mathit{type}(\mathit{lcsoa}(S')) = \mathit{Or}$, i.e., it is an or-state, and
- $\forall s \in S . \mathit{type}(s) = \mathit{Or} \wedge (\forall s' \in S' . s \in \mathit{sanc}(s')) \implies \mathit{lcsoa}(S') \in \mathit{desc}(s)$, i.e., the least common strict or ancestor is a descendant of each of the or-states that are strict ancestors of all states S'. This characterizes the deepest such or-state.

Example 13.1.4. Recall the state machine diagram from Figure 13.1. We have the following:

$$\mathit{lcsoa}(\{\mathsf{S4}, \mathsf{S5}\}) = \mathsf{R2} \qquad \mathit{lcsoa}(\{\mathsf{S3}, \mathsf{S6}\}) = \mathsf{root}$$
$$\mathit{lcsoa}(\{\mathsf{S2}\}) = \mathsf{root} \qquad \mathit{lcsoa}(\{\mathsf{S3}\}) = \mathsf{R1}$$
$$\mathit{lcsoa}(\{\mathsf{R2}\}) = \mathsf{R2}$$

Note that the least common strict or-ancestor of S3 is R1, and not S3, since S3 is not an or-state.

13.1.2 Transitions

Besides the state structure, the *transition relation* of a state diagram also needs to be formalized. We do this as follows.

Definition 13.1.5 (Transition structure)**.** The *transition structure* of a state diagram $\mathcal{T}$ is a set of transitions tr defined by 5-tuples (s_s, E_t, C, E_g, s_t), where

- $s_s \in S$ is the *source* state.

- $E_t \subseteq \mathcal{E}$ is the set of *triggers*, with $\mathcal{E}$ being the set of all events. Triggers are events that enable the transition whenever they occur; note that only one event in E_t needs to be available to trigger the transition.
- $C \subseteq S$ is the set of *guards*. As guards, we only consider references to diagram states. The guards express that a transition is only enabled if all states in C are part of the current configuration.
- $E_g \subseteq \mathcal{E}$ is the set of *activities* or *generated events*. These events are fired whenever the transition is followed.
- $s_t \in S$ is the *target* state.

When E_t is a singleton set $\{e\}$, we sometimes write e instead of $\{e\}$, likewise for E_g.

In this description of transitions we have already significantly limited the state machine diagrams that we formalize. We only allow transitions to proper states, and not pseudo-states, we do not allow compound transitions. The guards are limited to whether a given state is current or not, and we only allow generating trigger events, instead of arbitrary activities.

Example 13.1.5. Recall the state machine diagram from Figure 13.1. This state machine diagram has the following transitions:

- $(\mathsf{S1}, \mathsf{e1}, \emptyset, \mathsf{e2}, \mathsf{S2})$
- $(\mathsf{S2}, \mathsf{e4}, \emptyset, \emptyset, \mathsf{S6})$
- $(\mathsf{S3}, \mathsf{e2}, \emptyset, \mathsf{e4}, \mathsf{S3})$
- $(\mathsf{S4}, \mathsf{e2}, \emptyset, \mathsf{e4}, \mathsf{S5})$
- $(\mathsf{S5}, \mathsf{e3}, \emptyset, \mathsf{e4}, \mathsf{S4})$
- $(\mathsf{S6}, \mathsf{e2}, \emptyset, \emptyset, \mathsf{S2})$
- $(\mathsf{S7}, \mathsf{e1}, \emptyset, \mathsf{e2}, \mathsf{S7})$

A state machine is now described as $D = (\mathcal{S}, \mathcal{T})$.

13.2 State machine diagram semantics

We now proceed by describing the behavior of a state machine diagram when it is executed. In this section fix a state machine diagram $D = (\mathcal{S}, \mathcal{T})$, where $\mathcal{S} = (S, \mathit{type}, \mathit{chld}, \mathit{initial})$.

13.2.1 States

The current system state is described by a combination of states, at most one per hierarchical level. In Chapter 6 we have already seen that such combinations are called *state configurations*, or configurations, for short. In that chapter, we used configurations informally. Here we formalize this notion.

Definition 13.2.1 (State configuration)**.** For a given state $s \in S$, a set of states $\mathit{conf} \subseteq \mathit{desc}(s)$ is a *state configuration* w.r.t. s if:

- $s \in \mathit{conf}$, i.e., s is in all configurations rooted at s;
- $\forall s' \in \mathit{conf}.\mathit{type}(s') = \mathit{Or} \implies \exists s'' \in S.\mathit{chld}(s') \cap \mathit{conf} = \{s''\}$, i.e., for every or-state, exactly one child is part of the configuration; and
- $\forall s' \in \mathit{conf}.\mathit{type}(s') = \mathit{And} \iff \mathit{chld}(s') \subseteq \mathit{conf}$, i.e., for every and-state, all children (the regions) are part of the configuration.

Example 13.2.1. Recall the state machine diagram from Figure 13.1, the set $\{\mathsf{S2}, \mathsf{R1}, \mathsf{R2}, \mathsf{S3}, \mathsf{S4}\}$ would constitute a valid configuration. However, the set $\{\mathsf{S2}, \mathsf{R1}, \mathsf{S3}\}$ would not, since *all* children of and-state $\mathsf{S2}$ must be part of a configuration of which $\mathsf{S2}$ is part.

The operational state of a state machine diagram not only involves the current configuration, it also requires recording the available events. The UML standard for this prescribes the presence of an *event pool*, to which the environment can add events, and from which a scheduler dispatches one event at a time. The event that is dispatched is completely processed by the state machine diagram, before a next event can be dispatched. This is called run-to-completion semantics. The standard does not formalize exactly how this event pool mechanism works, and how dispatching is done. In practice, it is likely that the event pool is implemented as a queue with a scheduler that dispatches the next event. Here we choose an abstraction which allows the environment to add an arbitrary set of events to the event pool at any time, and that an arbitrary event that is currently in the event pool can be dispatched.[1] Events that appear as activities will be added to this event pool as well.

A *situation* in the semantics of a state machine diagram now is a configuration, along with the events that are currently in the event pool.

Definition 13.2.2 (Situation)**.** A *situation* is a pair $st = (conf, Ev)$, where *conf* is a configuration w.r.t. the root and *Ev* is a set of events, representing the event pool.

13.2.2 Transitions

From a given situation $(conf, Ev)$, an event $e \in Ev$ can be dispatched. This results in a new situation $(conf, Ev \setminus \{e\})$, with dispatched event e (so we remember the dispatched event). This event can enable a number of transitions.

Definition 13.2.3 (Enabled transitions)**.** Transition $tr = (s_s, E_t, C, E_g, s_t)$ is *enabled* at situation $(conf, Ev)$ with dispatched event e if and only if

- $s_s \in conf$, i.e., the source state is in the current configuration;
- $e \in E_t$, i.e., the dispatched event is a trigger of the transition; and
- $C \subseteq conf$, i.e., the guard is satisfied.

Given state machine diagram $D = (\mathcal{S}, \mathcal{T})$, situation *st*, and triggered event e, we use $enabled(st, e)$ to denote the set of all enabled transitions.

It is possible for more than one transition to be enabled at the same time within a state machine diagram. Some of these transitions can fire at the same time, others are conflicting. Transitions are conflicting if the set of states they exit overlap. To calculate the set of states that a transition exits, we first define its scope.

Definition 13.2.4 (Transition scope)**.** Given a transition $tr = (s_s, E_t, C, E_g, s_t)$, its scope is defined as follows:

$$scope(tr) = lcsoa(\{s_s, s_t\}).$$

The scope of a transition is the least common or-ancestor of the source and target state of the transition, i.e., it is the deepest or-state that contains both the source and the target of the transition.[2]

Example 13.2.2. Reconsider the state machine diagram from Figure 13.1. We have:

$$scope((\mathsf{S1}, \mathsf{e1}, \emptyset, \mathsf{e2}, \mathsf{S2})) = \mathsf{root} \qquad scope((\mathsf{S3}, \mathsf{e2}, \emptyset, \mathsf{e4}, \mathsf{S3})) = \mathsf{R1}$$
$$scope((\mathsf{S4}, \mathsf{e2}, \emptyset, \mathsf{e4}, \mathsf{S5})) = \mathsf{R2} \qquad scope((\mathsf{S2}, \mathsf{e4}, \emptyset, \emptyset, \mathsf{S6})) = \mathsf{root}$$

The *exit set* of a transition *tr* is the part of the current situation that is left when following *tr*. This set contains all states strictly contained in the scope of the transition that are ancestors of the source state.

To compute this, we first determine the unique exit state of the transition, which is the child of the scope of the transition that is also an ancestor of the source state. Given a configuration, the exit set contains all states in the configuration that are descendants of the unique exit state. Formally this is defined as follows.

[1] The event pool and the exact way it is handled is one of the variation points that the UML standard allows. Different tools may here make different choices.

[2] Note that whenever the source and target state of a transition are different, the least common strict or-ancestor of the source and target state is the same as the least common or-ancestor. However, for self-loops, i.e., the source and target state coincide, we need a strict ancestor to obtain a proper definition of unique exit state.

Definition 13.2.5 (Exit set of a transition)**.** Let $tr = (s_s, E_t, C, E_g, s_t)$ be a transition. The *unique exit state* of *tr*, denoted *UExt*(*tr*) is the single state $s \in S$ such that $s \in \mathit{anc}(s_s) \cap \mathit{chld}(\mathit{scope}(tr))$.

The *exit set* of *tr*, given configuration *conf* is defined as

$$\mathit{Ext}(tr, \mathit{conf}) = \mathit{conf} \cap \mathit{desc}(\mathit{UExt}(tr)).$$

Example 13.2.3. Again consider Figure 13.1. Consider transition $tr = (\mathsf{S4}, \mathsf{e2}, \emptyset, \mathsf{e4}, \mathsf{S5})$. The unique exit state of the transition is S4, since S4 is both an ancestor of itself and a child of R2, the scope of the transition. Given configuration $\mathit{conf} = \{\mathsf{root}, \mathsf{S2}, \mathsf{R1}, \mathsf{S3}, \mathsf{R2}, \mathsf{S4}\}$, $\mathit{Ext}(tr, \mathit{conf}) = \{\mathsf{S4}\}$, since $\mathit{anc}(\mathsf{S4}) = \{\mathsf{root}, \mathsf{S2}, \mathsf{R2}, \mathsf{S4}\}$, and $\mathit{chld}(\mathsf{R2}) = \{\mathsf{S4}, \mathsf{S5}\}$.

The unique exit state of the transition $tr_1 = (\mathsf{S3}, \mathsf{e2}, \emptyset, \mathsf{e4}, \mathsf{S3})$ is S3, and for the same configuration *conf*, we have $\mathit{Ext}(tr_1, \mathit{conf}) = \{\mathsf{S3}\}$.

We can now formally define when two transitions are conflicting.

Definition 13.2.6 (Conflicting transitions)**.** Given configuration *conf*, transitions tr_1 and tr_2 are conflicting if and only if $\mathit{Ext}(tr_1, \mathit{conf}) \cap \mathit{Ext}(tr_2, \mathit{conf}) \neq \emptyset$.

When a transition is fired, the set of states that is entered in the new configuration that is reached is defined in a similar way, using a unique enter state.

Definition 13.2.7 (Enter set)**.** Let $tr = (s_s, E_t, C, E_g, s_t)$ be a transition. The *unique enter state* of *tr*, denoted *UEnt*(*tr*), is the single state $s \in S$ such that $s \in \mathit{anc}(s_t) \cap \mathit{chld}(\mathit{scope}(tr))$.

The *enter set* of *tr*, denoted *Ent*(*tr*), is a (sub)configuration with respect to *UEnt*(*tr*) that is such that $s_t \in \mathit{Ent}(tr)$, and for all $s \in \mathit{Ent}(tr) \setminus \mathit{anc}(s_t)$, we have *initial*(*s*).

In other words, the enter set of *tr* is the set of states that we enter when following transition *tr*. Whenever the target state of a transition does not provide us a particular new basic-state, we enter the default state(s) of a reached and- or or-state. Note that we do not have to consider the ancestors of s_t, since the definition of configuration forces some of these states to be part of the enter set.

Example 13.2.4. Again consider Figure 13.1 with transition $tr = (\mathsf{S4}, \mathsf{e2}, \emptyset, \mathsf{e4}, \mathsf{S5})$. The unique enter state of the transition is S5, since S5 is an ancestor of itself and a child of R2, the scope of the transition. The enter set of the transition, *Ent*(*tr*), is {S5}, since this is a basic-state.

Example 13.2.5. Again consider Figure 13.1. We focus on the transition $tr = (\mathsf{S2}, \mathsf{e4}, \emptyset, \emptyset, \mathsf{S6})$. The unique exit state of the transition is S2, the unique enter state is S6. Given configuration $\mathit{conf} = \{\mathsf{root}, \mathsf{S2}, \mathsf{R1}, \mathsf{S3}, \mathsf{R2}, \mathsf{S4}\}$, $\mathit{Ext}(tr, \mathit{conf}) = \{\mathsf{S2}, \mathsf{R1}, \mathsf{S3}, \mathsf{R2}, \mathsf{S4}\}$, i.e., S2 and all descendants of S2 that are part of the current configuration. The enter set of the transition, *Ent*(*tr*), is {S6, S7}, that is, the set consisting of the target state of the transition, S6, plus the initial states contained in S6.

Exercise 13.1 Consider the UML state machine diagram in Figure 13.2. This is the same state machine as in Exercise 6.4, where we determined the types of some of the states, and all its configurations. Note that the transition c[S2] has S2 as guard

1. List all transitions in the state machine diagram.
2. For each of the transitions in the state machine diagram, give their scope.
3. For each of the transitions in the state machine diagram, give a configuration and trigger event that enable the transition, and with respect to then configuration, give the unique exit state, the exit set, the unique enter state, and the enter set of the transition.
4. Suppose the state machine is in configuration {Q, U, S, T, S2, T1}, with dispatched event c. Are there any conflicting transitions? Explain why (not).

Exercise 13.2 Consider the UML state machine diagram in Figure 13.3. Note that, e.g., the transition labelled d[S1] has S1 as guard, and e[S1, T2] has both S1 and T2 as guard.

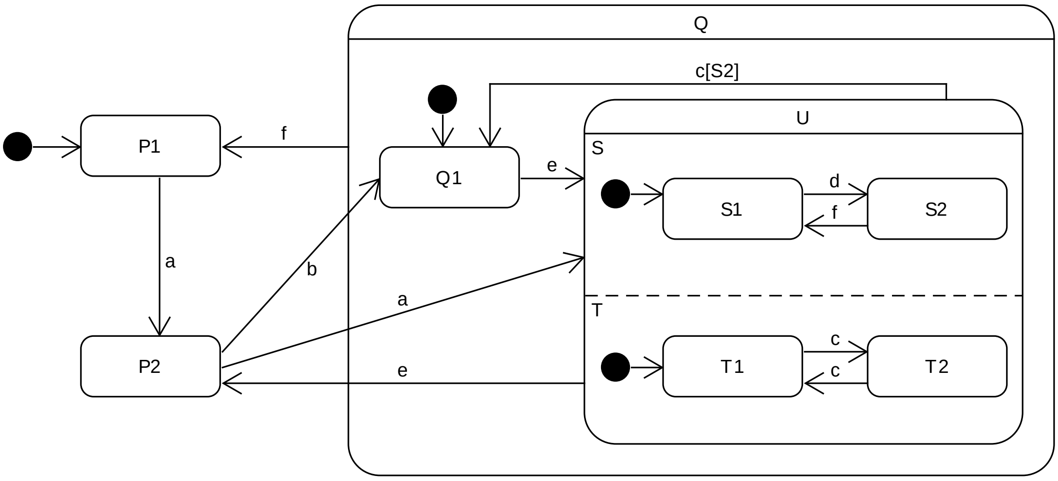


Figure 13.2: A state machine diagram.

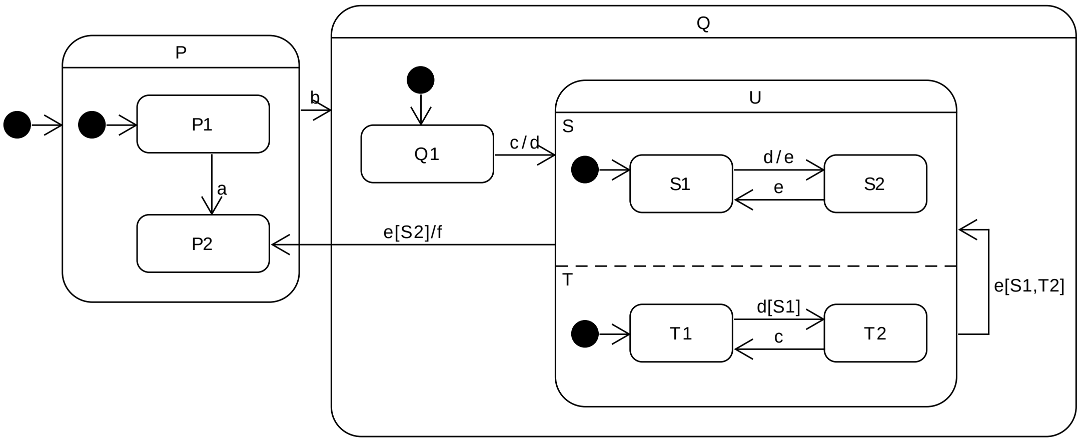


Figure 13.3: Another state machine diagram.

1. Determine the scope of the transitions labelled a, c/d, d/e and e[S2]/f in this state machine diagram.
2. For each of the above transitions, give a situation with a dispatched event in which it is enabled, and with respect to that situation, define the exit and enter set of the transition.

13.2.3 Calculating transitions

The UML standard gives some guidance on how to deal with conflicting transitions. For example, when two transitions $tr_1 = (s_{s,1}, E_{t,1}, C_1, E_{g,1}, s_{t,1})$ and $tr_2 = (s_{s,2}, E_{t,2}, C_2, E_{g,2}, s_{t,2})$ are both enabled, and $s_{s,1} \in sdesc(s_{s,2})$, then tr_1 gets priority, and tr_2 is not executed. Otherwise, there is no priority relation between the two.

We call all transitions that are not overridden by a transition with higher priority *fireable*.

Definition 13.2.8 (Fireable transitions)**.** Transition $tr = (s_s, E_t, C, E_g, s_t)$ is *fireable* at situation $st = (conf, Ev)$ with dispatched event e if and only if $tr \in enabled(st, e)$ and for all $tr' = (s'_s, E'_t, C', E'_g, s'_t)$ such that $tr' \in enabled(st, e)$, we have $s'_s \notin sdesc(s_s)$.

We refer to the set of fireable transitions at situation st with dispatched event e as $fireable(st, e)$.

Even though selecting only the fireable transitions resolves part of the conflicts, there can still be conflicts in the set of fireable transitions. An actual transition in the semantics consists of executing a maximal conflict-free subset of fireable transitions.

Definition 13.2.9 (Maximal micro-step)**.** Given situation $st = (conf, Ev)$ with dispatched event e, a maximal micro-step $MMS(st, e) \subseteq fireable(st)$ is a set such that:

- for all transitions $tr_1, tr_2 \in MMS(st, e)$, if $tr_1 \neq tr_2$, then tr_1 and tr_2 do not conflict, and
- for all transitions $tr_1 \in fireable(st, e) \setminus MMS(st, e)$ there exists a transition $tr_2 \in MMS(st, e)$ such that tr_1 and tr_2 conflict.

Example 13.2.6. Consider $(\{\mathsf{S2}, \mathsf{S3}, \mathsf{S4}\}, \{\mathsf{e4}\})$ as the situation of Figure 13.1, with dispatched event e2. Note that the transition from S3 to S3 and the transition from S4 to S5 are enabled. Furthermore, both transitions are non-conflicting, since their exit sets are $\{\mathsf{S3}\}$ and $\{\mathsf{S4}\}$, respectively. This means that both transitions are fireable, and together they form a maximal set of micro-steps.

Exercise 13.3 Consider the UML state machine diagram in Figure 13.3.

For each of the following situations, with dispatched events, calculate the set of all enabled transitions. Give the exit and enter set of each enabled transition. Indicate which transitions are not fireable (if any). Calculate maximal micro-steps for the situations and calculate the configuration that is reached after taking each MMS from its corresponding situation.

1. $(\{\mathsf{P}, \mathsf{P1}\}, \{\mathsf{a}\})$ with dispatched event b.
2. $(\{\mathsf{P}, \mathsf{P1}\}, \{\mathsf{b}\})$ with dispatched event a.
3. $(\{\mathsf{Q}, \mathsf{U}, \mathsf{S}, \mathsf{T}, \mathsf{S1}, \mathsf{T1}\}, \emptyset)$ with dispatched event d
4. $(\{\mathsf{Q}, \mathsf{U}, \mathsf{S}, \mathsf{T}, \mathsf{S2}, \mathsf{T2}\}, \emptyset)$ with dispatched event e

13.2.4 Labelled transition system of a state machine diagram

A maximal micro-step contains all transitions that should be executed as part of handling one event. In the general case, where activities are taken into account, the transitions are executed as part of a more complex *compound transition*, in which the transitions can be executed in any order. Since we restrict the syntax of state machine diagrams and do not support activities in general, and the order in which the transitions are executed is immaterial, we choose to execute all transitions in a maximal micro-step simultaneously.

Now, we have all the ingredients necessary to define an LTS for a state machine diagram. We refer to the state machine diagram as $D = (\mathcal{S}, \mathcal{T})$, and to its LTS as $T_D = (S, Act, \rightarrow, I)$. This LTS can be used to describe the behavior of state diagram D as a reaction to the triggers provided by an environment and by transitions fired by itself.

Definition 13.2.10 (LTS for a state machine diagram)**.** Let $D = (\mathcal{S}, \mathcal{T})$ be a state machine diagram. We construct its LTS $T_D = (S, Act, \rightarrow, I)$ as follows:

- $I = \{s_i\}$, the initial situation, where $s_i = (conf_0, \{\})$, such that $\forall s \in conf_0.initial(s)$.
- $S = S_e \cup S_t \cup S_D$, where S_e, S_t and S_D are pairwise disjoint; S_e is the set of *environment situations* (also containing the set I) in which new events can be added; S_t is the set of *trigger situations*, in which the environment dispatches a particular event; and S_D is the set of *state diagram situations*.
- Act consists of all subsets of the events $\mathcal{E}$ of D, the set of event $\mathcal{E}$, plus all combinations of such subsets E, E' in the form E/E', i.e., $Act = 2^{\mathcal{E}} \cup \mathcal{E} \cup \{E/E' \mid E, E' \in 2^{\mathcal{E}}\}$;

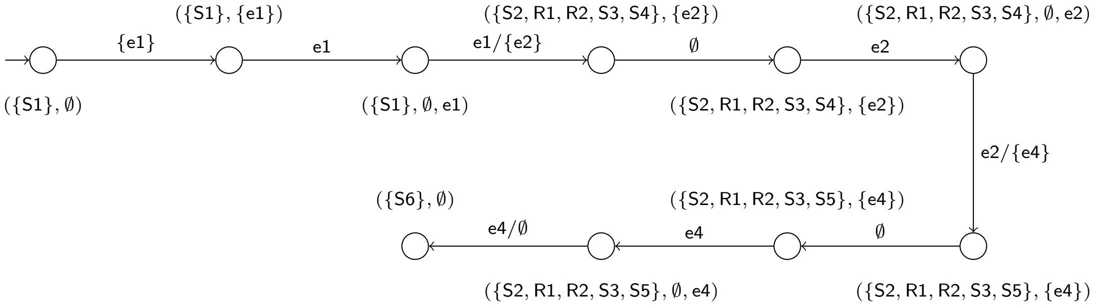

({S1}, ∅)
{e1}
({S1}, {e1})
e1
({S1}, ∅, e1)
e1/{e2}
({S2, R1, R2, S3, S4}, {e2})
∅
({S2, R1, R2, S3, S4}, {e2})
e2
({S2, R1, R2, S3, S4}, ∅, e2)
e2/{e4}
({S2, R1, R2, S3, S5}, {e4})
∅
({S2, R1, R2, S3, S5}, {e4})
e4
({S2, R1, R2, S3, S5}, ∅, e4)
e4/∅
({S6}, ∅)

Figure 13.4: LTS for the state machine diagram in Figure 13.1, where the environment initially delivers event e1, and subsequently does not add any events. State names are shown above/below the states.

- $\rightarrow$ is the set of transitions consisting of three (disjoint) sets of transitions $\rightarrow_e$, $\rightarrow_t$ and $\rightarrow_D$. These sets are defined as follows:
  - $\rightarrow_e \subseteq S_e \times 2^{\mathcal{E}} \times S_t$ is the set of transitions by which the environment adds events; transitions in this set are of the form:
  $$(\mathit{conf}, E) \xrightarrow{E'} (\mathit{conf}, E \cup E')$$
  , where E and E' are subsets of $\mathcal{E}$.
  - $\rightarrow_t \subseteq S_t \times \mathcal{E} \times S_D$ is the set of transitions by which the scheduler dispatches an event. These transitions are of the form:
  $$(\mathit{conf}, E) \xrightarrow{e} (\mathit{conf}, E \setminus \{e\}, e)$$
  , where $e \in E$.
  - $\rightarrow_D \subseteq S_D \times \mathcal{E} \times 2^{\mathcal{E}} \times S_e$ is the set of transitions by which the state diagram reacts. A transition of the form
  $$(\mathit{conf}, E, e) \xrightarrow{e/E_1} (\mathit{conf}', E')$$
  is in $\rightarrow_D$ if and only if either:
    - in situation (conf, E) with dispatched event e, a maximal micro-step, denoted $\mathit{MMS}((\mathit{conf}, E), e)$, exists. It consists of state diagram transitions $\{\mathit{tr}_0, \ldots, \mathit{tr}_n\}$, with the union of activities E_1, $\mathit{conf}' = \mathit{conf} \setminus (\mathit{Ext}(\mathit{tr}_0, \mathit{conf}) \cup \ldots \cup \mathit{Ext}(\mathit{tr}_n, \mathit{conf})) \cup (\mathit{Ent}(\mathit{tr}_0) \cup \ldots \cup \mathit{Ent}(\mathit{tr}_n))$, and $E' = E \cup E_1$, or
    - in situation (conf, E) with dispatched event e, $\mathit{MMS}((\mathit{conf}, E), e) = \emptyset$, and $\mathit{conf} = \mathit{conf}'$, $E = E'$, and $E_1 = \emptyset$, i.e., the dispatched event is consumed and the state is unchanged.

In general, the LTS corresponding to a state machine diagram becomes large due to the uncontrolled nature of the events provided by the environment, and the non-deterministic dispatching of them. In exercises and examples we therefore typically provide a sequence of (sets of) events provided by the environment, and only present the corresponding prefix of the labelled transition system (essentially continuing the LTS until there are no more events to be dispatched).

Example 13.2.7. Recall the state machine diagram from Figure 13.1, and suppose the environment initially provides the event e1, after which it does not provide any more events. The corresponding prefix of the LTS is shown in Figure 13.4.

We have indicated the initial state (the one at the top left) with an incoming arrow-head. Since root is trivially part of every configuration, we have excluded it from the listed configurations.

Exercise 13.4 Consider the UML state machine diagram in Figure 13.3. Suppose the environment provides its triggers in the following steps: {b}, {c}, and subsequently does not provide new triggers. Draw the corresponding part of the labelled transition system for the state machine diagram. Your solution to Exercise 13.3 may be helpful in solving this exercise.

13.3 Conclusion

In this chapter we have shown how the semantics of a subset of state machine diagrams can be formalized using labelled transition systems. Having such formal semantics helps to unambiguously design system behavior and reason about that behavior. The formalization also shows that there are many subtleties in the UML standard, even for the small subset considered in this chapter. When adding additional features such as history variables as well activities, things become even more involved. In particular, we cannot execute all non-conflicting transitions at once, but instead we need to execute them one after another, in an arbitrary order using run-to-completion semantics. Since the UML standard is (sometimes intentionally) vague in many places, there have been many different attempts at formalizing UML state machine diagrams. There are many subtle differences between the different formalizations. The interested reader is referred to, e.g., Lilius and Paltor (1999) for more complete formalizations of state machine diagrams.

Chapter 14

Model-Based Testing with LTSs

In this chapter, we describe a theory for model-based testing. The theory we study is called **ioco**, for input-output conformance. Models, test cases, and implementations are all represented by Labelled Transition Systems. It is formally defined when an implementation is considered "valid", i.e., it conforms to its specification. Furthermore, we give an algorithm to automatically derive and execute test-cases from specifications.

This theory has been described in great detail in Tretmans (2008). The paper is available via Canvas, and you need to be in the TU/e network in order to download its PDF. Since it is an excellent source for the theory, in this chapter we guide you through the paper, instead of giving the full details. We will repeat the most important definitions in this chapter.

We will often ask you to read a specific part of Tretmans (2008) first. In fact, the sections in this chapter follow the sections from Tretmans (2008).

Some of the exercises in this chapter have been taken from the material of the course "System Verification & Testing" at the Open University.[1]

At the end of this chapter, you can

- give an informal description of model-based testing;
- informally describe the main components of the model-based testing framework;
- explain the intuition behind quiescence;
- apply the formal definition of quiescence;
- explain the intuition behind **ioco**;
- give the formal definition of **ioco**;
- apply **ioco** on small examples;
- reason about important properties of **ioco**;
- discuss weaknesses and variations of **ioco**.

14.1 Introduction

Read section 1 of Tretmans (2008)

Model-based testing is a formal testing approach, in which a formal model is used to represent the behavior of the implementation under test (IUT). While treating the IUT as a black-box, i.e., we cannot observe the implementation details, we want to establish the IUT conforms to its specification.

14.2 Formal Testing

Read section 2 of Tretmans (2008)

[1] The exercises for the Open University course were originally created by Jeroen Keiren.

In this course, we are introducing a formal approach to model-based testing. From the specification, test cases are generated, that are executed on the implementation. Formally, an IUT conforms to the specification if and only if it passes all tests that can be generated from the specification.

In order to understand this, we need to ask ourselves what it means for an IUT to conform to the specification. In this case, we consider input-output conformance (**ioco**), in which the specification, the IUT, and the test cases are described as labelled transition systems. In the **ioco** theory, it is formalized what exactly it means for the IUT to conform to its specification. Intuitively, it is that after every (weak) trace allowed by the specification, the outputs produced by the implementation are admitted by the specification.

Input-output conformance testing is a black-box approach to testing, which means we do not assume any knowledge about the IUT, except for its interface. However, in order to use this approach we need to make the following assumption: the IUT can be described as a labelled transition system (even though we do not need explicit access to this LTS). This assumption is also called the *test assumption* g.

14.3 Models

As stated before, in model-based testing the specification, the implementation and the test cases are described using labelled transitions systems. Note that the definition given in Tretmans (2008) differs slightly from the definition given in Chapter 12.

In particular, Tretmans (2008) uses Q for the set of states (we used S before), the action labels are described using L (we used *Act* before;), and its initial state is denoted q_0. Be aware of these differences when studying the paper.

Now go ahead and read section 3 of Tretmans (2008). You can skip section 3.2, and for now you can skip section 3.5, to which we will return later.

14.3.1 Labelled Transition Systems

In general, the notation introduced in this section corresponds to the notation we introduced in Chapter 12. For example, the notation extending the transition relation to traces and weak traces is the same. Be careful in Definition 5.2, the traces that Tretmans (2008) defined here are, in fact, our *weak traces*, which we can see from the use of the double arrow notation ($\Longrightarrow$). Also, the paper uses *der* to denote the set of reachable states, where we previously introduced *Reach*.

The important new part of (Tretmans, 2008, Definition 5) is Definition 5.5.

Definition 14.3.1 ((Tretmans, 2008, Definition 5.5))**.** Let (Q, L, T, q_0) be an LTS. Let $P \subseteq Q$, and $A \subseteq L$.

$$P \textbf{ refuses } A = \exists p \in P . \forall \mu \in A \cup \{\tau\} . p \overset{\mu}{\not\rightarrow}$$

In other words, P refuses A, if there is a state p in P that cannot do any of the actions in A, nor can it take an internal transition.

Exercise 14.1 Consider the LTSs in Figure 2 on page 8 of Tretmans (2008). Give the following sets:

1. $\mathit{traces}(q)$
2. r_0 **after** but
3. r_0 **after** $\mathit{but} \cdot \mathit{but}$
4. v_0 **after** but
5. v_0 **after** $\mathit{but} \cdot \mathit{but}$

Exercise 14.2 Consider the LTSs in Figure 2 on page 8 of Tretmans (2008). For each of the following predicates, state whether it is true or false.

1. r_0 **after** but **refuses** $\{liq\}$
2. r_0 **after** $but \cdot but$ **refuses** $\{choc\}$
3. r_0 **after** but **refuses** $\{liq, but\}$
4. v_0 **after** but **refuses** $\{liq\}$

14.3.2 LTSs with inputs and outputs

Testing involves a *tester* sending inputs to an *IUT* and then the tester observes the outputs produced by the IUT and decides a verdict, *pass* or *fail.* We need to extend LTSs with the notions of inputs and outputs. Inputs are actions initiated by the environment of a system; outputs are initiated by the system itself. In other words, inputs are controlled by the environment; the system reacts to them. Outputs are controlled by the system; the environment reacts to them.

We recall the definition of *labelled transition system with inputs and outputs* (Tretmans, 2008, Definition 6) as an LTS where the set of action labels is partitioned into a set of inputs – notation L_I – and a set of outputs – notation L_U.[2]

Definition 14.3.2. An LTS with inputs and outputs is a 5-tuple (Q, L_I, L_U, T, q_0) where:

- $(Q, L_I \cup L_U, T, q_0)$ is an LTS;
- L_I and L_U are countable sets of input labels and output labels, respectively, which are disjoint: $L_I \cap L_U = \emptyset$.

We sometimes write L for $L_I \cup L_U$.

We can reinterpret the LTSs from (Tretmans, 2008, Figure 2) 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 14.1.

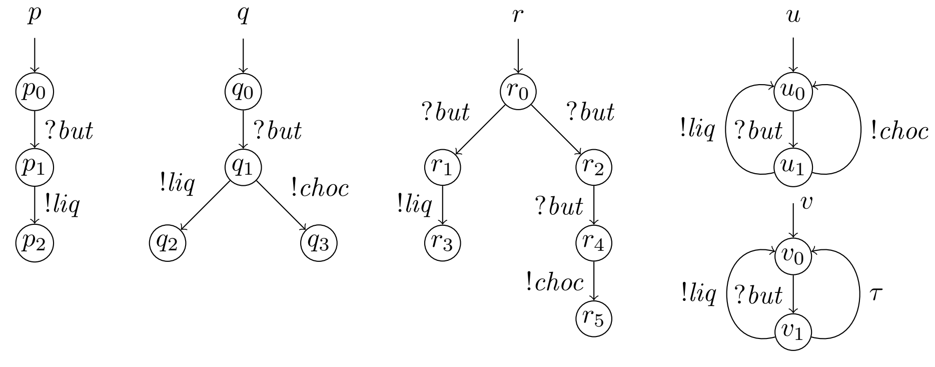


Figure 14.1: LTSs from (Tretmans, 2008, Figure 2) reinterpreted as LTSs with inputs and outputs using $L_I = \{but\}$ and $L_U = \{liq, choc\}$

14.3.3 Input-Output Transition Systems

The previous definition does not differ much from the definition of LTSs. In the theory, we need to identify systems that never refuse inputs. In the **ioco** theory, we will assume that implementations can be represented by such systems. The conceptual idea is that an IUT is always prepared to accept any input, that is, all inputs are enabled in all states. Essentially, this assumption requires the implementation to explicitly specify how to handle invalid inputs, which is a good defensive programming strategy by itself.

[2] The 'U' comes from "uitvoer", the Dutch word for output. It is preferred to avoid confusion between L_O (letter 'O') and L_0 (digit '0').

Definition 14.3.3 ((Tretmans, 2008, Definition 7))**.** An input-output transition system (IOTS) is a labelled transition system with inputs and outputs (Q, L_I, L_U, T, q_0) where all input actions are enabled in all reachable states:

$$\forall q \in der(q_0).\forall a \in L_I : q \xRightarrow{a}$$

There are many ways to make systems input enabled. A solution is to add to each state a self-loop with the input labels that are not accepted in that state. This is sometimes referred to as *Angelic Completion.* Another way is to add a special error state and add transitions to this error state for all non-specified inputs. This is sometimes referred to as *Demonic Completion.*

Exercise 14.3 Consider the LTSs with inputs and outputs from Figure 14.1. Which of these LTSs is/are IOTSs?

14.3.4 Quiescence and suspension traces

In order to define the *quiescence* relation, we need to extend the kinds of observations we can do in a labelled transition system. In particular, we need to be able to observe whether an LTS can proceed autonomously, without accepting an input from the environment. Essentially, this is the case when the system can do an output in the current state.

A state in which the system is completely idle and will not make any more progress without receiving an input first is called *quiescent.* A quiescent state q is denoted as $\delta(q)$, which is defined as follows:

Definition 14.3.4 ((Tretmans, 2008, Definition 8))**.** A state q of an LTS p is *quiescent*, denoted by $\delta(q)$ if and only if $\forall \mu \in L_U \cup \{\tau\} : q \overset{\mu}{\not\rightarrow}$.

It is important to notice that quiescence includes the impossibility to perform internal actions. In practice, this means that such a stable state of the IUT is observable.

Exercise 14.4 Consider the LTSs with inputs and outputs from Figure 14.1. Which states in these LTSs are quiescent?

Since we assume that quiescence is observable, we extend LTSs with the action δ (L_δ), for quiescence, and add transitions showing quiescence in the LTS. We call such an extended LTS a *suspension automaton.* The suspension traces (*STraces*) are then defined as the traces of the suspension automaton.

Definition 14.3.5 ((Tretmans, 2008, Definition 9))**.** Let $p = (Q, L_I, L_U, T, q_0)$ be an LTS with inputs and outputs. Recall that $L = L_I \cup L_U$. We write $L_\delta = L \cup \{\delta\}$.

1. The suspension automaton for p is $p_\delta = (Q, L_I, L_U \cup \{\delta\}, T \cup T_\delta, q_0)$, where $T_\delta = \{q \xrightarrow{\delta} q \mid q \in Q \wedge \delta(q)\}$
2. The *suspension traces of* p are $STraces(p) = \{\sigma \in L_\delta^* \mid p_\delta \xRightarrow{\sigma}\}$

Note that in this definition, the suspension traces of p are defined to be the *traces* of p_δ. Note that p_δ denotes the suspension automaton, L_δ is the set of action labels extended with δ, and T_δ is the extended transition relation.

Exercise 14.5 Consider the LTS with inputs and outputs r from Figure 14.1. Give the set of suspension traces $STraces(r)$.

14.4 Formal conformance relation

Conformance defines when an *IUT* meets a *specification.* It is therefore a relation between two entities: the implementation and the specification. In **ioco**, conformance is expressed as a *subset* relation between the outputs possibly produced by the implementation and the outputs possibly

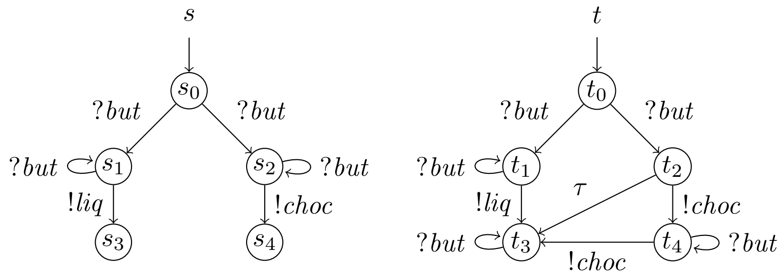


Figure 14.2: LTS with inputs and outputs s (left), and IOTS t (right), where $L_I = \{?but\}$, and $L_U = \{!liq, !choc\}$

produced by the specification. We check this relation in all states reached after all traces in a *set of traces*. To express the conformance relation we need to define the following:

1. the set of suspension traces, that is, the observable behavior;
2. the set of states reached after performing a trace;
3. the set of outputs, that is, the visible behaviors on which conformance is checked;
4. the relation between two sets of outputs.

Read the introduction to section 4, and read section 4.1 in Tretmans (2008).

14.4.1 Output set

The first element of the formal **ioco** relation is the *out*-set. This is the union of all output actions together with the quiescence action. The second part of the definition generalizes the *out*-set to sets of states. In the way in which the *out*-set is used in the **ioco** definition, this second part is to deal with non-determinism.

Definition 14.4.1. Let q be a state in a transition system. Let Q be a set of states. We define:

1. $out(q) = \{x \in L_U \mid q \xrightarrow{x}\} \cup \{\delta \mid \delta(q)\}$
2. $out(Q) = \bigcup\{out(q) \mid q \in Q\}$

Exercise 14.6 Consider the LTSs in Figure 14.2. Determine the following *out*-sets.

1. $out(s_0)$
2. $out(s_1)$
3. $out(s_2)$
4. $out(s_3)$
5. $out(\{s_0, s_1\})$
6. $out(\{s_1, s_2\})$
7. $out(\{s_1, s_2, s_3\})$
8. $out(\{t_0\}$ **after** $?but)$
9. $out(\{t_0\}$ **after** $?but \cdot ?but)$
10. $out(\{t_0\}$ **after** $?but \cdot \delta)$
11. $out(\{t_0\}$ **after** $?but \cdot !choc)$

14.4.2 The ioco conformance relation

The intuition behind **ioco** is to check that any output produced by the implementation has been foreseen by the specification. This intuition is formally expressed by a subset relation. The set of outputs produced by the implementation i must be a subset of the set of outputs produced by the specification s. This must hold for all states reachable over all possible suspension traces of the *specification.* Formally, we have the following definition:

Definition 14.4.2 ((Tretmans, 2008, Definition 12))**.** Given a set of inputs L_I and a set of outputs L_U, the relation **ioco** is defined as follows:

$$i \ \mathbf{ioco} \ s \Leftrightarrow \ \forall \sigma \in \mathit{STraces}(s): \ \mathit{out}(\ i \ \mathbf{after} \ \sigma) \ \subseteq \ \mathit{out}(\ s \ \mathbf{after} \ \sigma)$$

Exercise 14.7 Consider the LTSs from Figure 14.2, and use s as specification and t as implementation. Does the implementation conform to the specification, according to the **ioco** definition? Motivate your answer.

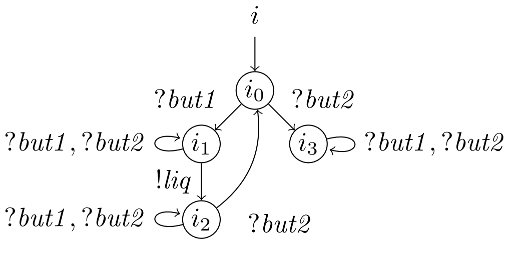


Figure 14.3: LTSs of implementation i and specification s. Input alphabet $L_I = \{?\mathit{but1}, ?\mathit{but2}\}$. Output alphabet $L_U = \{!\mathit{liq}\}$

Exercise 14.8 Figure 14.3 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.

14.4.3 Generalization

In general, the specification used in **ioco**-testing is a *partial specification.* This is due to two reasons: first, the **ioco**-definition only requires the out-sets of the implementation to be *included* in the out-sets of the specification; second, the definition only considers the suspension traces from the specification.

By changing the sets of traces considered by the specification, we can obtain variations of **ioco**. In order to do this, we can make the set of traces the definition considers a parameter, say $\mathcal{F}$. We will use this parameter in the next section about underspecified traces.

Definition 14.4.3 ((Tretmans, 2008, Definition 13))**.** Let $\mathcal{F} \subseteq (L_I \cup L_U \cup \{\delta\})^*$ be a set of suspension traces, i an IOTS and s an LTS with inputs and outputs.

$$i \ \mathbf{ioco}_{\mathcal{F}} \ s \equiv \ \forall \sigma \in \mathcal{F}.\mathit{out}(i \ \mathbf{after} \ \sigma) \subseteq \mathit{out}(s \ \mathbf{after} \ \sigma)$$

Note that when choosing $\mathcal{F} = \mathit{STraces}(s)$ in this definition, we recover the original definition of **ioco**. There are several other options for choosing $\mathcal{F}$, that all result in variations of **ioco**.

14.4.4 Underspecified traces and uioco

A particular form of underspecification arises when we consider suspension traces in non-deterministic LTSs. This is caused by the fact that subtraces may end up in states in which their continuation is not enabled. The paper Tretmans (2008) illustrates this using an example.

To overcome this, we can consider the set of *UTraces*, which is the set of suspension traces not allowing for this kind of underspecification.

Definition 14.4.4 ((Tretmans, 2008, Definition 15))**.** Let i be an Input-Output Transition System over L_I and L_U and let s be an LTS with inputs and outputs over L_I and L_U.

1. $\mathit{UTraces}(s) =_{def} \{\sigma \in \mathit{STraces}(s) \mid \forall \sigma_1, \sigma_2, a \in L_I.$
   $\sigma = \sigma_1 \cdot a \cdot \sigma_2$ implies not s **after** σ_1 **refuses** $\{a\}\}$
2. i **uioco** $s \equiv i$ $\mathbf{ioco}_{\mathit{UTraces}(s)}$ s

Note that in this definition, $\sigma = \sigma_1 \cdot a \cdot \sigma_2$ splits up the suspension trace, and σ is only in the set *UTraces*, if it is not the case that s **after** σ_1 **refuses** $\{a\}$. Now, remember that s **after** σ_1 is the set of states reached after executing σ_1. Furthermore, **refuses** is defined such that this set of states we reach refuses $\{a\}$ if and only if there is a state in this set that cannot do an a transition. So, if a trace is in $\mathit{UTraces}(s)$, for every possible partial execution of that trace starting in s, it is possible to continue executing the trace.

14.4.5 Compositional testing

Ideally, when developing a large system, we would like to be able to test the components in isolation, and from that infer that the system as a whole still satisfies the specification. Unfortunately, for **ioco**, this is not the case: if i_1 **ioco** s_1 and i_2 **ioco** s_2, this does not mean that the implementation in which i_1 and i_2 communicate **ioco** conforms to the specification in which s_1 and s_2 communicate. In Tretmans (2008) an example of this is shown. Note that, when the specification is input-enabled (i.e. it is a IOTS instead of an LTS with inputs and outputs), compositional testing can safely be applied.

14.5 Test cases and test generation

So far, we have defined the **ioco** relation on pairs of LTSs. Recall, however, that we want to use **ioco** as a black-box testing approach, so we do not have the details of the implementation. We will, therefore, use the specification to generate test cases, and describe how to execute those test cases on the implementation.

14.5.1 Test cases

Read section 3.5 in Tretmans (2008).

Test cases are described using test transition systems (TTSs), which are LTSs with a very specific structure. In particular, a test case is always finite-state and deterministic, and it has a tree-structure that ends in leaves that are either **pass** or **fail**. Furthermore, it is important to observe that, when the specification from which the test case is generated has inputs L_I, and outputs L_U, the test case will swap these, and use L_U for inputs, and L_I for outputs (in the presentation of the test cases, we will write $?o$ for $o \in L_U$, and $!i$ for $i \in L_I$). The intuition is that the test case will actually *provide* inputs to the IUT, and *observe* outputs from the IUT.

Every state q in the test case has the following possibilities:

- either, it provides exactly input $a \in L_I$ to the IUT, and it specifies how to deal with all outputs in L_U that the IUT could produce, or
- it specifies for all outputs L_U that the IUT could produce how the test proceeds, and it observes quiescence, and checks whether that is allowed.

For a formal definition of test cases, check (Tretmans, 2008, Definition 10). Note that the special action θ is used to detect quiescence.

Exercise 14.9 Assume that we have a specification with $L_I = \{?but\}$ and $L_U = \{!liq, !choc\}$. Figure 14.4 shows four LTSs with inputs and outputs.

1. For each of the LTSs in Figure 14.4 indicate whether it is a TTS. In other words, does it satisfy the requirements of test cases. If not, argue your answer.

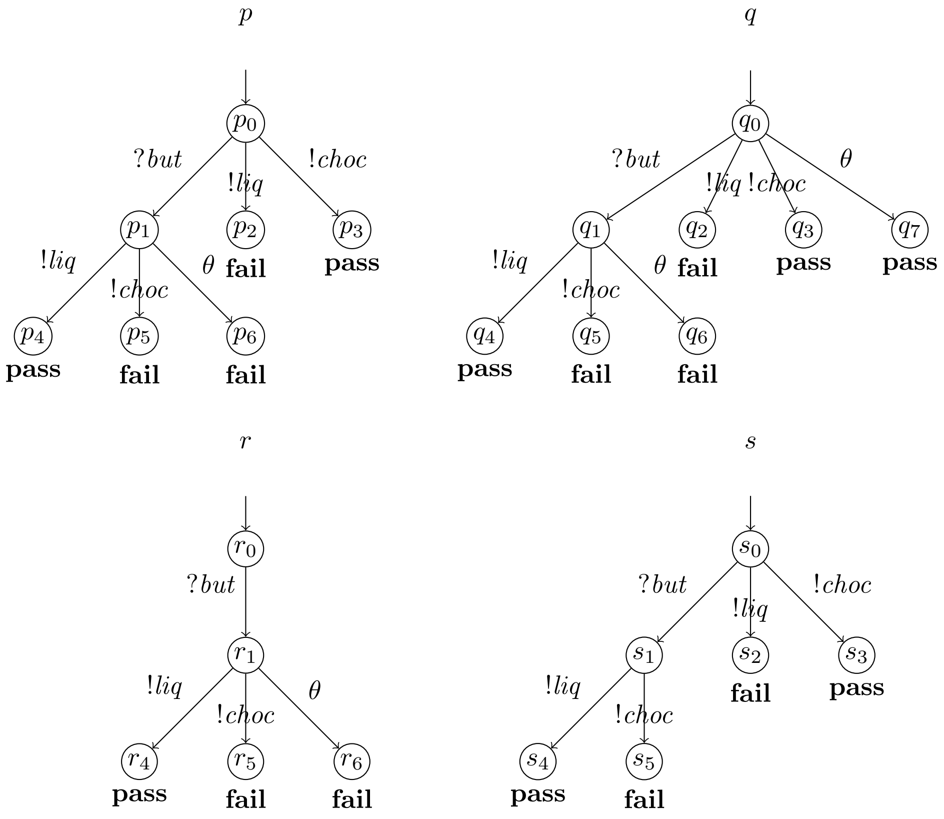


Figure 14.4: Four LTSs with inputs and outputs with $L_I = \{?but\}$, $L_U = \{!choc, !liq\}$

2. Suppose that we remove the transition $p_0 \xrightarrow{!choc} p_3$ from the LTS for p in the figure. Is the result a TTS? Why (not)?
3. Suppose that we remove the transition $p_1 \xrightarrow{!liq} p_4$ from the LTS for p in the figure. Is the result a TTS? Why (not)?

14.5.2 Test execution

Read the introduction to section 5, and read section 5.1 in Tretmans (2008).

Tests are executed by running the test case and the IUT in parallel. This is described using a variation of the parallel composition operator: $t\rceil\!|i$ denotes the test case t and implementation i running in parallel. Note that the parallel composition is similar to the parallel composition that we have discussed in Chapter 12. In this case, synchronization happens on all labels in $L = L_I \cup L_U$, and the labels δ and θ also synchronize.

Test execution is formally defined in (Tretmans, 2008, Definition 16). Note that it uses three inference rules for defining the test execution. It uses the more abstract notation $t\rceil|i$ for the parallel composition of test t and implementation i, assuming that t and i are described more abstractly as processes instead of as LTSs.

The first rule defines how internal transitions of the implementation are executed.

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

The implementation simply executes the internal transition, and updates its state locally. Note that specifications do not have internal transitions, so a similar rule for specifications is not needed.

The second rule defines the synchronization of the test case and the IUT on an action a. Note that action a can either be input or output.

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

This simply says that when both components can perform a, the composition can also do a.

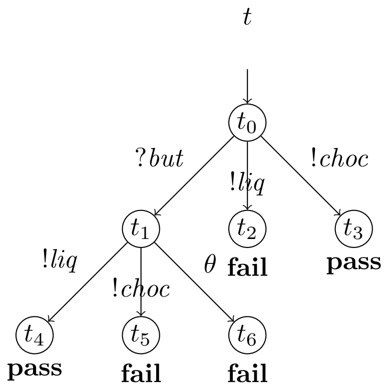
The third rule specifies the synchronization between test case and IUT in case quiescence is observed.

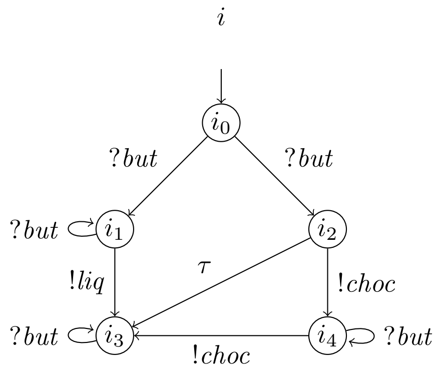
$$\frac{t \xrightarrow{\theta} t', i \xrightarrow{\delta} i'}{t\rceil\!|i \xrightarrow{\theta} t'\rceil\!|i'}$$

It is essentially the same as the previous one, but we deal with the special status of the actions δ and θ.

A test run of t with i is a trace of $t\rceil\!|i$ terminating in a verdict **pass** or **fail** of t (see (Tretmans, 2008, Definition 16.2)). Implementation i passes test case t if all test runs go to a **pass**-state of t (see (Tretmans, 2008, Definition 16.3)). An implementation i passes a test suite T if it passes all test cases in T (see (Tretmans, 2008, Definition 16.4)).

Exercise 14.10 Consider the following test case t, and implementation i.





Give all possible test runs of the test case with the implementation.

14.5.3 Test generation

Read section 5.2 in Tretmans (2008).

The test case generation algorithm described in (Tretmans, 2008, Algorithm 1) generates test cases as follows. It considers a leaf in the test case completed if it is either **pass** or **fail**. If a leaf is not completed, the test case is extended at the leaf in one of three ways:

1. either it stops with a verdict **pass**, since no error has been found so far, or
2. it sends an input to the IUT and listens for outputs from the IUT (note that quiescence is not checked in this case); outputs that are invalid lead to a **fail** verdict, other outputs lead to uncompleted leaves, or
3. it listens for outputs from the IUT and detects quiescence; again invalid outputs (and quiescence if the state in the specification is not quiescent) lead to a **fail** verdict, other outputs lead to uncompleted leaves.

Exercise 14.11 Consider the LTS with inputs and outputs from Figure 14.5. Use the **ioco** test-generation algorithm to derive a test case that contains at least !, ?, θ, **pass**, and **fail**.

Exercise 14.12 Consider the LTS with inputs and outputs from Figure 14.6. Use the **ioco** test-generation algorithm to derive a test case that contains at least !, ?, θ, **pass**, and **fail**.

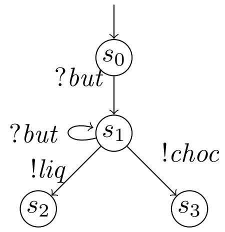


Figure 14.5: LTS with inputs and outputs where $L_I = \{?but\}$, and $L_U = \{!liq, !choc\}$

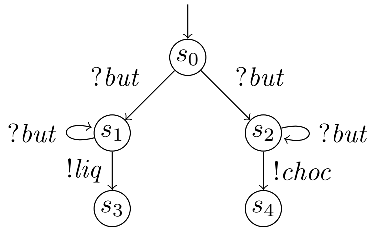


Figure 14.6: LTS with inputs and outputs where $L_I = \{?but\}$, and $L_U = \{!liq, !choc\}$

14.5.4 Completeness of test generation

Read section 5.3 in Tretmans (2008).

Ideally, we want a test suite to be complete. This means that an IUT conforms to its specification if and only if it passes all tests. We break down completeness into two different criteria, soundness and exhaustiveness, such that the test case is complete whenever it is both sound and exhaustive.

A test suite T is sound if and only if, whenever the implementation conforms to the specification, the implementation passes all tests in T. Intuitively, this means that a sound test suite detects *real* errors. We can see this more clearly by looking at the contraposition of this statement: if the implementation fails at least one test, then the implementation does not conform to the specification.

A test suite T is exhaustive if and only if, whenever the implementation passes all tests in T, then the implementation conforms to the specification. It is easier to read this in contrapositive form: if an IUT does not conform to the specification, then there is a test case in the test suite that detects an error. An exhaustive test suite can, therefore, detect all errors in an IUT. A formal definition of soundness, exhaustiveness and completeness is given in (Tretmans, 2008, Definition 17).

An important property of the test generation algorithm described in Tretmans (2008) is that it is complete. However, in general, for the test suite to be exhaustive, infinitely many test cases are needed, so the algorithm will not always terminate. On the other hand, any subset of test cases that can be constructed by applying the test generation algorithm is sound, and can thus be used to generate a useable test suite.

14.6 Exercises

We conclude this chapter with some more complicated exercises.

Exercise 14.13 Consider the labelled transition system in Figure 14.7 specifying a coffee machine. The label ?*button* is an input; !*coffee*, !*espresso* and !*cappuccino* are outputs.

In this exercise, we are investigating five implementations. The implementations are shown in Figure 14.8.

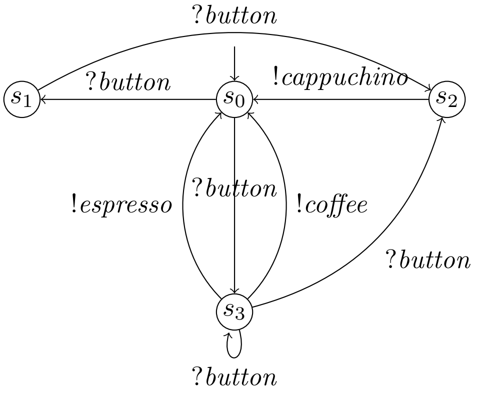
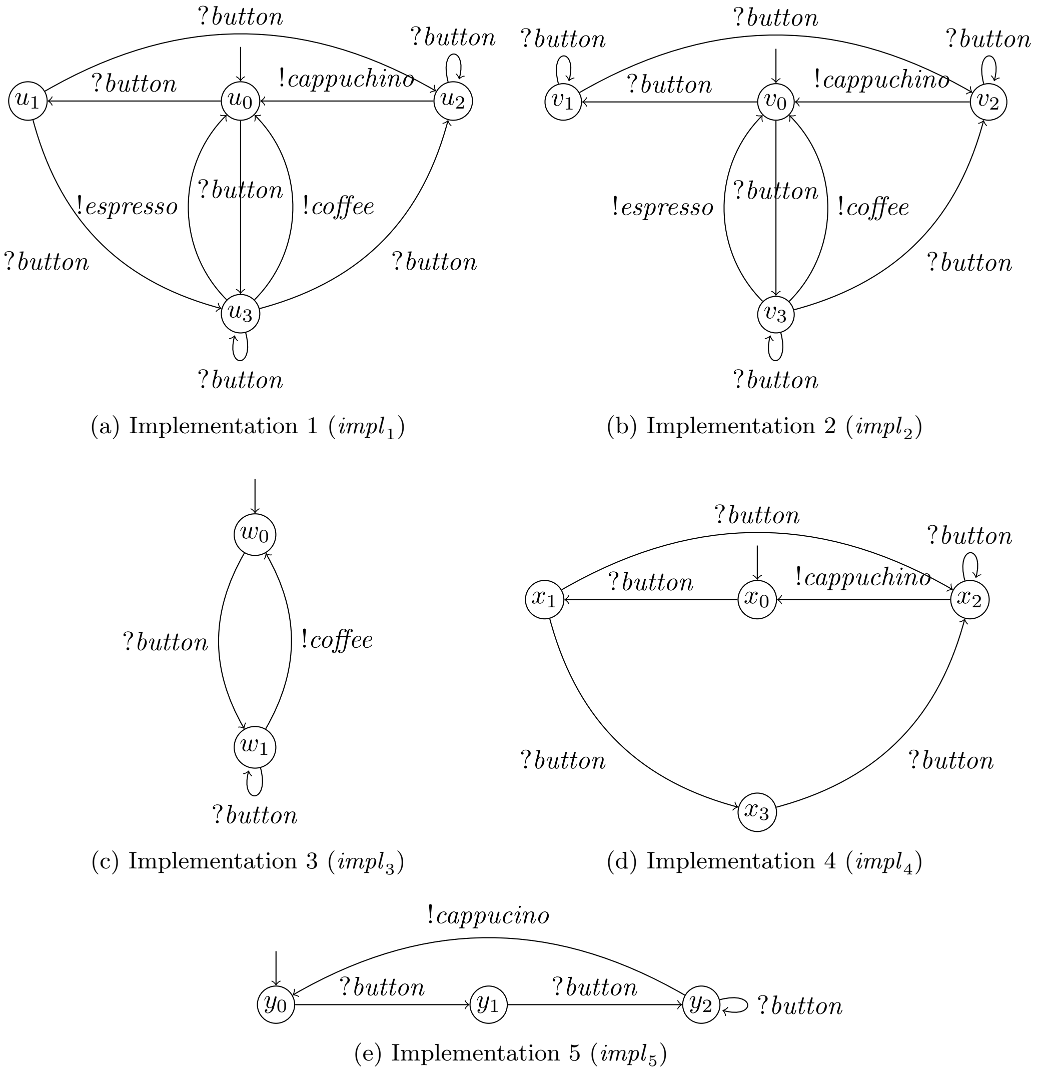


Figure 14.7: Specification (*spec*) of a simple coffee machine



(a) Implementation 1 ($impl_1$)

(b) Implementation 2 ($impl_2$)

(c) Implementation 3 ($impl_3$)

(d) Implementation 4 ($impl_4$)

(e) Implementation 5 ($impl_5$)

Figure 14.8: Five implementations of a coffee machine

1. For each of these five implementations, determine whether it is **ioco**-conforming w.r.t the specification *spec*. Motivate your answer.
2. Make your own *correct* implementation that cannot give *cappuccino*. That is, make a new implementation that is **ioco**-correct with respect to the specification in Figure 14.7 but that cannot give *cappuccino*.
3. Make your own **incorrect** implementation that can produce *coffee* and *espresso*. That is, make a new implementation that is **ioco**-incorrect with respect to the specification in *Figure 14.7* but that can produce *coffee* and *espresso*.
4. Derive a test suite (i.e., a set of test cases), using the **ioco**-test generation algorithm, that can detect all the errors in the **ioco**-incorrect implementations from the first part of this exercise. (Hint: you need to specify at most one test case generated by the algorithm for each of the **ioco**-incorrect implementations. Some test cases may cover more than one implementation.)
5. Explain in at most 5 short sentences what it means that your test cases/test suites are *sound* w.r.t. to the **ioco**-relation.
6. Are your test cases/test suites *sound* w.r.t. to the **ioco**-relation? Why?
7. Explain in at most 5 short sentences what it means that your test cases/test suites are *exhaustive* w.r.t. to the **ioco**-relation?
8. Are your test cases/test suites *exhaustive* w.r.t. to the **ioco**-relation? Why?

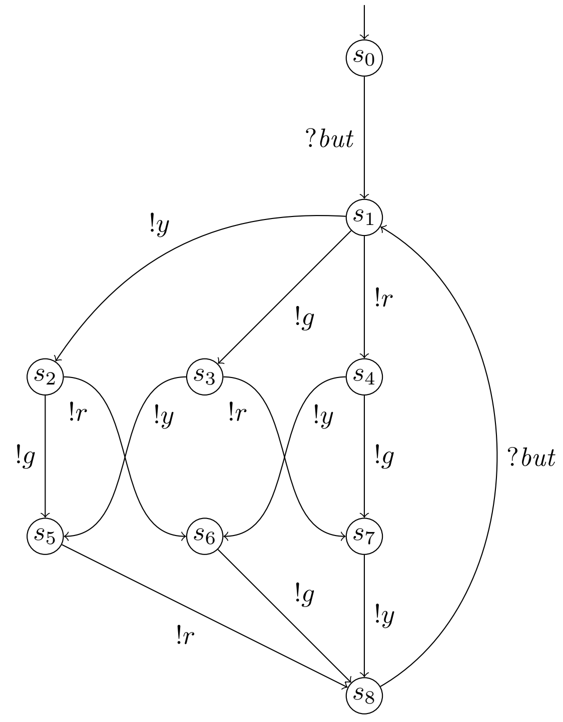


Figure 14.9: Specification of a light system in which a button can be pushed (?*but*), and that can flash red, yellow and green (!*r*, !*y*, !*g*).

Exercise 14.14 Consider a system with three colored lights (*red*, *yellow*, and *green*, denoted *r*, *y* and *g* in the LTS) which can flash after a *button* has been pushed (denoted *but*). Each light flashes exactly once after the *button* has been pushed, and after all three lights have flashed the *button* can be pushed again to repeat the procedure. The order in which the lights flash is not

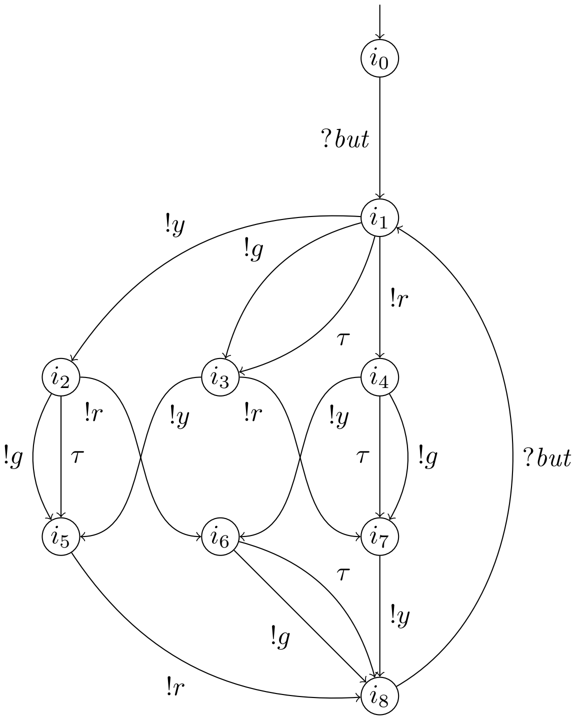


Figure 14.10: Implementation of a light system

important. A specification of this system is given in Figure 14.9, which is a visualization of the labelled transition system.

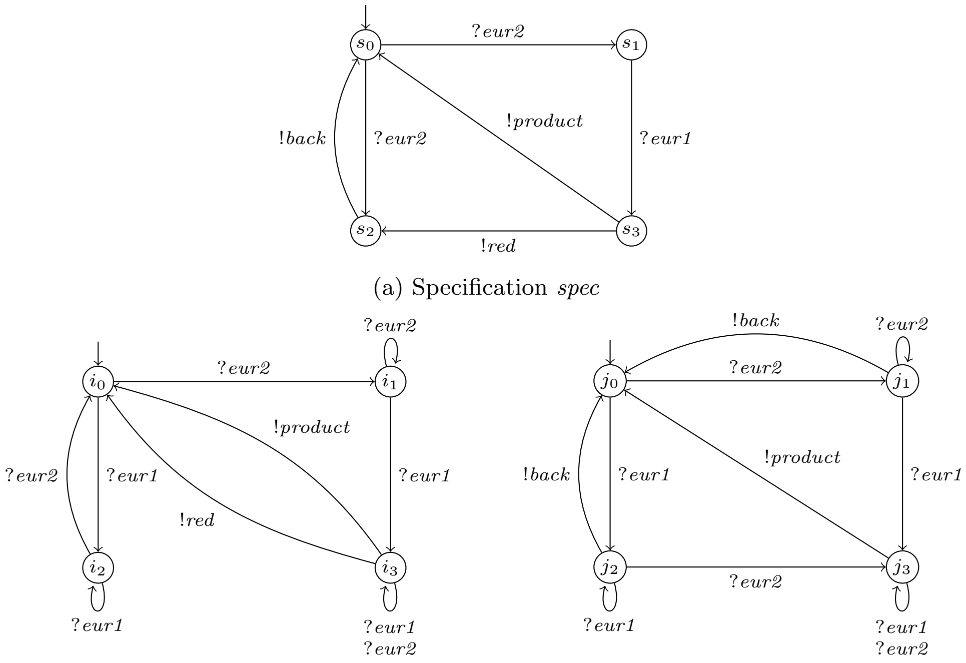
1. Consider the implementation model in Figure 14.10. Is the implementation **ioco**conforming to the specification in Figure 14.9? If yes, explain why. If not, what is wrong?

2. Give an LTS describing an implementation in which the lights always and only flash in the order *red*, *yellow*, *green*.

3. Is the LTS you gave in part 2 of this exercise **ioco**-conforming to the specification in Figure 14.9? Explain your answer.

4. Finally, give an implementation in which the button can be pushed again immediately after any of the flashes, aborting the series of three flashes and starting a new series of three flashes. Is this an **ioco**-conforming implementation of the system given in Figure 14.9? Motivate your answer.

Exercise 14.15 Consider the vending machines that are given in Figure 14.11 as labelled transition systems, with $L_I = \{EUR1, EUR2\}$ and $L_U = \{product, red, back\}$. The vending machine accepts €1 and €2 coins and returns a product (action label *product*). The products can be sold-out; in that case a red light turns on (action label *red*), and the money is returned (action label *back*). In Figure 14.11 the specification of a vending machine *spec* is given, together with two implementations i_1 and i_2.



(a) Specification *spec*

(b) Implementation *i*

(c) Implementation *j*

Figure 14.11: Specification and two implementations of a vending machine

1. Which states of *spec* are *quiescent*?
2. Which of the implementations *i* and *j* are **ioco**-conforming to specification *spec*?
3. Of course products should only be delivered after payment. Make a test case using the **ioco**-test generation algorithm, that tests whether an implementation in the starting state cannot deliver products for free.
4. Make a test case using the **ioco**-test generation algorithm, that tests whether an implementation after inserting in total €3, indeed can deliver a product.
5. Give an **ioco**-conforming implementation that never delivers a product.
6. Give an implementation that is not **ioco**-conforming that can do the following: deliver a product, turn red, return the money.

14.7 Conclusion

Read section 6 in Tretmans (2008).

In this chapter we described a formal approach to testing based on labelled transition systems and the conformance relation **ioco**. Implementations are described as input-output transition systems, specifications are described as labelled transition systems with inputs and outputs, and test cases are described using test transition systems. We formally described the notion of **ioco**-conformance. In essence, the possible outputs of the implementation are included in the possible outputs of the specification, after executing any suspension trace from the specification. The test generation algorithm described is complete.

In the next chapter we illustrate how **ioco**-based conformance testing can be extended to the setting of timed systems.

Bibliography

ISO/IEC 9126-1:2001 Software Engineering — Product Quality — Part 1: Quality Model. ISO/IEC, 2001. URL `https://www.iso.org/standard/22749.html`.

ISO/IEC 25010:2011 Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models. ISO/IEC, 2011. URL `https://www.iso.org/standard/35733.html`.

ISO/IEC/IEEE 29148:2018 Systems and Software Engineering - Life Cycle Processes - Requirements Engineering. ISO/IEC/IEEE, 2018. doi: 10.1109/IEEESTD.2018.8559686.

R. E. Al-Qutaish. Quality Models in Software Engineering Literature: An Analytical and Comparative Study. *Journal of American Science*, 6(3):166–175, 2010. URL `https://pdfs.semanticscholar.org/77b5/1002b53d002278997c71c72eaf2300a87ec7.pdf`.

C. Baier and J.-P. Katoen. *Principles of Model Checking.* The MIT Press, 2008. ISBN 978-0-262-02649-9.

M. Ben-Ari. *Principles of the Spin Model Checker.* Springer London, London, 2008. ISBN 978-1-84628-769-5 978-1-84628-770-1. doi: 10.1007/978-1-84628-770-1.

B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod, and M. J. Merritt. Characteristics of Software Quality. TRW-SS-73-09, 1973.

M. T. Goodrich, R. Tamassia, and M. H. Goldwasser. *Data Structures and Algorithms in Java.* Wiley, sixth edition, international student version edition, 2015. ISBN 978-1-118-80857-3.

G. J. Holzmann. Mars code. *Communications of the ACM*, 57(2):64–73, Feb. 2014. doi: 10.1145/2560217.2560218.

J. Lilius and I. P. Paltor. Formalising UML State Machines for Model Checking. In R. France and B. Rumpe, editors, *≪UML≫'99 — The Unified Modeling Language*, Lecture Notes in Computer Science, pages 430–444, Berlin, Heidelberg, 1999. Springer. ISBN 978-3-540-46852-3. doi: 10.1007/3-540-46852-8_31.

S. Mauw, M. A. Reniers, and T. A. C. Willemse. Message sequence charts in the software engineering process. In *Handbook of Software Engineering and Knowledge Engineering*, pages 437–463. World Scientific Publishing Company, Dec. 2001. ISBN 978-981-02-4973-1. doi: 10.1142/9789812389718_0018.

J. A. McCall, P. K. Richards, and G. F. Walters. Factors in software quality. Volume I. Concepts and definitions of software quality. Technical Report RADC-TR-77-369, General Electric Company, 1977. URL `https://apps.dtic.mil/docs/citations/ADA049014`.

Object Management Group. OMG Unified Modelling Language (UML), version 2.4.1. Technical Report, 2011. URL `https://www.omg.org/spec/UML/2.4.1`.

Object Management Group. OMG Unified Modelling Language (UML), version 2.5.1. Technical Report, 2017. URL `https://www.omg.org/spec/UML/2.5.1`.

G. L. Peterson. Myths about the mutual exclusion problem. 12(3):115–116. doi: 10.1016/0020-0190(81)90106-X.

N. B. Ruparelia. Software Development Lifecycle Models. *SIGSOFT Softw. Eng. Notes*, 35(3): 8–13, May 2010. doi: 10.1145/1764810.1764814.

M. Seidl, M. Scholz, C. Huemer, and G. Kappel. *UML @ Classroom: An Introduction to Object-Oriented Modeling*. Undergraduate Topics in Computer Science. Springer International Publishing, 2015. ISBN 978-3-319-12741-5. URL `https://www.springer.com/gp/book/9783319127415`.

J. Tretmans. Model based testing with labelled transition systems. *Formal Methods and Testing*, pages 1–38, 2008. doi: 10.1007/978-3-540-78917-8_1.

Appendix A

Solutions to exercises

Solution to Exercise 3.1

1. According to the syntax we should have for this requirement [**Subject**][**Action**][**Object**][**Constraint of Action**]. So, we could reformulate the requirement as follows:

   > The system [**Subject**] shall process [**Action**] executing jobs [**Object**] at a load of at least 40 simultaneous jobs [**Constraint of Action**].

2. The resource provider is an actor and should be mentioned in the attributes of the requirement not in the requirement itself. A better formulation of the requirement is the following:

   > The system [**Subject**] shall support [**Action**] the visualization of resource-project relations [**Object**].

3. This requirement has the same issue as the previous requirement. A better formulation would be:

   > The system [**Subject**] shall allow [**Action**] system administrator commands [**Object**] on computers running Windows XP, Mac OS X, or Linux [**Constraint**].

   Note that it could be argued whether the version of Mac OS X and Linux must be part of the constraint.

4. In this requirement, it is not clear what ‘it’ refers to. Is it a resource or the system? We should reformulate the requirement in such a way that this becomes clear, and that follows syntax. We could, e.g., write:

   > If a resource that is executing a job disappears [**Condition**], the system [**Subject**] shall requeue [**Action**] the job that was executed by that resource [**Object**].

Solution to Exercise 3.2 For each requirement, the analysis follows.

1. “a standard computer” is vague. What is standard? Do you mean “every computer connected to the CERN network”? Also the requirement is in the passive form.
   The upgraded management system [Subject] shall give access [Action] of the information stored in the database [Object] to all computers connected to the CERN network [Constraint].
   The new formulation is verifiable.
2. The requirement is not about the management system, but about people giving car stickers. A better formulation would be:
   The upgraded management system [Subject] shall create [Action] car stickers [Object] only for users with a valid CERN ID.

However, it is hard to verify, because it seems that the check for a valid ID is made by a person and not the system.

3. "Under normal conditions" is not precise enough. Also, 3-4 seconds is imprecise. A better formulation would be:
   The software [Subject] shall open [Action] in less than 4 seconds [Constraint of Action].
4. There is more than one requirement in this one. Also, they are not system centric. A better formulation would be:

   R1 The system [Subject] shall give access [Action] to a French-English dictionary [Object].
   R2 The system [Subject] shall support [Action] email services [Object].

   The rest of the description is part of the attributes of requirement [R2].
5. Requirements must not use "will". Also, 24/7 is vague. A better formulation would be:
   The system [Subject] shall run [Action] at a reliability level of 99,99% [Constraint of Action].
   Someone should of course define "reliability level" here.

Solution to Exercise 3.3 We go through the rest of the text, and give some (maybe not all) of the requirements for this system, and point out how we obtained them.

Popular titles are bought in multiple copies.

We can add the option for multiple copies:

R4 The system shall support the registration of multiple copies of titles.

Old books, and magazines are removed when they are out of date or on poor condition.

We can also add the "remove" features; we can wonder whether the titles or copies really need to be removed from the system, or whether they just are not available any more. We here opt for the latter.

R5 The system shall support marking titles as out of date.
R6 The system shall support marking copies to be in poor condition.
R7 When a title is marked out of date, the system should never mark available any its copies.
R8 When a copy is marked in poor condition, the system should never mark it as available.

The LIS should support two kinds of users: librarians and borrowers.

This is a clear requirement.

R9 The LIS system shall support users of two kinds: librarians and borrowers.

Librarians can create, update, and delete information about library titles and borrowers.

This means we now have more information about the actor of some of the requirements constructed before. Also, it becomes clear that titles and borrowers can be created, and their information can be updated and deleted. We therefore split some of the requirements expressed earlier and refine them.

R10 (Replaces R3) The librarian can create library titles.
R11 (Replaces R3) The librarian can modify information about library titles.
R12 (Replaces R3) The librarian can create borrowers.
R13 (Replaces R3) The librarian can modify information about borrowers.
R4v2 (Replaces R4) The librarian can register multiple copies of titles.
R5v2 (Replaces R5) The librarian can mark titles as out of date.
R6v2 (Replaces R6) The librarian can mark copies to be in poor condition.

Note that we have simply updated the version for some of the requirements, and replaced some requirements by others. We always indicate which ones they replace for traceability.

They can also generate a report about borrowers and about titles in the LIS.

R14 The librarian can generate reports about library titles.

R15 The librarian can generate reports about borrowers.

A borrower is a library user who can make a loan of a copy of a title that is part of the library holdings. A borrower can also reserve a book or magazine that is not currently available in the library, so that when it is returned or acquired by the library, that person is notified.

These are clear statements about the capabilities of borrowers.

R16 When a copy is available and not reserved, a borrower can create a loan.
R17 When a loan on a copy is created, the copy is marked unavailable.
R18 When a copy is not available and not reserved, a borrower can reserve the copy.
R19 When a copy that is reserved becomes available, the LIS system shall notify the reserving borrower by e-mail.

The reservation is cancelled when the borrower checks out the book or magazine or through an explicit cancelling procedure.

Two operations are described here, that both affect the title copy attributes. We split these in two separate requirements.

R20 When a copy is available and reserved, the reserving borrower can create a loan on that copy.
R21 When a loan on a reserved copy is created, the system removes the reservation on the item.
R22 When a copy is reserved, the reserving borrower can remove the reservation on the item.

Librarians and borrowers make requests and interact with the LIS through a library kiosk (a computer interface).

This is a clear statement.

R23 The LIS system shall have a library kiosk computer interface through which all interactions can be performed.

Loans and returns of library material are processed using an ID scanner.

This should be explicitly stated as a technical requirement of the LIS system.

R24 The LIS system shall provide an ID scanner to process loans and returns of a copy.

Solution to Exercise 3.4

- After being powered on, needs to be initialized at the zero position.

  R3 (M) When inpInit is high, outReady is low, and inpAt0 is low [**Condition**], the system [**Subject**] shall set [**Action**] the outMove0 bit [**Object**] within 1 ms [**Constraint of Action**].
  R4 (M) When inpAt0 is high, and outReady is low [**Condition**], the system [**Subject**] shall set [**Action**] the outReady bit [**Object**] within 1 ms [**Constraint of Action**].

- After the system is initialized, the piston can be moved by sending a move to end or a move to zero commands

  R5 (M) When inpAt0 and outMove0 are high, the system shall unset the outMove0 bit within 1 ms
  R6 (M) When inpAt1 and outMove1 are high, the system shall unset the outMove1 bit within 1 ms
  R7 (M) When inpMove0 is high, inpAt0 is low, outReady is high, outMove0 is low, and outMove1 is low, the system shall set the outMove0 bit within 1 ms
  R8 (M) When inpMove1 is high, inpAt1 is low, outReady is high, outMove0 is low, and outMove1 is low, the system shall set the outMove1 bit within 1 ms

- The system has an emergency stop button. Once this button is pressed, the cylinder will immediately stop moving. The emergency procedure will also re-initialize the cylinder to the zero position.

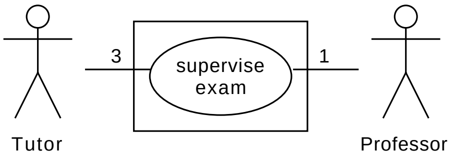
  R9 (M) When inpStop signal is received, the system shall unset the outReady bit within 1 ms.

  R10 (M) When either outMove0 or outMove1 is high and outReady is low, the system shall unset outMove0 and outMove1 within 1 ms.

- Consistency of the signals outMove0, outMove1 and outReady.

  R11 (S) When either outMove0 or outMove1 is high and outReady is low, the system shall unset outMove0 and outMove1 within 1 ms

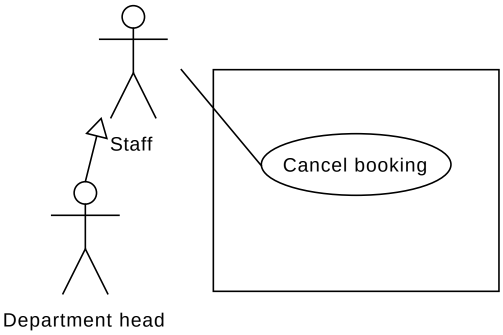
Solution to Exercise 4.1



Solution to Exercise 4.2

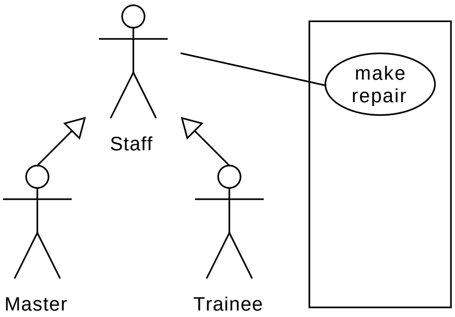
1. False.
2. False.
3. True.
4. True.

Solution to Exercise 4.3



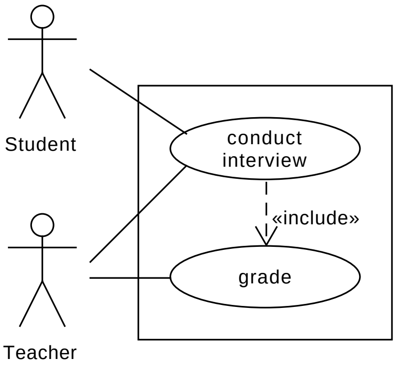
Note that there should *not* be an association between department head and the use case; if this association is there in addition to the one between staff and the use case, this means both a staff member *and* the department head are needed to cancel a booking.

Solution to Exercise 4.4



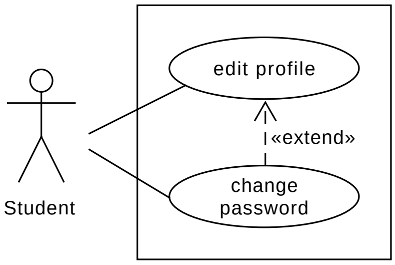
Note that staff should *not* be abstract, since the description indicates there may be other employees besides master and trainee. In case it would have been clear from the context that master and trainee are the only employees that can make the repair we could have used a common abstract actor instead.

Solution to Exercise 4.5



Note that using an extend relation here is not appropriate since the teacher *always* has to grade the student. The exercise clearly suggests that grading is a separate use case, which is why we model the situation as above. Note, however, that theoretically this leaves the possibility that the teacher that grades is different from the one that conducts the interview, this is due to the level of abstraction of use cases. If it is really important that this is one and the same teacher, you may argue that there is just one use case conduct interview that involves a student and a teacher.

Solution to Exercise 4.6



Note that using an include relation here is not appropriate since the student can change their password, but this is optional. Also, if you model strictly according to the UML standard, student must have an association to change password, since the associations are not inherited in any direction along the extends relation (nor are they inherited along an include relation)

Solution to Exercise 4.7

1. For use case A, the valid combinations are:

    - X
    - Y

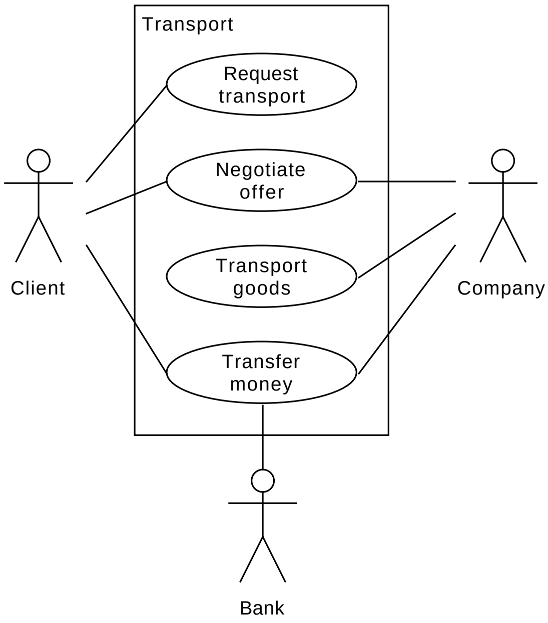
    Note that only one actor at a time can communicate with this use case.

2. For use case B, the only valid combination is Y, since the association from X to A is not inherited along the include relation from A to B.

Solution to Exercise 4.8

1. If we look at the textual specification, we can identify three potential actors: Client, Company and Bank.

2. We create the following use case diagram.

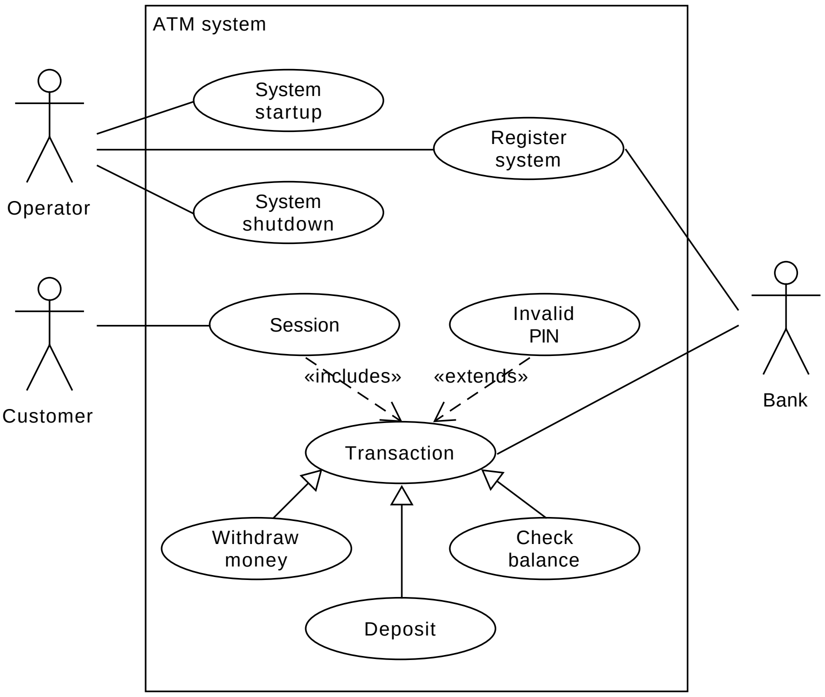


3. We give a detailed description of negotiate offer.

Name:	Negotiate offer
Short description:	The company determines the amount of time required for the transport and scheduled period of transport and makes an offer to the client.
Pre-condition:	The client has requested transport
Post-condition:	The client agreed to the offer
Error situations:	There is no transport available The client does not agree to the offer
Actors:	Client, Company
Trigger:	A transport request has been filed
Standard process:	(1) The company finds the next available vehicle which can accommodate the package (2) The company makes an offer based on the duration of the trip and sends it to the client (3) The client agrees with the terms of the offer. (4) The company sends an invoice and the transport is scheduled.
Alternative process:	(1') No transport that can accommodate the package is available. The use case is terminated. (3") The client does not agree with the terms of the offer. Either restart the use case at (1) or terminate the use case.

Solution to Exercise 4.9

1. The actors are the operator, the customer and the bank.
2. The use case diagram is the following:



3. Detailed use case description:

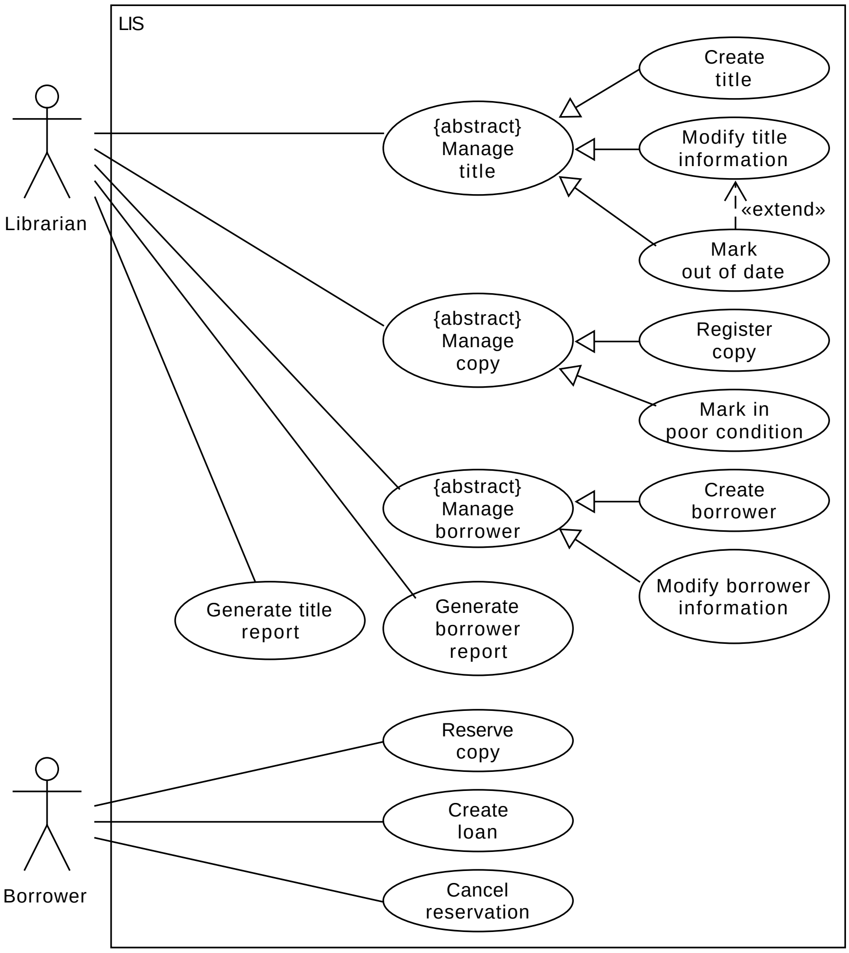
Name:	Withdraw money
Short description:	When the ATM is running, the customer can withdraw money from the ATM.
Pre-condition:	The system is running
Post-condition:	The customer received their money
Error situations:	Invalid PIN was entered Account balance is too low
Actors:	Customer, Bank
Trigger:	A customer requests to withdraw money
Standard process:	(1) The customer enters a valid PIN (2) The customer enters the amount to be withdrawn (3) The account balance is sufficient according to the bank (4) The money is delivered to the customer
Alternative process:	(1') The customer enters an invalid PIN. The Invalid PIN use case is triggered. (3") The account balance is insufficient according to the bank (4") No money is delivered and an error message is shown

Solution to Exercise 4.10

1. We already discussed the actors when talking about requirements. When we again analyse the textual specification, we can identify the actors Librarian and Borrower. In addition, the ID scanner is a potential actor. We assume that the ID scanner is part of the LIS, so we do not include it in the list of actors.

2. To identify the use cases we go through all textual requirements. For managing titles, copies and borrowers we create abstract use cases to be able to register common associations. We consider Mark out of date as one of the possibilities for modifying title information.

3. Now we associate our actors and the use cases. The final use case diagram is the following.



4. Finally, we give a detailed description of one of the use cases.

Name:	Reserve copy
Short description:	When a copy is not available and not reserved, a borrower can reserve the copy
Pre-condition:	The copy is not available
Post-condition:	The borrower has reserved the copy
Error situations:	There is no copy that is not reserved
Actors:	Borrower
Trigger:	A borrower requests to reserve a copy
Standard process:	(1) The borrower checks if at least one copy is not reserved (2) The borrower sets one copy to reserved
Alternative process:	(2') No copy that is not reserved is available. The reservation is aborted.

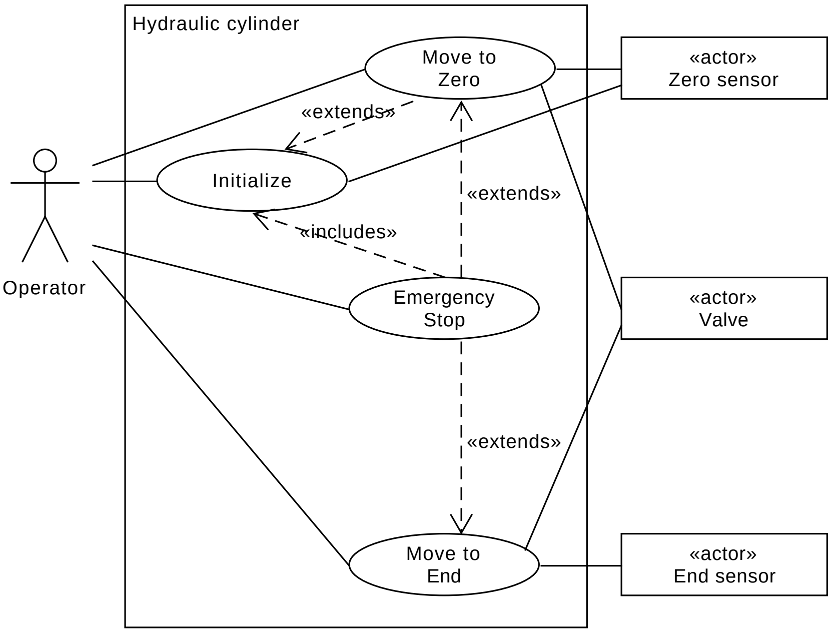
Note that this requirement was directly inferred from requirement R14.

Solution to Exercise 4.11

1. When looking at the description and the requirements, we can identify a number of actors. The operator (or user), the zero- and end-sensors and the valve. One can debate whether the LEDs should be included in the use cases. Furthermore, the emergency stop button may or may not be its own actor. Here we abstract from the LEDs, and we assume the

operator is responsible for pushing an the emergency stop button, so the emergency stop button is part of the system.

We identify four use cases: initialize, move to zero, move to end, and stop. This results in the following use case diagram.



2. Detailed description of the use cases:

Initialization:

Name:	Initialize
Short description:	Initialize the cylinder
Pre-condition:	The cylinder is powered on
Post-condition:	The cylinder is initialized at the zero position. Output outReady is set
Error situations:	
Actors:	Operator
Trigger:	The operator issues the cInit command
Standard process:	(1) The operator issues command cInit (2) The system reads that inpAt0 is set (3) The cylinder sets outShow0
Alternative process:	(2’) The cylinder reads inpAt0 is not set (3) The use case continues with executing use case Move to zero

Move to zero:

Name:	Move to zero
Short description:	The cylinder moves to the zero position
Pre-condition:	outReady is set
Post-condition:	The cylinder is at the zero position. Output outShow0 is set
Error situations:	Cylinder does not reach start position within 10ms
Actors:	Operator, Valve
Trigger:	The operator issues the cMove0 command
Standard process:	(1) The operator issues command cMove0 (2) The system reads inpAt0 is not set (3) The cylinder sets outMove0 (4) The cylinder reads inpAt0 until it is set (5) The cylinder sets outShow0
Alternative process:	(2') The cylinder reads inpAt0 is set. The use case continues at (5) (4') inpAt0 remains not set for more than 10 ms. Use case continues with Emergency Stop.

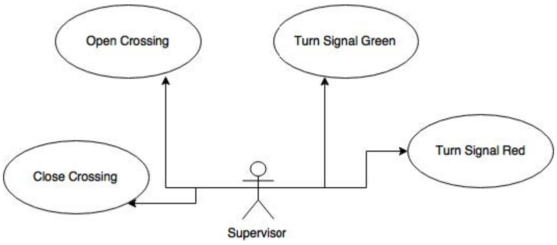
Move to end:

Name:	Move to end
Short description:	The cylinder moves to the end position
Pre-condition:	outReady is set
Post-condition:	The cylinder is at the end position. Output outShow1 is set
Error situations:	Cylinder does not reach end position within 10ms
Actors:	Operator, Valve
Trigger:	The operator issues the cMove1 command
Standard process:	(1) The operator issues command cMove1 (2) The system reads inpAt1 is not set (3) The cylinder sets outMove1 (4) The cylinder reads inpAt1 until it is set (5) The cylinder sets outShow1
Alternative process:	(2') The cylinder reads inpAt1 is set. The use case continues at (5) (4') inpAt1 remains not set for more than 10 ms. Use case continues with Emergency Stop.

Emergency stop:

Name:	Emergency stop
Short description:	The cylinder quickly stops and reinitializes
Pre-condition:	outReady is set
Post-condition:	The cylinder is not moving and outputs its position. Position might be unknown
Error situations:	-
Actors:	Operator, Valve
Trigger:	The operator issues the cStop command, or other use cases detect an emergency
Standard process:	(1) The operator issues command cStop (2) The cylinder unsets outMove0 (3) The cylinder unsets outMove1 (4) The cylinder executes Initialize
Alternative process:	None

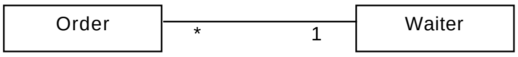
Solution to Exercise 4.12



Solution to Exercise 5.1

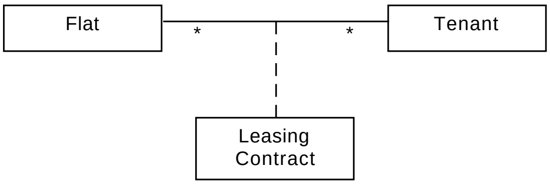
1. center, radius
2. color
3. there are no public attributes.
4. there are no private operations.
5. calculateArea
6. draw, getCenter, getRadius, getArea, setSize

Solution to Exercise 5.2

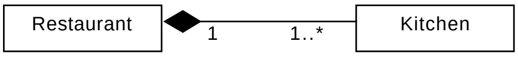


Note that it is also correct to write 0..∗ instead of ∗. If you write 1..∗ that may be correct depending on the situation, however, the description does not give you any information that requires one waiter to handle at least one order, hence 0..∗, or simply ∗, is preferred.

Solution to Exercise 5.3 This is a typical case for using an association class. Leasing contract is an association class that belongs to the association between flats and tenants. The class diagram is as follows.

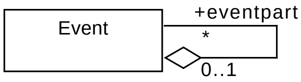


Solution to Exercise 5.4



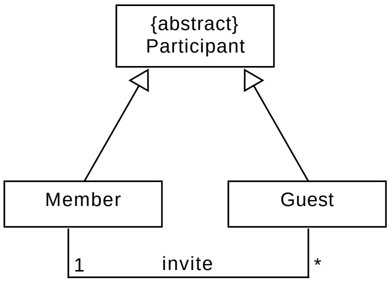
Note that when the restaurant is destroyed, the kitchen also is destroyed, hence we use a composition. The diamond for the composition is always on the side of the whole. The multiplicity 1 can be omitted, since it is assumed the default.

Solution to Exercise 5.5



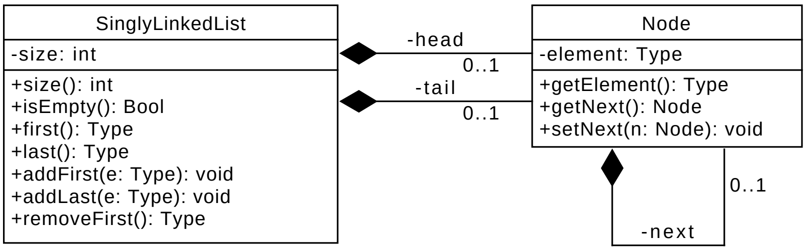
The event may be part of another event, so there is a part/whole relationship. However, an event is not always part of another event, so composition is too strong.

Solution to Exercise 5.6



Note that the question specifies there are exactly two kinds of participants, and we model both kinds explicitly, hence there are no instances of Participant, so it is marked abstract.

Solution to Exercise 5.7 The singly linked list can, e.g., be modelled using the following class diagram.



Note that we here chose to model the references to the head and tail of the list as associations, similar to what is done in (Seidl et al., 2015, Figure 4.12 (a)). The reference to the next node is done in a similar way. We choose to record size as an attribute, since that allows constant time retrieval of the size of the list; if size is not maintained as attribute, its implementation is linear in the size of the list. We also define appropriate operations for the other methods described in the informal text.

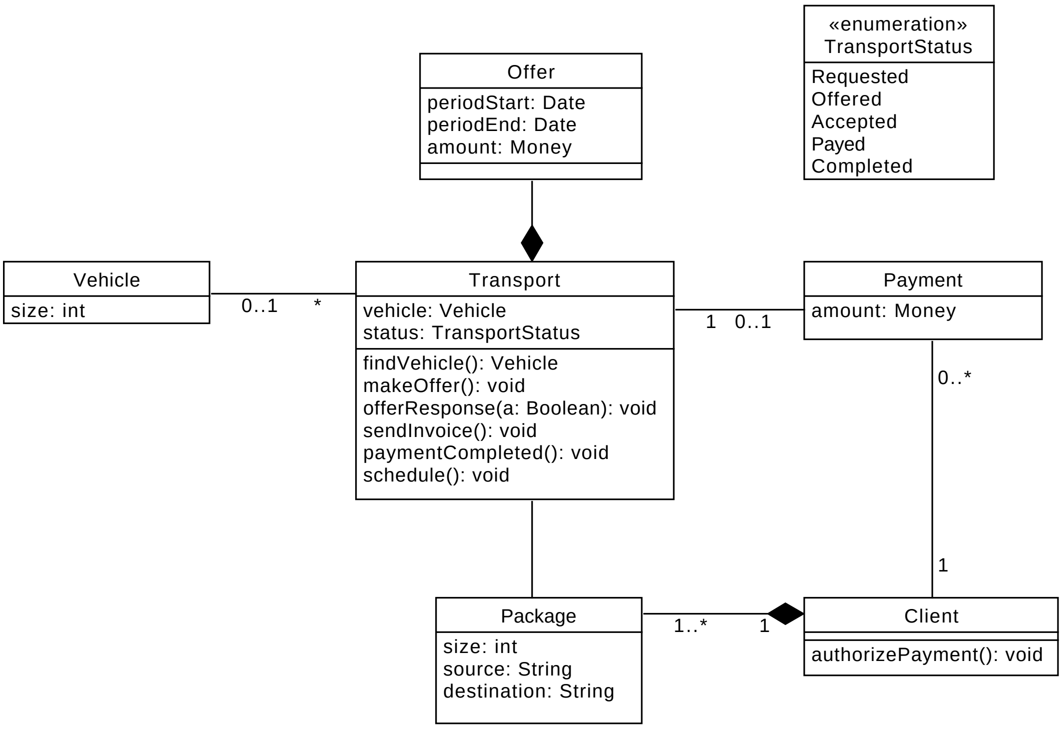
We have chosen a composition relation for head, tail, and next, reflecting the intuition that nodes cannot exist without the linked list object, i.e., there is a clear part/whole relationship.

Solution to Exercise 5.8 First we define the classes. In this case we have a Transport that can be requested, a Vehicle that transports the packages, an Offer, a Client that wants a package transported, a Package to be transported and a Payment (transaction) that is handled by the bank.

We then add the variables to the classes. A vehicle has a size. The transport has a vehicle that it is associated with, and a status. The client has a package, and a package has a given size, source and destination. A payment has an amount paid. An offer made for a transport has a period (pair of dates) in which the transport can be scheduled and an amount to be paid for the transport.

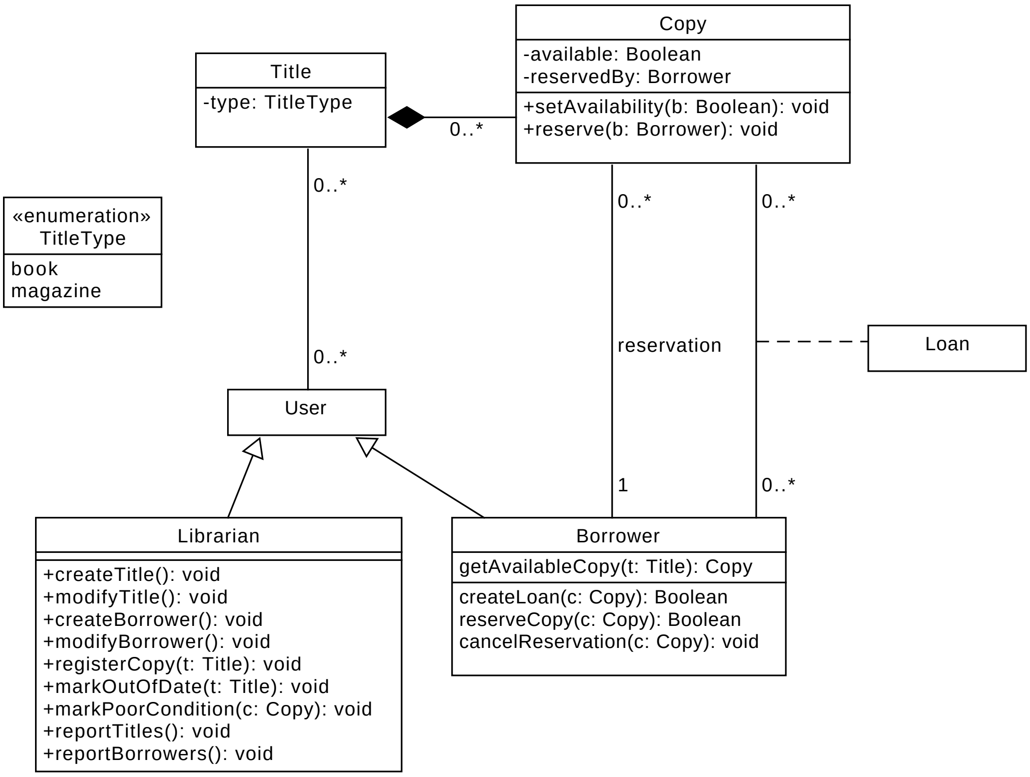
We then add operations; in particular, the client can authorize the payment. Most of the work is done in the transport: we can find a vehicle, make an offer, register the response to the offer, send an invoice, register that payment has been completed, and schedule the transport as specified in the offer.

Finally, we add the relations between the classes. This all results in the following class diagram.

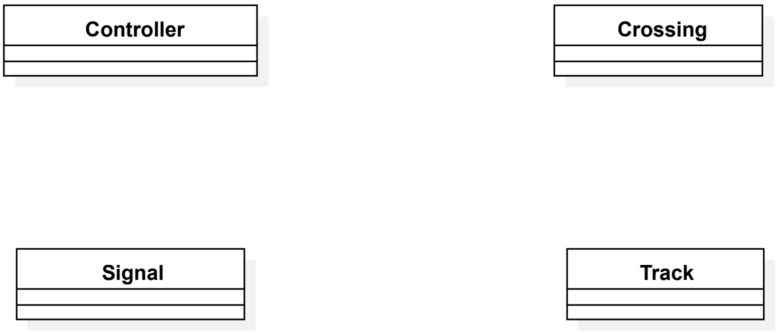


Solution to Exercise 5.9 First we define the classes. These are, at least, Title, Copy, User, Librarian. One could decide to make classes for Book and Magazine, however, according to the description these are treated in exactly the same way, so here we decide to make these explicit classes. Instead, we create the data type TitleType; this is a more extensible solution when the library wants to add other types such as ebook that should also be handled in the same way. Depending on the scope of your system, you may also want to include the IDScanner as a class.

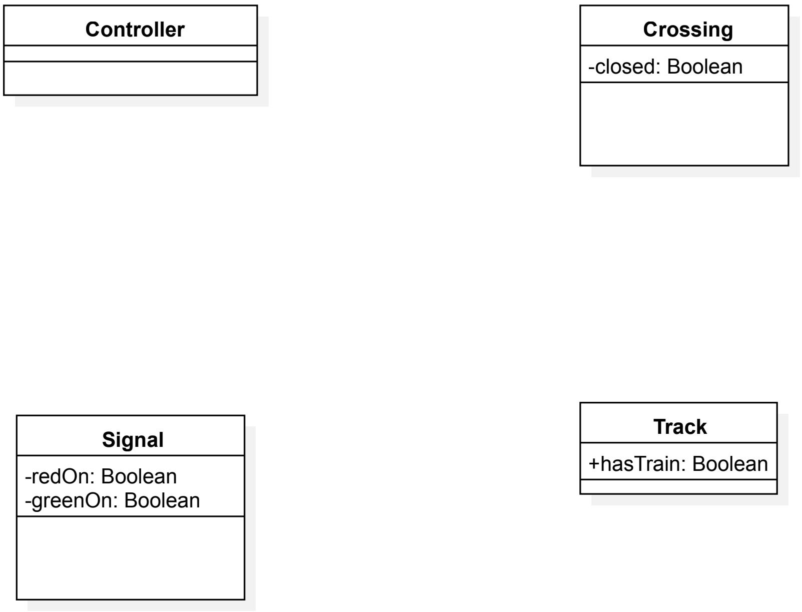
We next add the attributes to the classes, based on the information from the description and the requirements. We also add the operations and associations. Note that we use an association class for the loans, so that an item can be borrowed by multiple borrowers, and multiple items can be borrowed by a single borrower. For reservation, the requirements suggest that any copy can only be reserved by one borrower at a time, and it appears there is no need for keeping a history, hence we make the reservation an attribute of the copy. The class diagram is the following.



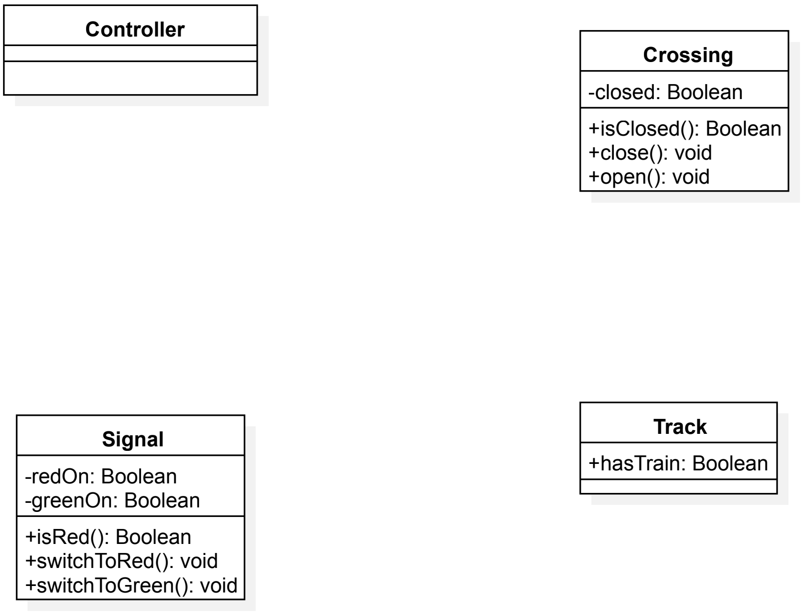
Solution to Exercise 5.10 After examining the system description above one can find the following classes:



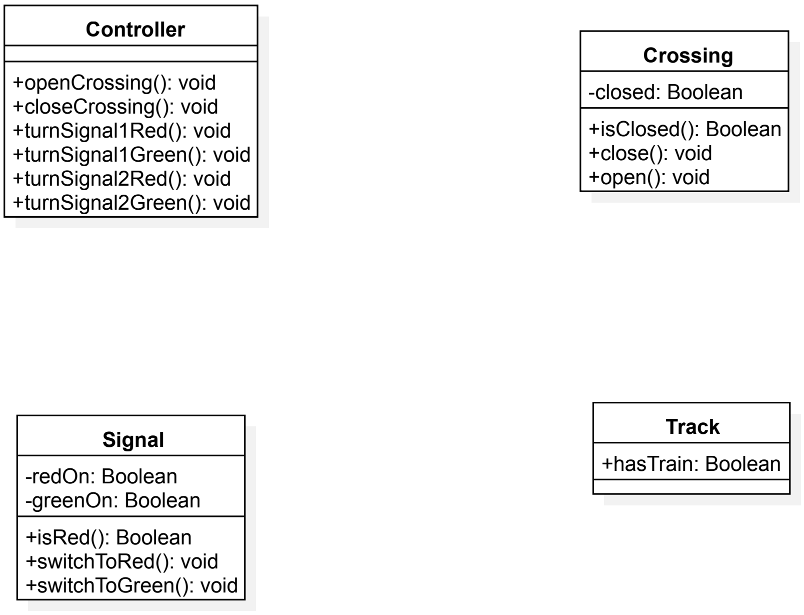
Now let's think about some attributes that the classes above can above. Here are some valid examples:



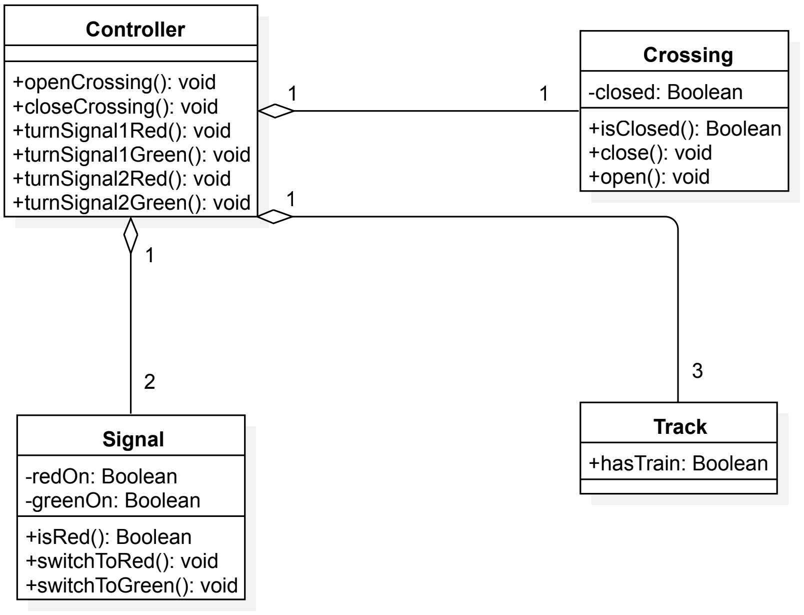
Now let's see what operations are necessary:



But this is not enough, now let's think about the controller's operations:

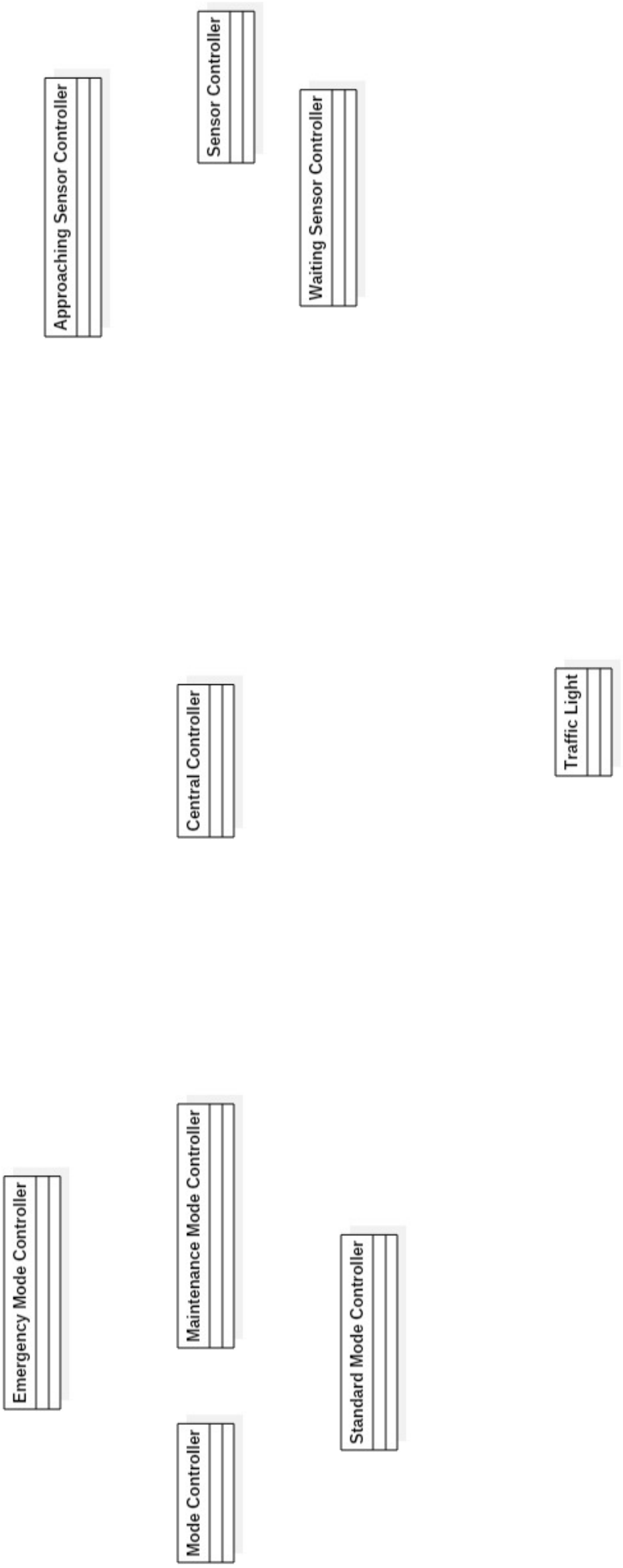


And last but not least let's see what are the relationships between the classes:

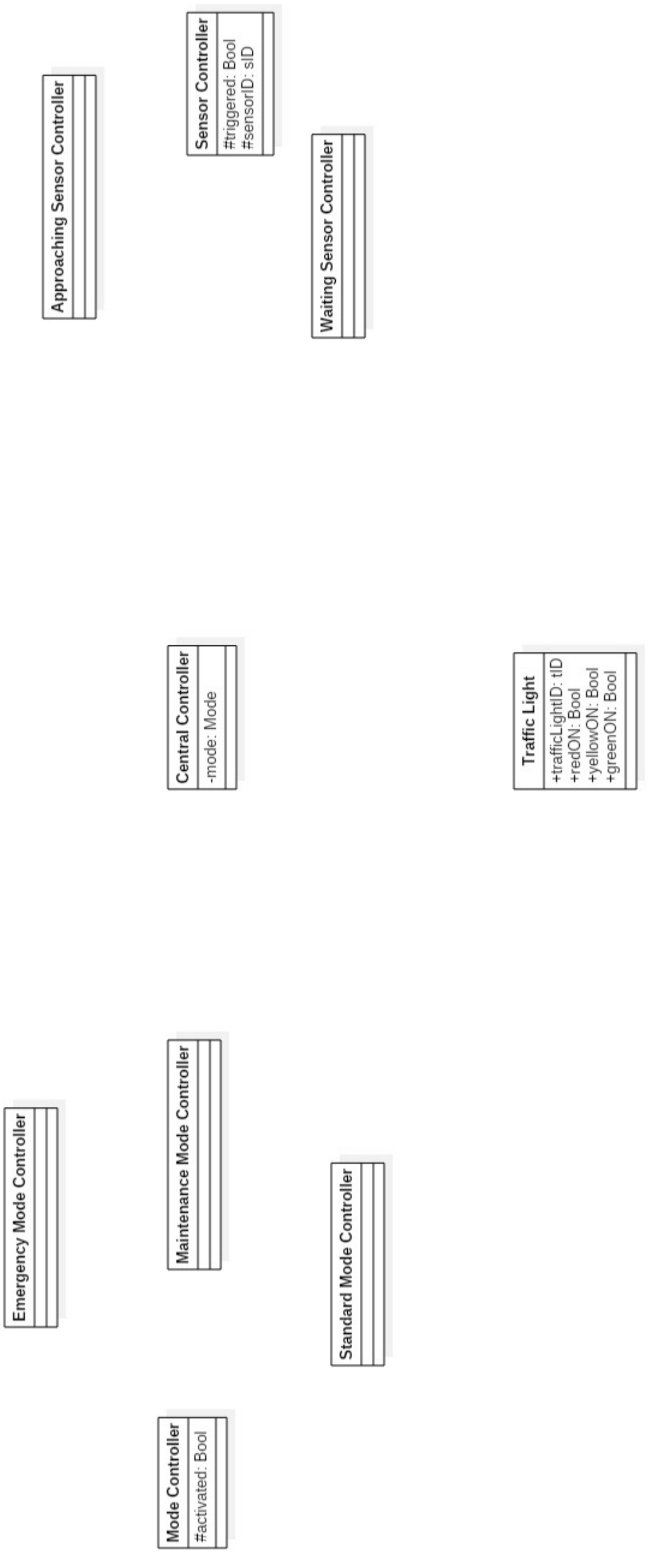


Note that in this case we decided to make the multiplicities exactly as described in the assignment. Of course, you could think about generalizing the class diagram in different ways. For instance, you could be less strict in the multiplicities, but that would mean you also need to reconsider the operations of the controller class.

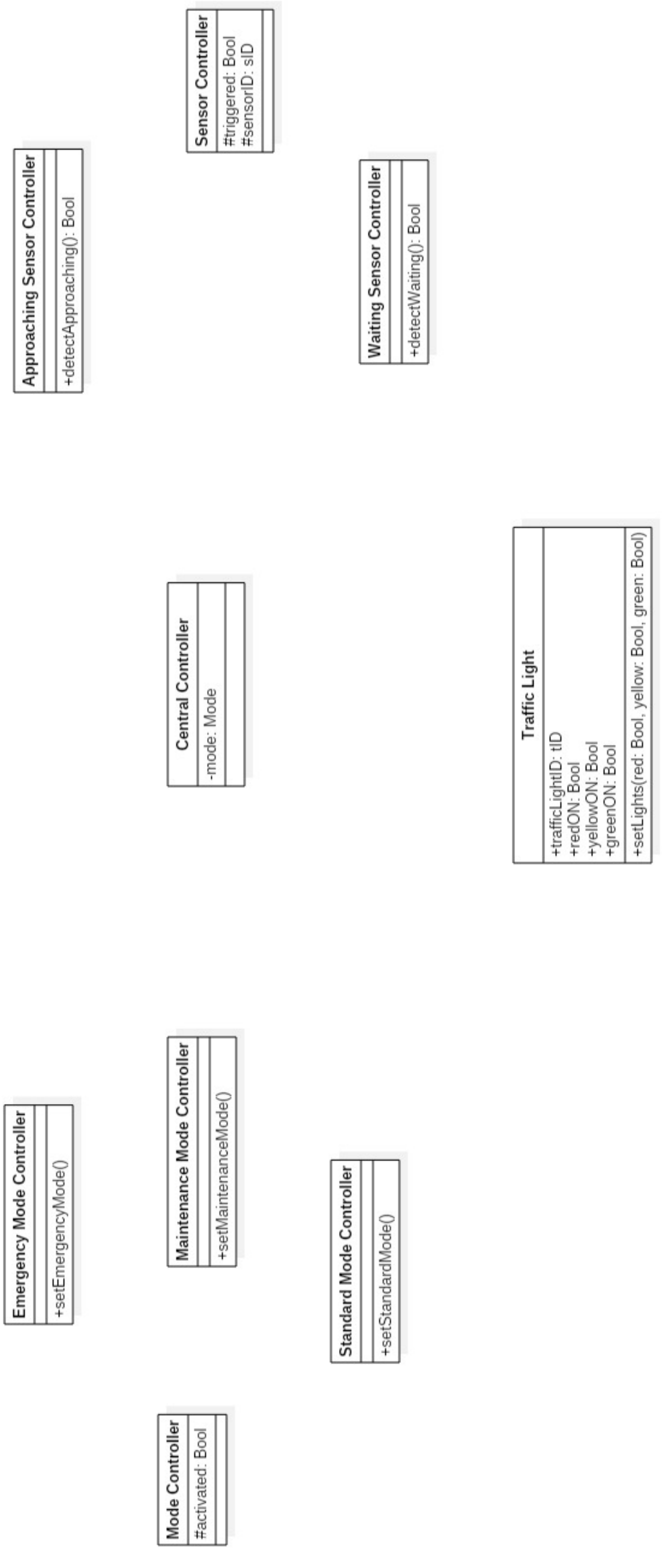
Solution to Exercise 5.11 After examining the system description above one can find the following classes:


Approaching Sensor Controller
Sensor Controller
Waiting Sensor Controller
Central Controller
Traffic Light
Emergency Mode Controller
Maintenance Mode Controller
Standard Mode Controller
Mode Controller

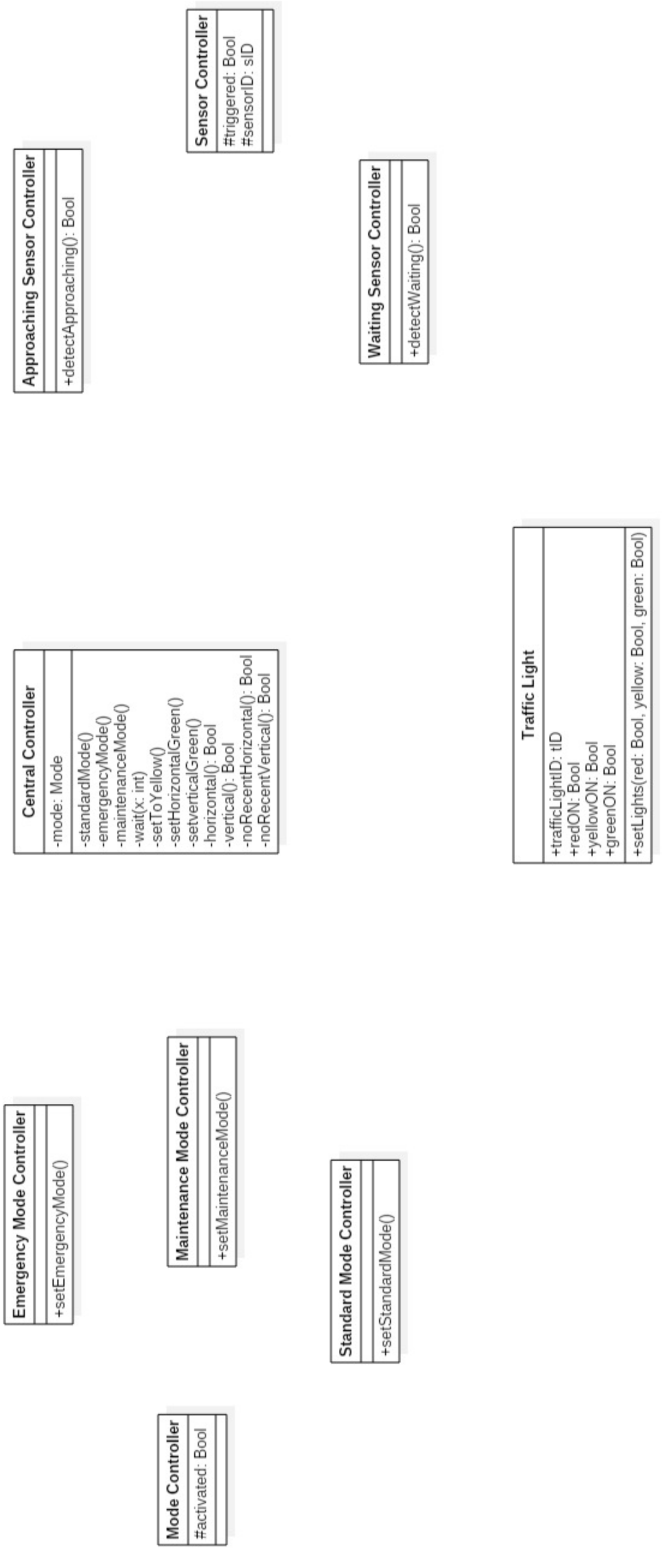
Now let's think about some attributes that the classes above can above. Here are some valid examples:



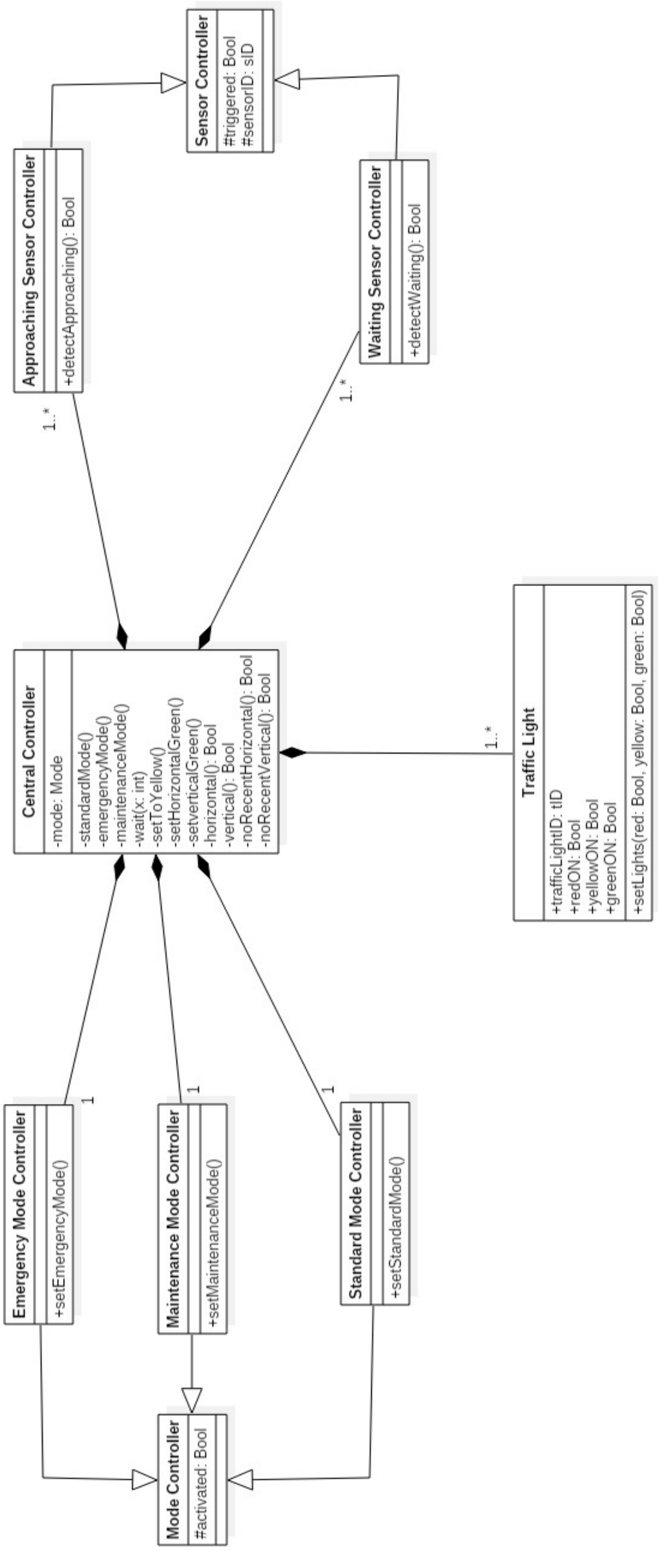
Now let's see what operations are necessary:



But this is not enough, now let's think about the fun part, the central controller's operations:



And last but not least let’s see what the associations are:



Solution to Exercise 6.1

1. We give a table containing the different configuration that we move through. Note that

after the second e4 the completion transition of S1 is executed.

Event	Configuration
Start	{S3}
e2	{S1, S1.1}
e4	{S1, S1.2}
e4	{S2}

2. We again give a table. Note that the e3 event is consumed and does not have any effect.

Event	Configuration
Start	{S3}
e1	{S1, S1.2}
e3	{S1, S1.2}
e4	{S2}

Solution to Exercise 6.2

1. In configuration {A, C, E}, when event e1 occurs, the transition from A is executed, and the machine reaches configuration {G}. In that configuration, when e3 occurs, state A is entered through the deep history pseudo-state, so the state machine returns to configuration {A, C, E}.
2. In configuration {A, C, E}, when event e1 occurs, the transition from A is executed, and the machine reaches configuration {G}. In that configuration, when e5 occurs, state A is entered through the history pseudo-state, so only the history of the children of state A is preserved, so the state machine returns to state C, but its descendants are initialized normally, so the state machine goes to configuration {A, C, D}.
3. In configuration {A, C, E}, when event e1 occurs, the transition from A is executed, and the machine reaches configuration {G}. In that configuration, when e6 occurs, state A is not entered through any history pseudo-states, so we go to its initial child. The state machine hence ends up in configuration {A, B}.

Solution to Exercise 6.3

1. We present the table with the configurations the state machine diagram moves through.

Event	Configuration
Start	{S1, S2}
e1	{S1, S3}
e2	{S5}
e2	{S1, S3}
e1	{S1, S4}

So, the state machine is in configuration {S1, S4}.

2. We present the table with the configurations the state machine diagram moves through.

Event	Configuration
Start	{S1, S2}
e1	{S1, S3}
e2	{S5}
e3	{S1, S2}
e1	{S1, S3}

So, the state machine is in configuration {S1, S3}. The difference with the previous execution is that the transition with event e3 does not go to the history state, so the default initialization of state S1 is used.

3. We present the table with the configurations the state machine diagram moves through.

Event	Configuration
Start	{S1, S2}
e1	{S1, S3}
e1	{S1, S4}
e2	{S5}
e2	{S1, S4}

4. We present the table with the configurations the state machine diagram moves through.

Event	Configuration
Start	{S1, S2}
e1	{S1, S3}
e1	{S1, S4}
e2	{S5}
e2	{S1, S4}
e1	{S1, S2}
e1	{S1, S3}
e2	{S5}
e2	{S1, S3}

Solution to Exercise 6.4

1. P1 is a simple state, Q is a composite state, U is an orthogonal state (which is, in fact, a special kind of composite state), and S is a region in orthogonal state U; for all practical purposes, S can itself be handled as a composite state.
2. We have the following configurations in terms of the root:
   - Every configuration contains root.
   - Every configuration contains either P1 or P2 or Q.
   - Every configuration that contains Q contains either Q1 or U.
   - Every configuration that contains U contains both S and T since U is an orthogonal state.
   - Every configuration that contains S contains either S1 or S2
   - Every configuration that contains T contains either T1 or T2.

   The configurations with respect to the root of the diagram are thus the following. Note that we assume root is an implicit member of all configurations.

   - {P1}
   - {P2}
   - {Q, Q1}
   - {Q, U, S, T, S1, T1}
   - {Q, U, S, T, S2, T1}
   - {Q, U, S, T, S1, T2}
   - {Q, U, S, T, S2, T2}

3. We present the table with the configurations the state machine diagram moves through.

Event	Configuration
Start	{P1}
a	{P2}
a	{Q, U, S, T, S1, T1}
d	{Q, U, S, T, S2, T1}

   Note that after the second a, we immediately go to S1, but since we need to end up in a valid configuration, T, which is also part of orthogonal state U is also initialized.
4. We present the table with the configurations the state machine diagram moves through. Note that this extends the previous question.

Event	Configuration
Start	{P1}
a	{P2}
a	{Q, U, S, T, S1, T1}
d	{Q, U, S, T, S2, T1}
f	{Q, U, S, T, S1, T1}

Note that when handling event f, there are two transitions that we could take, the one from Q and the one from state S2. Since S2 is a descendant of Q, the standard prescribes that the transition from S2 is taken.

5. We present the table with the configurations the state machine diagram moves through.

Event	Configuration
Start	{P1}
a	{P2}
b	{Q, Q1}
e	{Q, U, S, T, S1, T1}
c	{Q, U, S, T, S1, T2}

Solution to Exercise 6.5 We present the table with the configurations the state machine diagram moves through, along with the values of the variables. We also indicate the order in which activities are handled. The table only lists a configuration in the row that processes the last activity of an event.

1. For the sequence e2, e1, e4, e3, e5 we obtain the following.

Event	Configuration	Comment	x
Start		start	1
		entry of A	2
	{A, B}	entry of B	3
e2		$x++$ (self-transition)	4
	{A, B}	entry of B	5
e1	{A, C}		
e4		exit of C	6
	{A, C}	$x = x * 2$ (self-transition)	12
e3		$x++$ in C	13
e5		exit of C	14
		exit of A	27
	{D}	entry of D	26

So, the value is 26.

2. For sequence e1, e3, e4, e4, e3, e5 we get:

Event	Configuration	Comment	x
Start		start	1
		entry of A	2
	{A, B}	entry of B	3
e1	{A, C}		
e3		$x++$ in C	4
e4		exit of C	5
	{A, C}	$x = x * 2$ (self-transition)	10
e4		exit of C	11
	{A, C}	$x = x * 2$ (self-transition)	22
e3		$x++$ in C	23
e5		exit of C	24
		exit of A	47
	{D}	entry of D	46

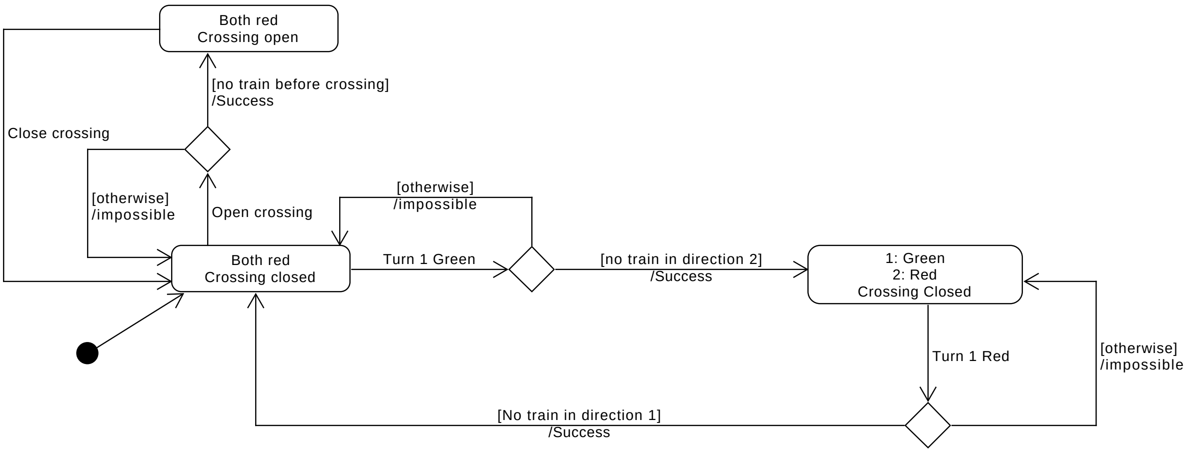
So, the value is 46.

3. For the sequence e2, e2, e1, e3, e5 we get:

Event	Configuration	Comment	x
Start		start	1
		entry of A	2
	{A, B}	entry of B	3
e2		$x++$ (self-transition)	4
	{A, B}	entry of B	5
e2		$x++$ (self-transition)	6
	{A, B}	entry of B	7
e1	{A, C}		
e3		$x++$ in C	8
e5		exit of C	9
		exit of A	17
	{D}	entry of D	16

So, the value is 16.

Solution to Exercise 6.6 We present part of the state machine diagram. In particular, we only show the part of the state machine where we attempt to turn signal 1 to green; the part of the state machine for signal 2 is similar.

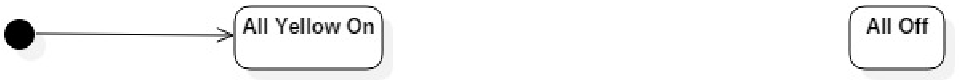


The controller is initially in a safe state, with the crossing closed, and both signals red. When an event occurs, the state machine checks the conditions, and depending on the outcome, modes to another state.

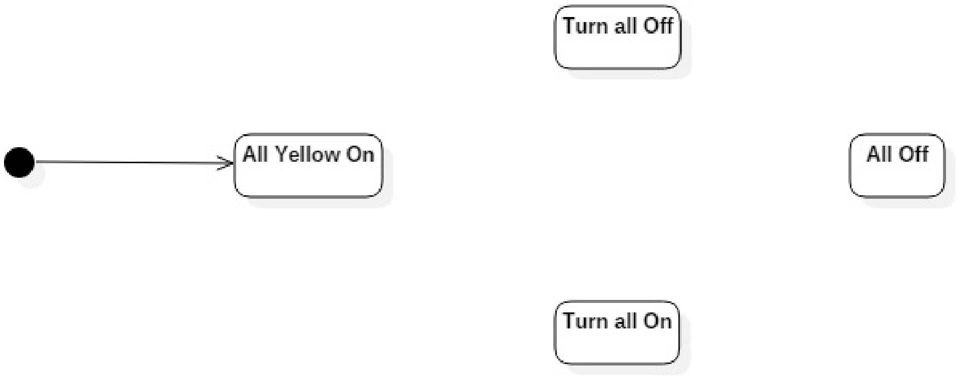
Solution to Exercise 6.7 First, we need to identify the states in which our system can be. We can see that the system has 3 modes, namely: standard, maintenance and emergency. These modes/states in turn have states contained in them. So we need to think about them too. Only ”Emergency Mode” can be seen as a single state, since all it does is keep all lights on both horizontal and vertical road red. So lets keep it as a single state, but we need to model both ”Maintenance Mode” and ”Standard Mode” as separate state diagrams, which we can then combine in one bigger state diagram which is going to represent the complete state diagram of the system. So lets go.

We will start with the state machine diagram representing the ”Maintenance Mode”. Which, while no button is pressed, the system only blinks all traffic lights yellow.

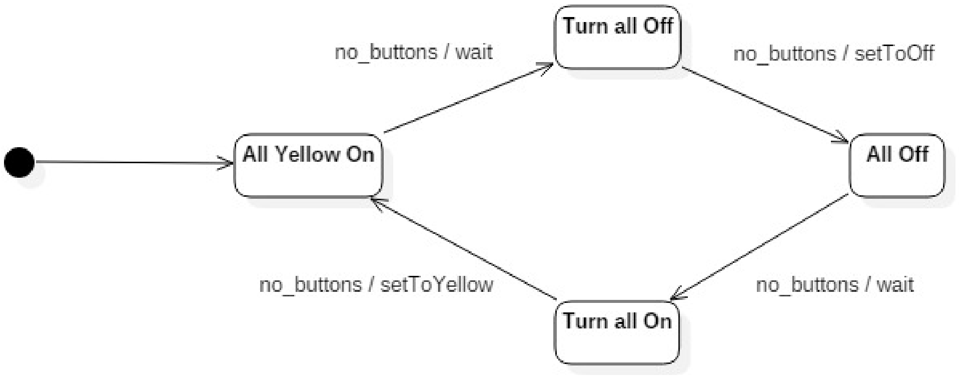
We start off by having the initial state, all yellow lights on, and a state for all lights off:



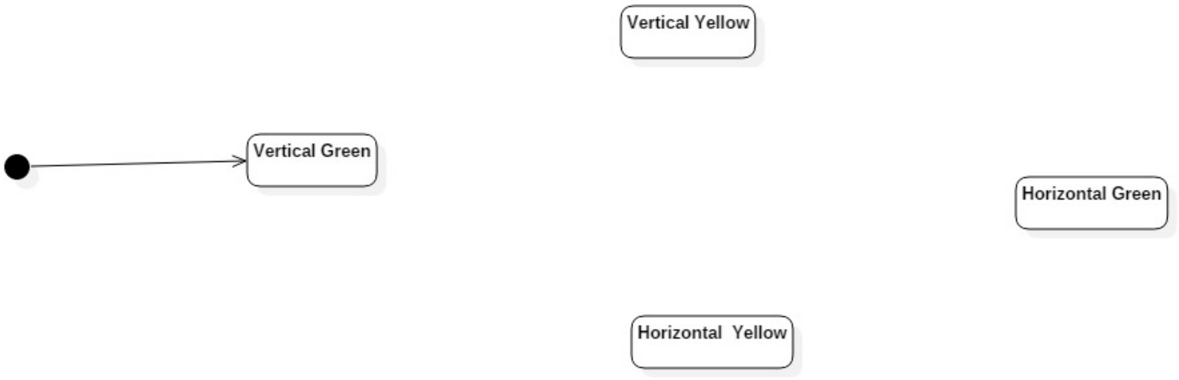
However, we will also need to model them as on or off for some time, so let's add two intermediate states which are going to be used to switch the lights to on or off.



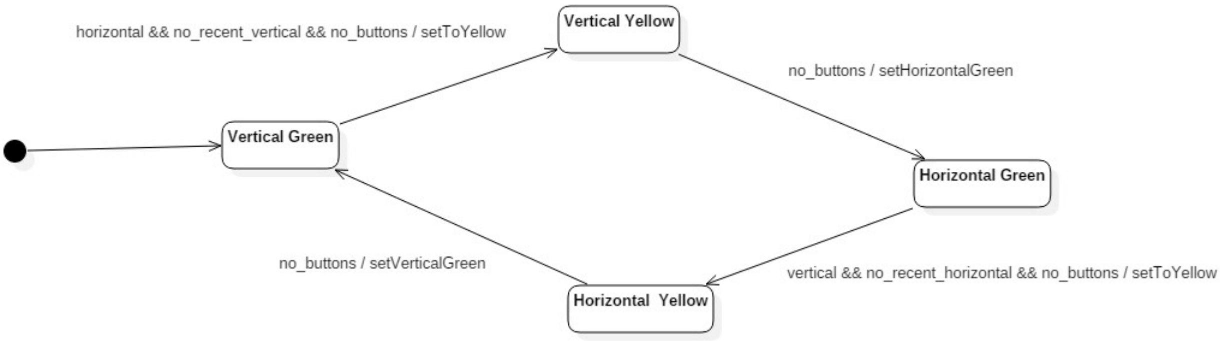
And to finalize it, let's add some transitions.



Now lets do the "Standard Mode". As a start we need to 4 states: Vertical Green (here horizontal are red), Vertical Yellow (here horizontal are red), Horizontal Green (here vertical are red), Horizontal Yellow (here vertical are red).

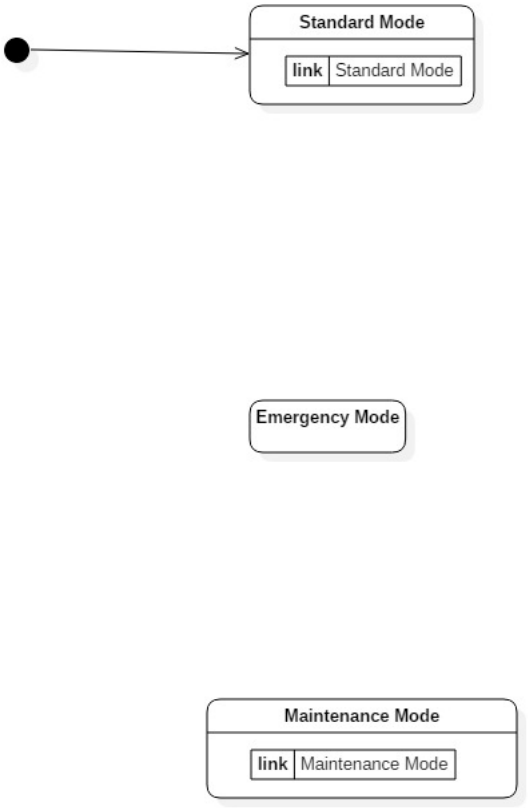


Here we do not really need any additional states, so lets just add the transitions.

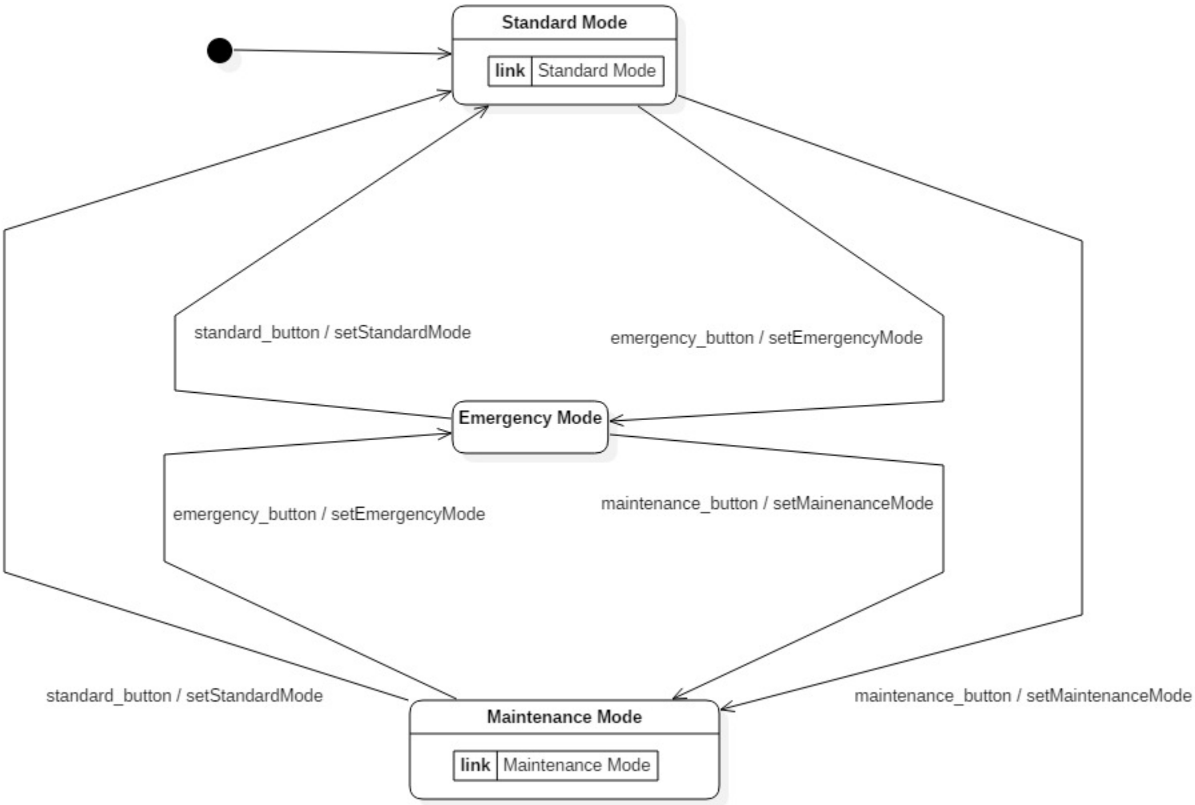

horizontal && no_recent_vertical && no_buttons / setToYellow
Vertical Yellow
no_buttons / setHorizontalGreen
Vertical Green
Horizontal Green
no_buttons / setVerticalGreen
vertical && no_recent_horizontal && no_buttons / setToYellow
Horizontal Yellow

And now we need the whole system. Lets build the state diagram using the diagrams we have built above.

Firstly add the states:



Well this is enough actually, so lets finish it up by adding the transitions.


Standard Mode
link | Standard Mode
standard_button / setStandardMode
emergency_button / setEmergencyMode
Emergency Mode
emergency_button / setEmergencyMode
maintenance_button / setMainenanceMode
standard_button / setStandardMode
maintenance_button / setMaintenanceMode
Maintenance Mode
link | Maintenance Mode

Solution to Exercise 7.1 Since the sequence diagram contains a par fragment, there are no chronological connection between the messages from the different operands. There only is a chronological connection between a and b, and between c and d. Therefore, we get the following traces:

- $a \to b \to c \to d$
- $a \to c \to b \to d$
- $a \to c \to d \to b$
- $c \to a \to b \to d$
- $c \to a \to d \to b$
- $c \to d \to a \to b$

Solution to Exercise 7.2 Since the sequence diagram contains a par fragment at the highest level, messages in each of the operands are executed in parallel. Note that the messages b and c within the critical fragment cannot be interleaved with the other messages, so b and c appear atomically. However, within the critical fragment, the standard order seq applies, and since there are no dependencies between b and c if we look at the lifelines, b and c may appear in any order. Therefore, we get the following traces:

- $a \to b \to c \to d$
- $a \to d \to b \to c$
- $b \to c \to a \to d$
- $b \to c \to d \to a$
- $d \to a \to b \to c$
- $d \to b \to c \to a$
- $a \to c \to b \to d$

- $a \rightarrow d \rightarrow c \rightarrow b$
- $c \rightarrow b \rightarrow a \rightarrow d$
- $c \rightarrow b \rightarrow d \rightarrow a$
- $d \rightarrow a \rightarrow c \rightarrow b$
- $d \rightarrow c \rightarrow b \rightarrow a$

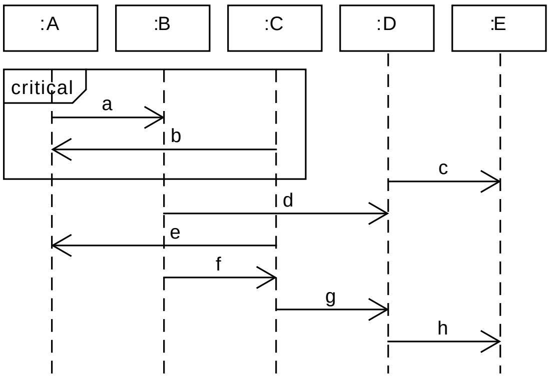
Note that in each of these traces, b and c appear next to one another (but in any order).

Solution to Exercise 7.3 The class has at least all operations which are invoked by itself and other classes meaning that a message is sent from a class to this particular class. A message may be labeled with the syntax `name (arguments)` A return message may be labeled with `attribute=name:value` or short `name:value`.

- Class A. Only one operation of A is invoked, y with parameter 123, and `int`, hence `y(int): void`
- Class B. Here two operations of B are invoked:
  1. x without parameters. The return message has not been modeled explicitly, thus we assume that there is no return value (`void`).
  2. x without parameters with a return message (return value "abc" - apparently a `String`).

  Based on the sequence diagram, the correct answers therefore are `x():void` and `x(): String`. Note that may languages (especially strongly typed ones) forbid the presence of both methods in a single class. However, this is still the information we can infer from the sequence diagram.
- Class C. Only one operation of C is invoked, Z without parameters, and the return message has not been modelled explicitly. Hence the answer is `z(): void`

Solution to Exercise 7.4 There are multiple possible solutions. The most straightforward is to put a and b into a critical fragment.

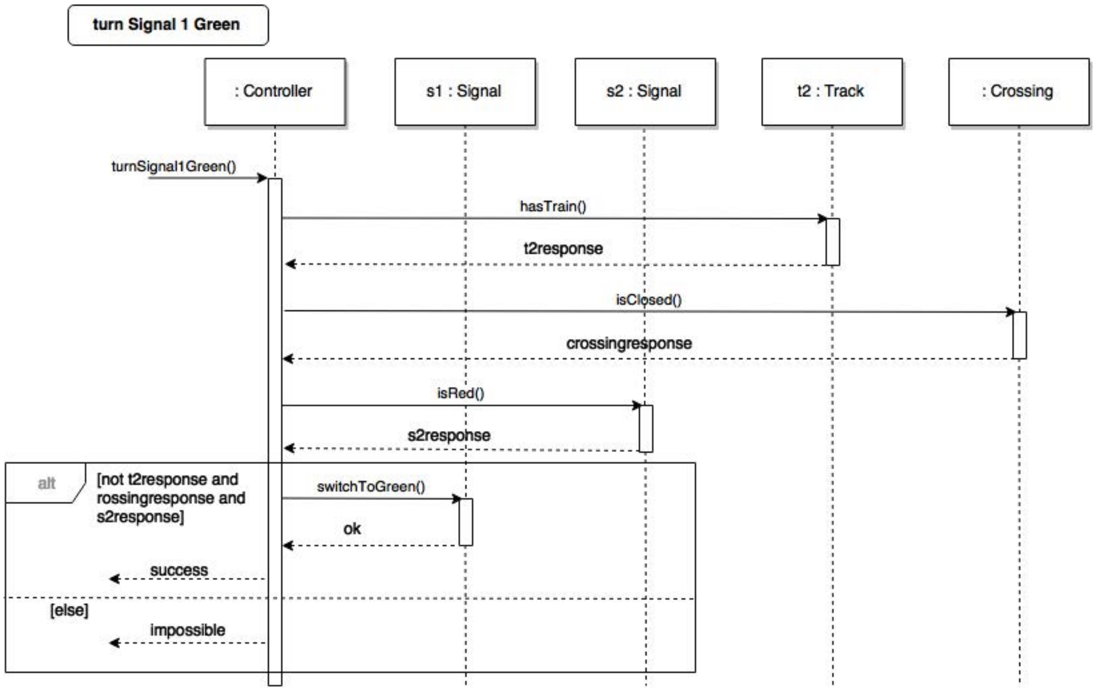


Solution to Exercise 7.5 The following sequence diagram describes how to turn signal 1 Green.

The description states that, if the supervisor decides to change a signal to green, the controller should make sure that:

1. there is no train coming in the opposite direction towards the signal
2. the crossing is closed
3. the signal in the opposite direction is red.

If all these are true then the controller can switch the signal to green, if not it should not. This gives the following sequence diagram:

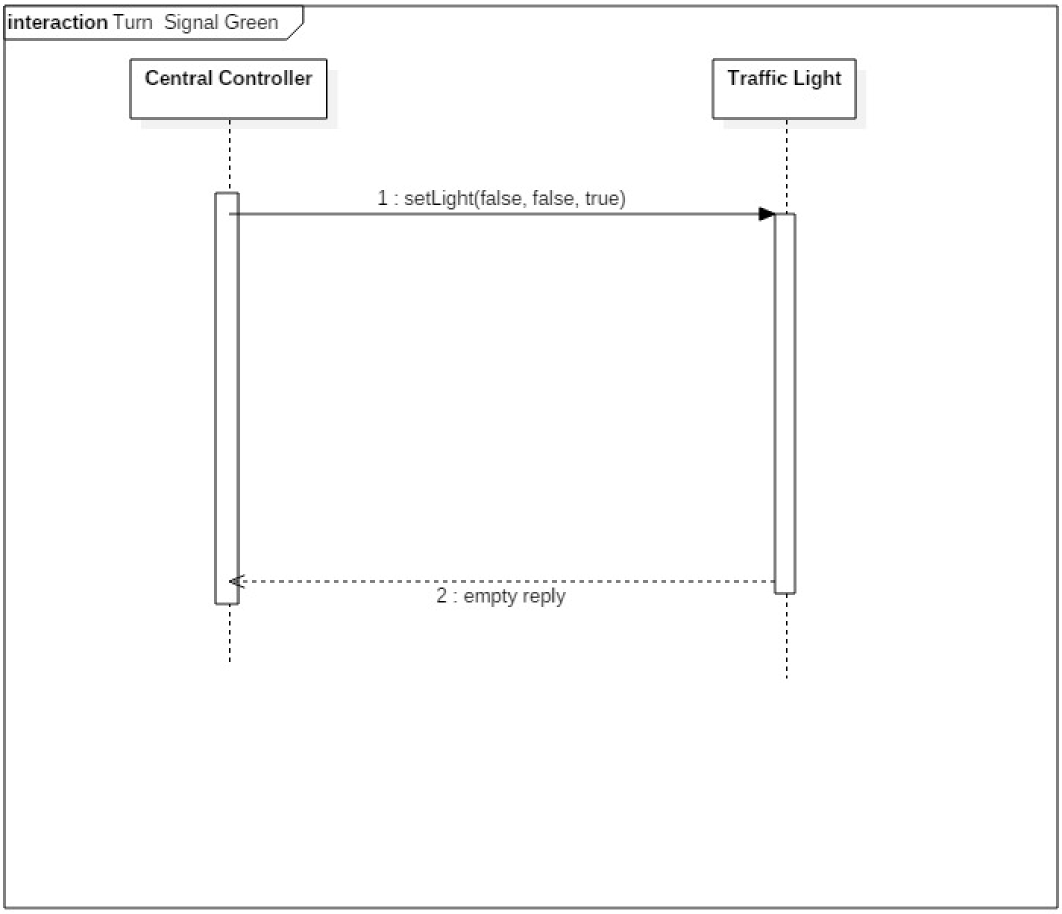


Observe that all objects in the sequence diagrams are instances of classes from the class diagram. Also, note that all messages are operations in the class diagram, and the objects are connected appropriately in the class diagram.

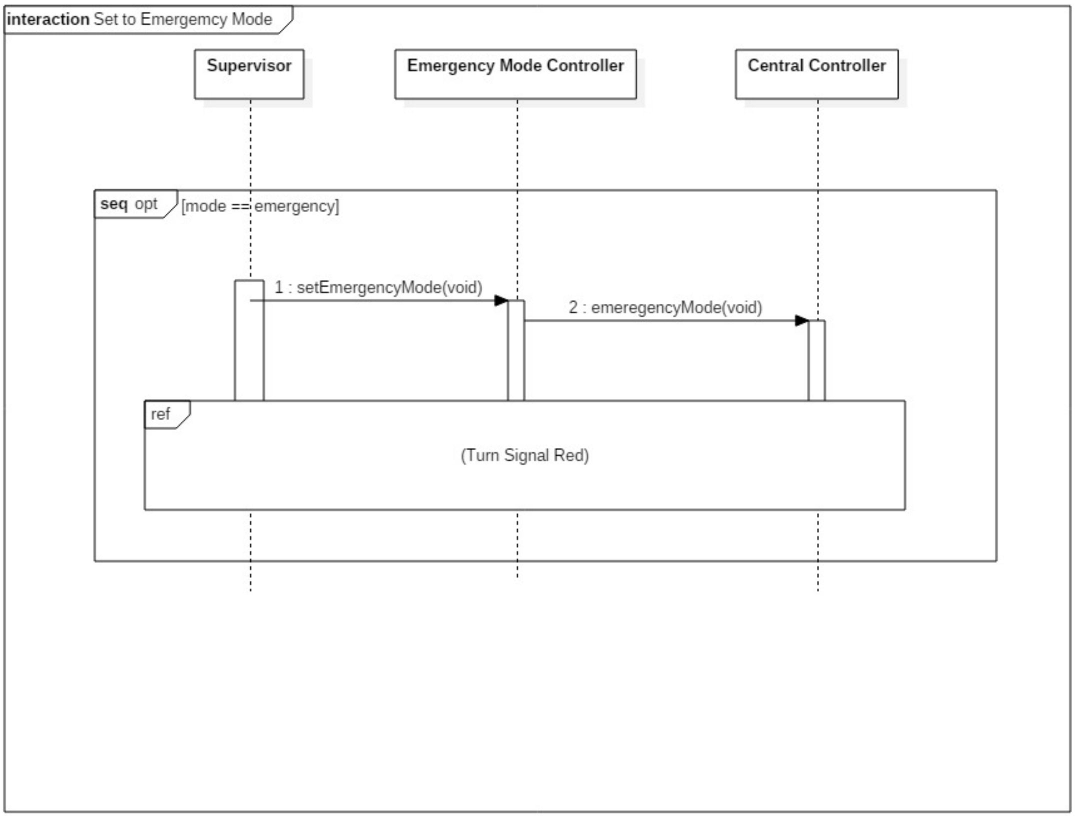
Solution to Exercise 7.6 We give the sequence diagrams for two of the use cases.

First let's look at how to change the colours of the lights. All three cases (change lights to green, yellow, or red) are basically the same. They only differ in what parameters the Central Controller sends to the Traffic Light. The blinking yellow light for Maintenance mode is done via a loop frame in the sequence diagram.

This is how the Central Controller sets the lights of the Traffic Light to green.


interaction Turn Signal Green
Central Controller
Traffic Light
1 : setLight(false, false, true)
2 : empty reply

Now that we know how to change the light of the Traffic Light to some colours lets see how we can represent the different modes in a sequence diagram. All three modes are analogous, so here only the case for Emergency Mode is presented.


interaction Set to Emergemcy Mode
Supervisor
Emergency Mode Controller
Central Controller
seq opt [mode == emergency]
1 : setEmergencyMode(void)
2 : emeregencyMode(void)
ref
(Turn Signal Red)

Solution to Exercise 8.1

- $A \to C$
- $A \to B \to C$
- $B \to A \to C$

Solution to Exercise 8.2

- $A \to C$
- $A \to B$
- B

Solution to Exercise 8.3

- $A \to C$
- $A \to B \to D$

Solution to Exercise 8.4

- $A \to C$
- $A \to B \to C$

Solution to Exercise 8.5

- $A \to B$

- $A \rightarrow C$

Solution to Exercise 8.6

- $A \rightarrow B \rightarrow D$
- $A \rightarrow C \rightarrow D$
- $A \rightarrow B \rightarrow C \rightarrow D$
- $A \rightarrow C \rightarrow B \rightarrow D$

Solution to Exercise 8.7

- $A \rightarrow D$
- $A \rightarrow B \rightarrow D$
- $A \rightarrow B \rightarrow C$

Solution to Exercise 8.8 Note this diagram has exactly the same meaning as the one in Exercise 8.7. The book writes about this: "If a token is present at an incoming edge of an activity final node, the entire activity is terminated—that is, all active actions of this activity are terminated."

- $A \rightarrow D$
- $A \rightarrow B \rightarrow D$
- $A \rightarrow B \rightarrow C$

Solution to Exercise 8.9 Compared to exercises Exercise 8.7 and 8.8, we use a join node to combine both paths. To terminate the activity, both C and D need to be terminated.

- $A \rightarrow B \rightarrow C \rightarrow D$
- $A \rightarrow B \rightarrow D \rightarrow C$'
- $A \rightarrow D \rightarrow B \rightarrow C$

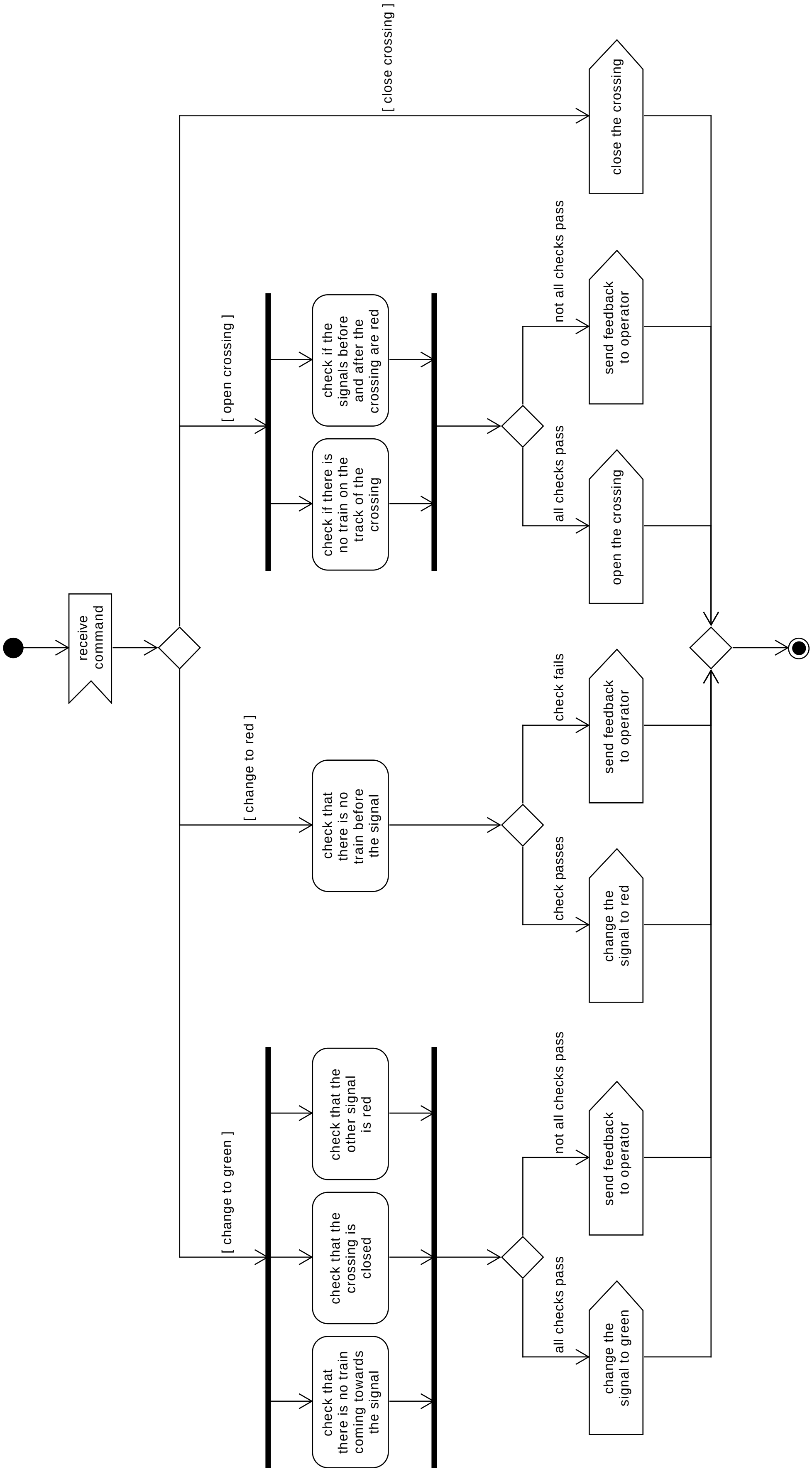
Solution to Exercise 8.10 According to the description, the supervisor decides to change a signal to green, the controller should make sure that:

1. there is no train coming in the opposite direction towards the signal
2. the signal in the opposite direction is red
3. the crossing is closed

Similar checks have to be performed for changing the signal to other colors. Each time, we carry out the checks in parallel, since there is no order dependency.

We then get the following activity diagram. (Rotated for readability)[1]

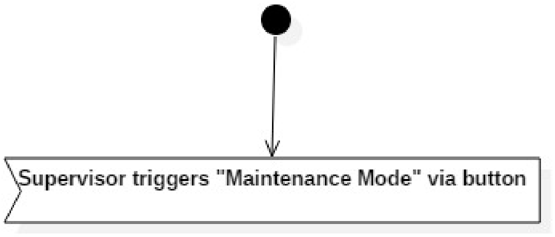
[1] This solution was provided by Jaro Reinders


receive command
[close crossing]
close the crossing
[open crossing]
check if there is no train on the track of the crossing
check if the signals before and after the crossing are red
all checks pass
open the crossing
not all checks pass
send feedback to operator
[change to red]
check that there is no train before the signal
check passes
change the signal to red
check fails
send feedback to operator
[change to green]
check that there is no train coming towards the signal
check that the crossing is closed
check that the other signal is red
all checks pass
change the signal to green
not all checks pass
send feedback to operator

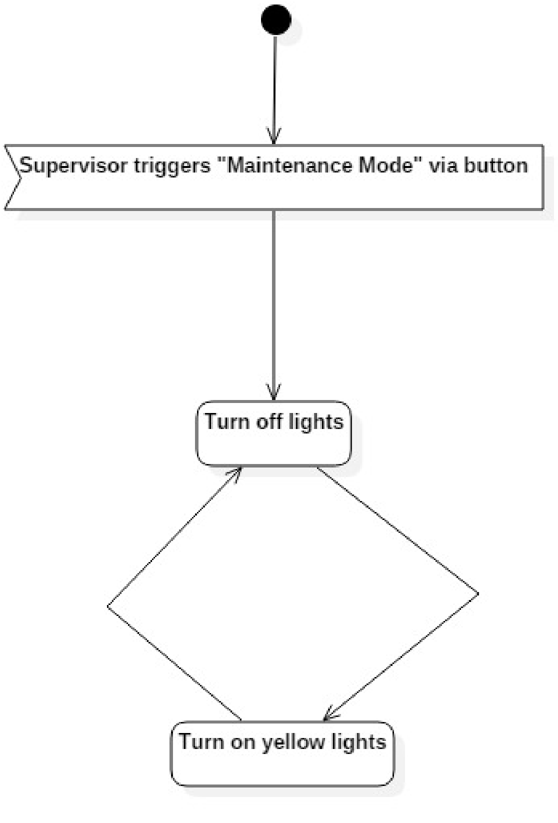
Solution to Exercise 8.11 To create an Activity Diagram we should first think about what operations (of course depending on the activity we want to represent) the system at hand can perform. Also one must be familiar with the semantics of an Activity Diagram, i.e. what do the different shapes and symbols mean.

Assuming that we are all set up and ready to go, let's decide on what activities we would like to represent using UML Activity Diagrams. Let's do a simpler example first, say when the system goes in "Emergency Mode".

As always we first need to have an initial state. However, this by itself is not enough, we also need some kind of stimulus which is going to set everything in motion for the activity diagram. But we already have this stimulus, it is when the supervisors initiates the "Emergency Mode" by lets say a button. So this is what we have so far.

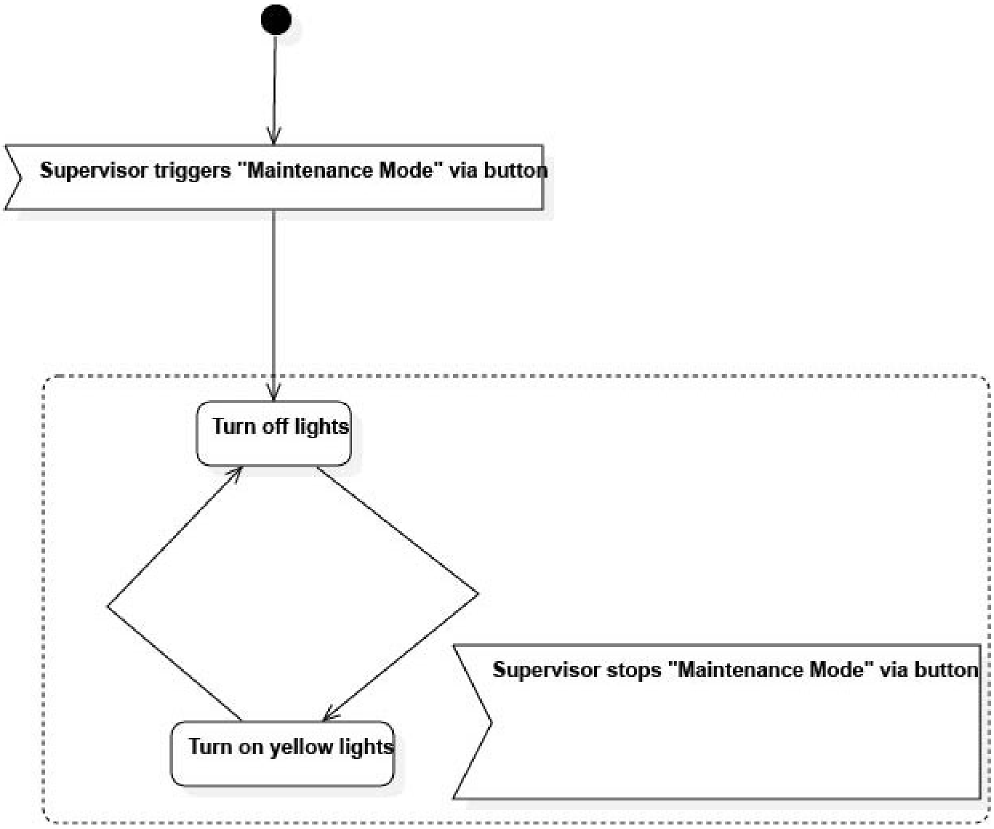


So what happens next? The system must start blinking all four yellow lights on each of the traffic lights. In the picture below we are looking at a single traffic light, but this happens simultaneously in all four traffic lights. (One can try and extend the diagram using a "Fork" and "Join" symbol.)

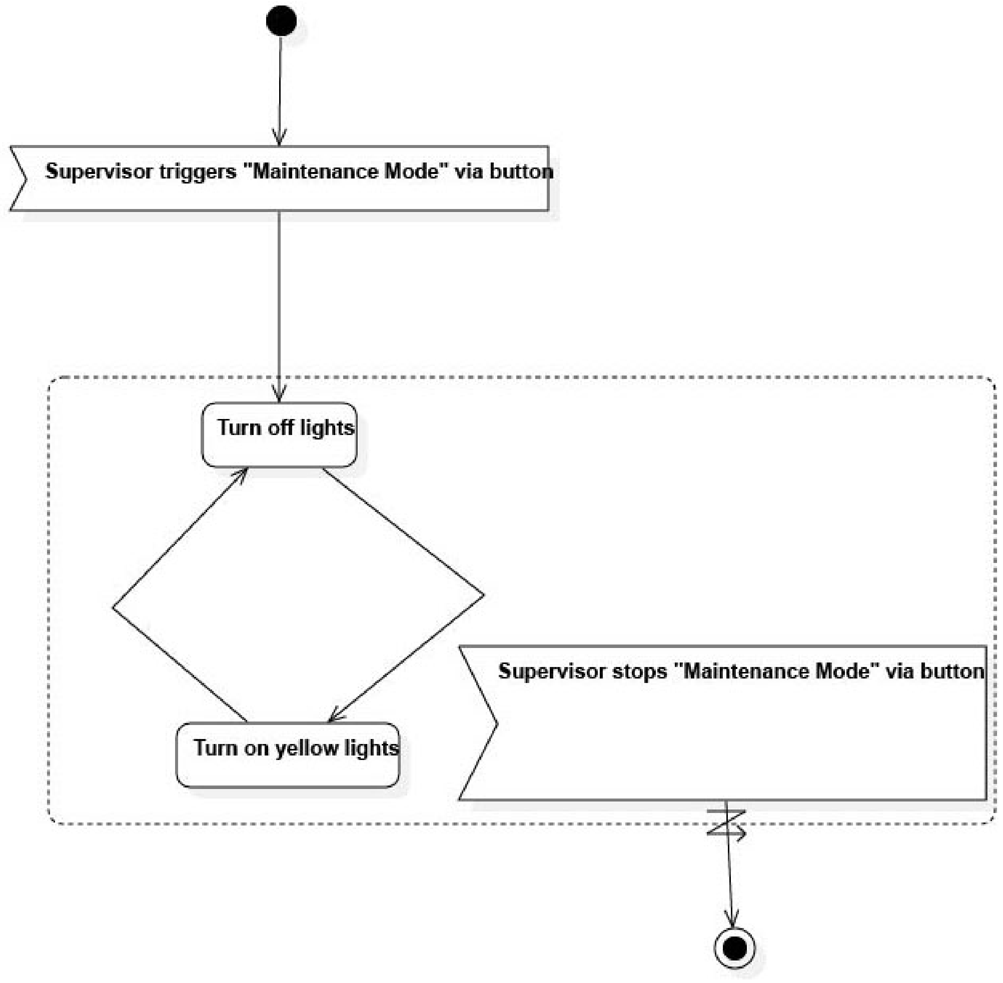


But this is not enough currently it is just going to continue on blinking forever. So we lets add a stopping rule, which lets assume is done by the supervisors with a button "End Emergency

Mode". We can use an interruptible activity region. One can also make it using a "Decision symbol" with guards. Both representations are correct, depends on how you want to model it.

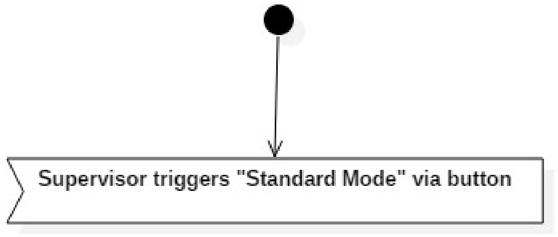

Supervisor triggers "Maintenance Mode" via button
Turn off lights
Turn on yellow lights
Supervisor stops "Maintenance Mode" via button

And to finalize we just need to add the "End symbol".

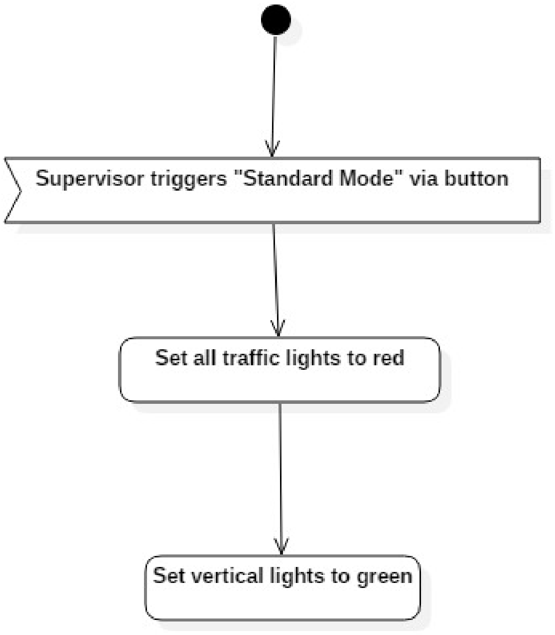

Supervisor triggers "Maintenance Mode" via button
Turn off lights
Turn on yellow lights
Supervisor stops "Maintenance Mode" via button

Now that we have done something easier, lets do something a bit more complicated. Which lets model how the system should work in ”Standard mode”. Here we are again representing the two vertical and two horizontal traffic lights as a single vertical and a single horizontal.

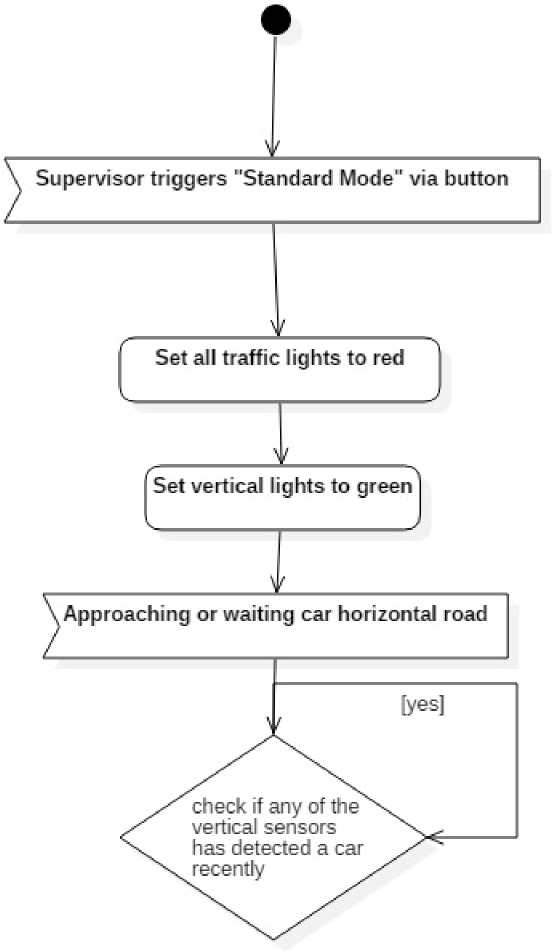
We start the same way as before. When the system is turned on, we will assume that the supervisor must turn on ”Standard mode” and when he wants to end it, he can just press a button say ”End Standard Mode”



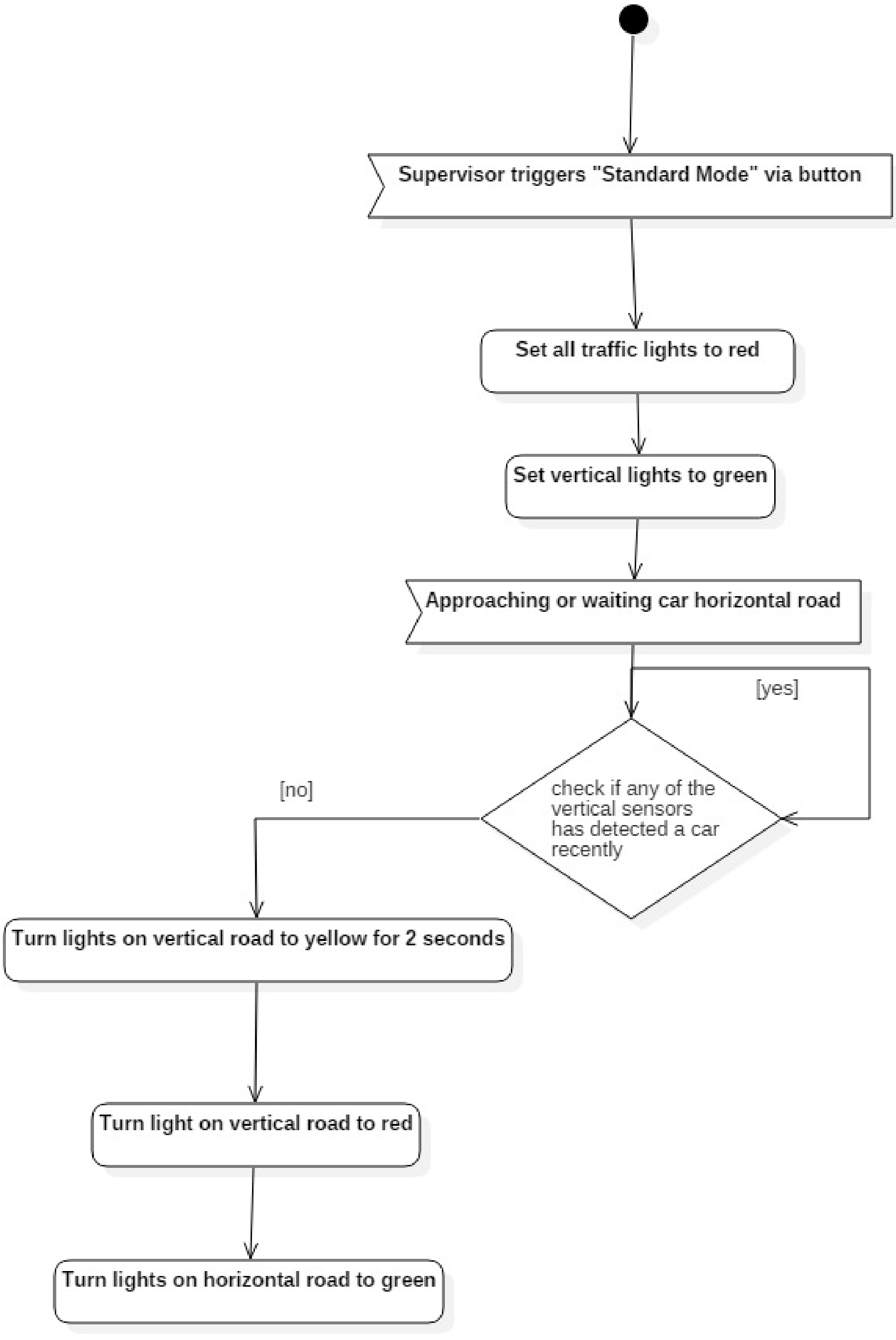
As it is described above, we know that the vertical traffic light has priority over the other, so when the system starts up, the vertical traffic lights are green and other ones are red. This can be done as follows:


Supervisor triggers "Standard Mode" via button
Set all traffic lights to red
Set vertical lights to green

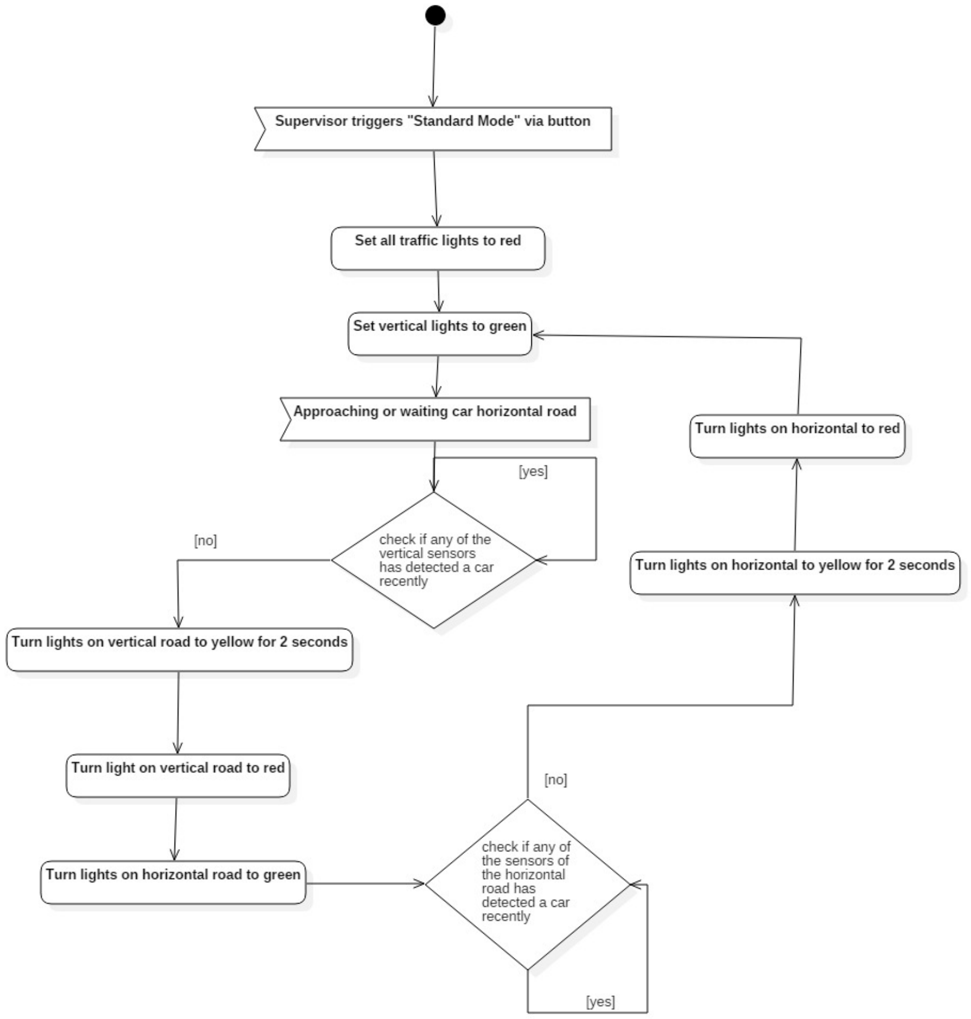
Now we want to make sure that if any of the sensors on the horizontal road detects a car, the system will keep the lights green for the vertical road until no car has passed in the last 2 seconds on the vertical road.


Supervisor triggers "Standard Mode" via button
Set all traffic lights to red
Set vertical lights to green
Approaching or waiting car horizontal road
[yes]
check if any of the vertical sensors has detected a car recently

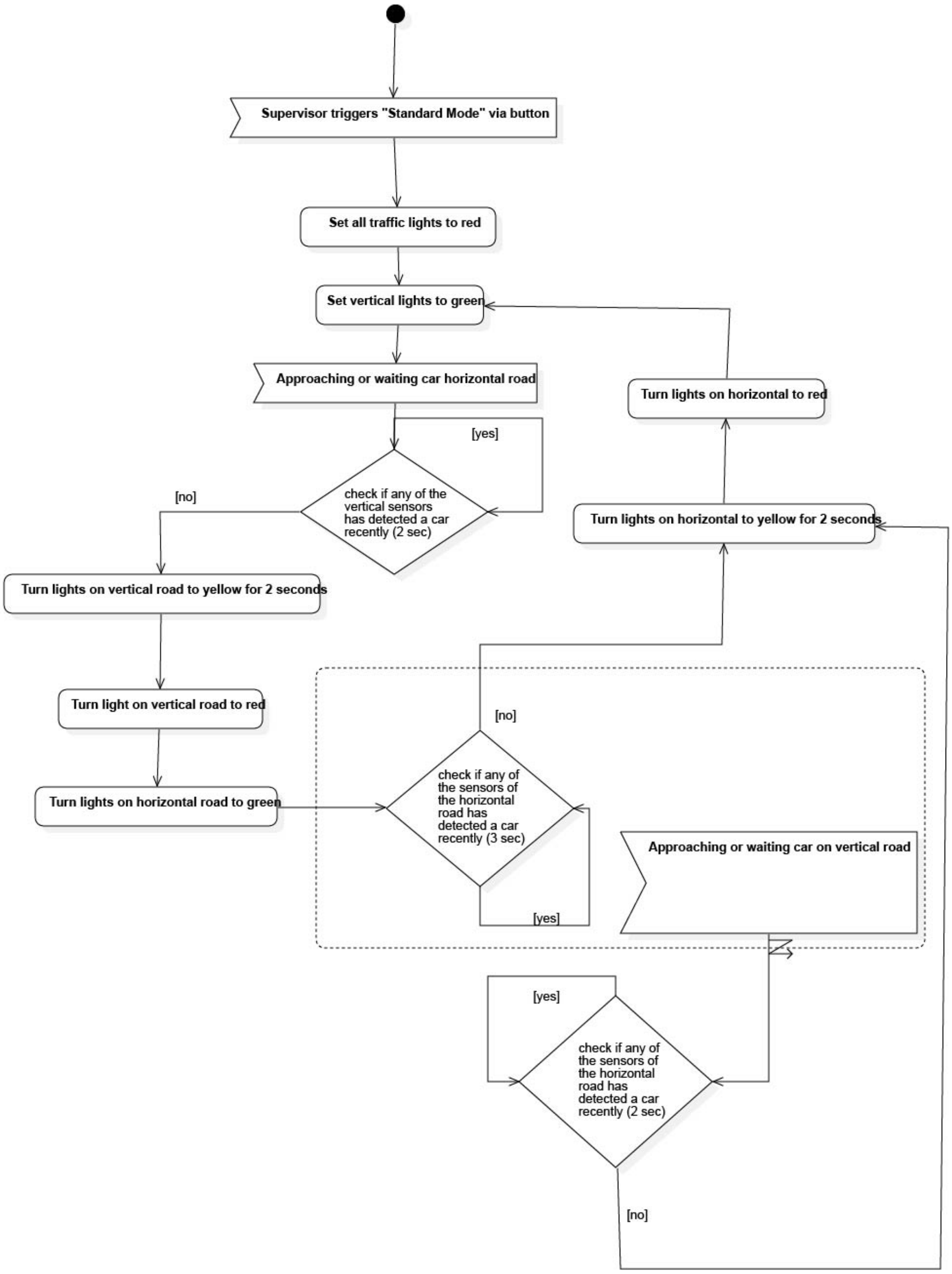
Now that this is done, if no car has passed in the last 2 seconds on the vertical road we can set the vertical lights to yellow, then red, then set the the horizontal lights to green.


Supervisor triggers "Standard Mode" via button
Set all traffic lights to red
Set vertical lights to green
Approaching or waiting car horizontal road
[yes]
[no]
check if any of the vertical sensors has detected a car recently
Turn lights on vertical road to yellow for 2 seconds
Turn light on vertical road to red
Turn lights on horizontal road to green

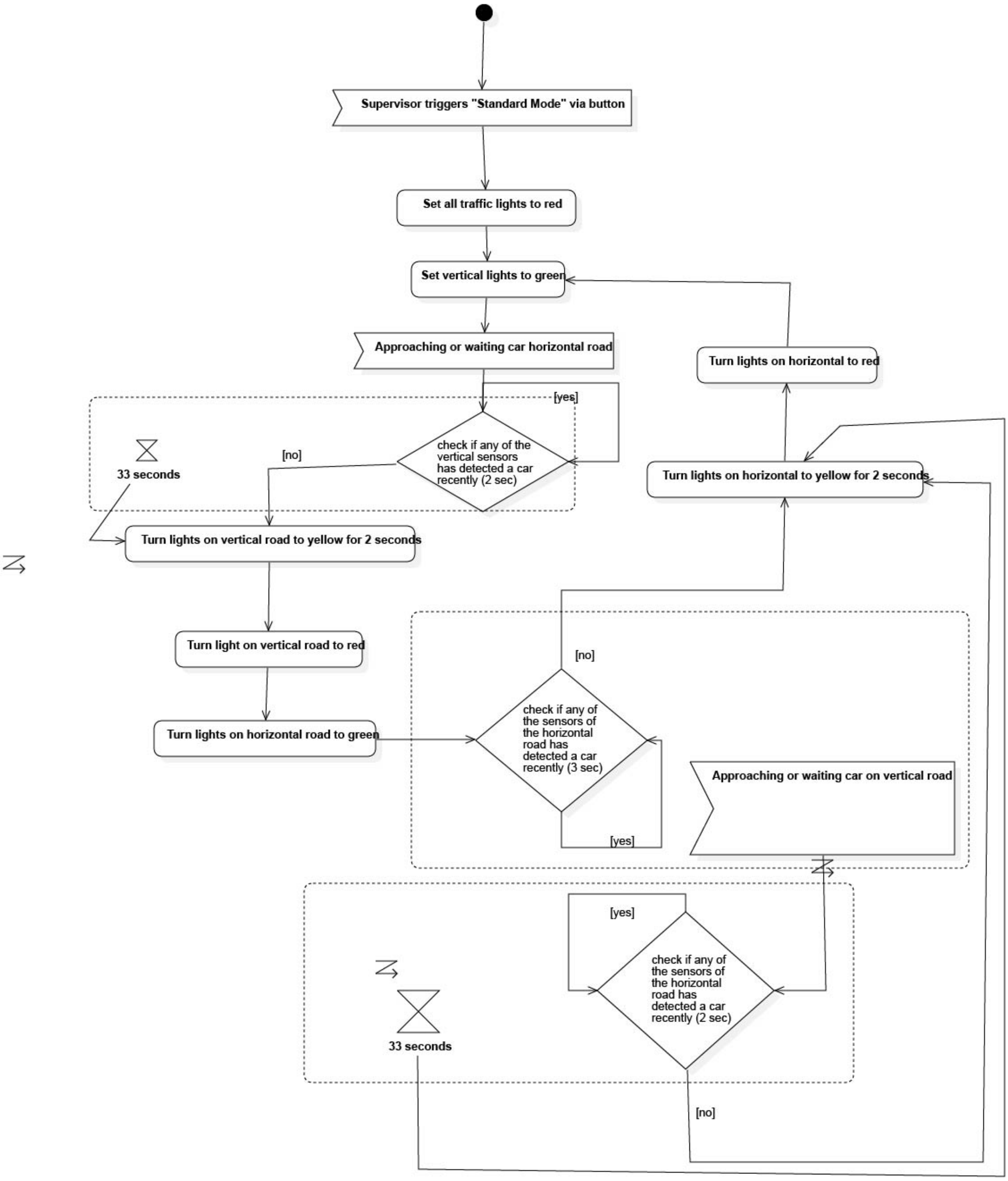
So what now, check if any of the horizontal sensors has detected a car in the last 3 second. If it has, stay in the same state, if it hasn't then set horizontal to yellow, then red, then vertical to green.


Supervisor triggers "Standard Mode" via button
Set all traffic lights to red
Set vertical lights to green
Approaching or waiting car horizontal road
[yes]
[no]
check if any of the vertical sensors has detected a car recently
Turn lights on vertical road to yellow for 2 seconds
Turn light on vertical road to red
Turn lights on horizontal road to green
[no]
check if any of the sensors of the horizontal road has detected a car recently
[yes]
Turn lights on horizontal to yellow for 2 seconds
Turn lights on horizontal to red

Okay! But this is not enough. Currently the system is missing some key point from the assumptions we have made above. Firstly while the horizontal lights are green, if there is an approaching or waiting car on the vertical road, the system does not do anything, lets change this.


Supervisor triggers "Standard Mode" via button
Set all traffic lights to red
Set vertical lights to green
Approaching or waiting car horizontal road
[yes]
[no]
check if any of the vertical sensors has detected a car recently (2 sec)
Turn lights on vertical road to yellow for 2 seconds
Turn light on vertical road to red
Turn lights on horizontal road to green
[no]
check if any of the sensors of the horizontal road has detected a car recently (3 sec)
[yes]
Approaching or waiting car on vertical road
[yes]
check if any of the sensors of the horizontal road has detected a car recently (2 sec)
[no]
Turn lights on horizontal to yellow for 2 seconds
Turn lights on horizontal to red

We're almost there! What's left. Well what about the "33 seconds assumption". Lets add it in!

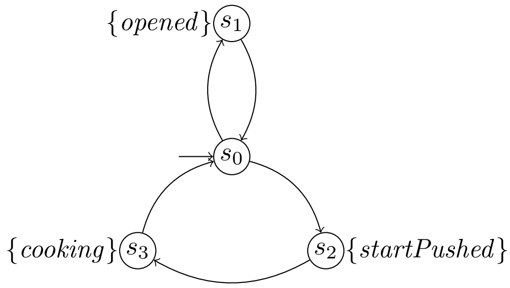

Supervisor triggers "Standard Mode" via button
Set all traffic lights to red
Set vertical lights to green
Approaching or waiting car horizontal road
[yes]
33 seconds
[no]
check if any of the vertical sensors has detected a car recently (2 sec)
Turn lights on vertical road to yellow for 2 seconds
Turn light on vertical road to red
Turn lights on horizontal road to green
Turn lights on horizontal to red
Turn lights on horizontal to yellow for 2 seconds
[no]
check if any of the sensors of the horizontal road has detected a car recently (3 sec)
[yes]
Approaching or waiting car on vertical road
[yes]
check if any of the sensors of the horizontal road has detected a car recently (2 sec)
33 seconds
[no]

And all done!

However, this task can have multiple correct solutions depending on requirements, assumptions, different interpretations etc. This was just one of many.

Solution to Exercise 9.1

1. This could, for instance, be modelled using the following Kripke structure.



The formal definition, and the labelling, are as follows.

- $S = \{s_0, s_1, s_2, s_3\}$
- $\rightarrow = \{(s_0, s_1), (s_1, s_0), (s_0, s_2), (s_2, s_3), (s_3, s_0)\}$
- $AP = \{opened, cooking, startPushed\}$
- $L(s_0) = \emptyset$
- $L(s_1) = \{opened\}$
- $L(s_2) = \{startPushed\}$
- $L(s_3) = \{cooking\}$

Solution to Exercise 9.2 We describe the Kripke structure $TTT = (S, s_0, \rightarrow, AP, L)$. For every i, j with $1 \leq i, j \leq 3$ we fix two atomic propositions $\mathsf{marked}_\times(i,j)$ and $\mathsf{marked}_\bigcirc(i,j)$ to encode which cells are marked. To encode whose turn it is we fix two atomic propositions $\mathsf{turn}_\times$ and $\mathsf{turn}_\bigcirc$. Finally we fix the propositions $\mathsf{win}_\times$, $\mathsf{win}_\bigcirc$ and draw to denote the outcome of the game:

$$AP = \{\mathsf{marked}_y(i,j) \mid y \in \{\times, \bigcirc\} \wedge 1 \leq i, j \leq 3\} \cup \{\mathsf{turn}_\times, \mathsf{turn}_\bigcirc, \mathsf{win}_\times, \mathsf{win}_\bigcirc, \mathsf{draw}\}$$

Let s_0 be the initial state. We construct the rest of the states from s_0 depending on the labelling of each state. We decide that player × starts on an empty grid: $L(s_0) = \{\mathsf{turn}_\times\}$.

Let s be an arbitrary state. We add state labels based on the current labelling $L(s)$ (the marking), and add transitions.

- If s has a winner, then we add one of the corresponding propositions $\mathsf{win}_\times$ or $\mathsf{win}_\bigcirc$ to $L(s)$. Formally we say that $y \in \{\times, \bigcirc\}$ wins s if
  - $\mathsf{marked}_y(i,1), \mathsf{marked}_y(i,2), \mathsf{marked}_y(i,3) \in L(s)$ for some i; or
  - $\mathsf{marked}_y(1,j), \mathsf{marked}_y(2,j), \mathsf{marked}_y(3,j) \in L(s)$ for some j; or
  - $\mathsf{marked}_y(1,1), \mathsf{marked}_y(2,2), \mathsf{marked}_y(3,3) \in L(s)$; or
  - $\mathsf{marked}_y(3,1), \mathsf{marked}_y(2,2), \mathsf{marked}_y(1,3) \in L(s)$.

  For all states s and all $y \in \{\times, \bigcirc\}$, whenever y wins s we fix that $\mathsf{win}_y \in L(s)$ and $s \rightarrow s$.
- If s is a full grid and has no winner, then we add draw to $L(s)$. Formally we say that s is a draw if s has no winner and for all $1 \leq i, j \leq 3$ we have $\mathsf{marked}_y(i,j) \in L(s)$ for some $y \in \{\times, \bigcirc\}$. For all states s such that s is a draw we fix $\mathsf{draw} \in L(s)$ and $s \rightarrow s$.
- Otherwise s denotes a non-draw grid with no winner and we add transitions according to the game rules. For all such states we do the following:
  - If $\mathsf{turn}_\times \in L(s)$ then for every cell i, j such that neither $\mathsf{marked}_\times(i,j)$ nor $\mathsf{marked}_\bigcirc(i,j)$ is in $L(s)$ we create a fresh state s' with $(s, s') \in \rightarrow$. For such i, j we label s' as follows:

    $$L(s') = (L(s) \cup \{\mathsf{marked}_\times(i,j), \mathsf{turn}_\bigcirc\}) \setminus \{\mathsf{turn}_\times\}\,.$$
  - If $\mathsf{turn}_\bigcirc \in L(s)$ then the construction proceeds analogously.

Note 1: One demand of a Kripke structure is that the transition relation is total. The self-loops $s \to s$ is winning states and draw states are only present to ensure this.
Note 2: This construction is naive in terms of the state space since there are many states that are labelled exactly the same. Moreover many board configurations are equivalent up to rotation and mirroring.

Solution to Exercise 10.1

1. We can model the tokens that are put on a grid as follows:

```
mtype:player = {cX, cO, cNone};
```

2. To represent the grid, we use a one-dimensional array, of which the indices correspond to the positions shown in the exercise. Initially the grid is empty. So, we declare the following global variable and immediately initialise it.

```
mtype grid[9] = {
        cNone, cNone, cNone,
        cNone, cNone, cNone,
        cNone, cNone, cNone
};
```

3. Player `p` wins with configuration `grid` if a row, column or diagonal is filled with its token. Since we identify tokens and player, we can encode this as follows.

```
#define WINS(grid, p) \
        (       (grid[0] == p && grid[1] == p && grid[2] == p) \
        ||      (grid[3] == p && grid[4] == p && grid[5] == p) \
        ||      (grid[6] == p && grid[7] == p && grid[8] == p) \
        ||      (grid[0] == p && grid[3] == p && grid[6] == p) \
        ||      (grid[1] == p && grid[4] == p && grid[7] == p) \
        ||      (grid[2] == p && grid[5] == p && grid[8] == p) \
        ||      (grid[0] == p && grid[4] == p && grid[8] == p) \
        ||      (grid[2] == p && grid[4] == p && grid[6] == p) )
```

4. We first encode that the entire grid is covered, and then check that neither of the players wins. The resulting macro is then as follows.

```
#define DRAWS(grid) \
        (       (grid[0] != cNone && grid[1] != cNone && grid[2] != cNone \
        &&      grid[3] != cNone && grid[4] != cNone && grid[5] != cNone \
        &&      grid[6] != cNone && grid[7] != cNone && grid[8] != cNone) \
        && !WINS(grid, cX)
        && !WINS(grid, cO) )
```

Solution to Exercise 10.2 We essentially use the same construction as for the matrix, and we add the initialization row-by-row.

```
typedef rows {
    mtype:player row[M];
};
rows grid[N];
rows[0].row = {cNone, cNone, cNone};
rows[1].row = {cNone, cNone, cNone};
rows[2].row = {cNone, cNone, cNone};
```

Solution to Exercise 10.3 The following process models the repeated placement of `cX` on an empty grid position. Note that we refer to the global variable `grid`.

```
active proctype PlayerX()
{
    do
    ::
        if
        :: (grid[0] == cNone) -> grid[0] = cX;
        :: (grid[1] == cNone) -> grid[1] = cX;
        :: (grid[2] == cNone) -> grid[2] = cX;
        :: (grid[3] == cNone) -> grid[3] = cX;
        :: (grid[4] == cNone) -> grid[4] = cX;
        :: (grid[5] == cNone) -> grid[5] = cX;
        :: (grid[6] == cNone) -> grid[6] = cX;
        :: (grid[7] == cNone) -> grid[7] = cX;
        :: (grid[8] == cNone) -> grid[8] = cX;
        fi
    od
}
```

Solution to Exercise 10.4 We perform the following sequence of commands:

```
spin -a peterson.pml
gcc -DSAFETY -DNOREDUCE -o pan pan.c
./pan
```

It reports the following output:

```
(Spin Version 6.5.2 -- 6 December 2019)

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 28 byte, depth reached 33, errors: 0
        58 states, stored
        59 states, matched
        117 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.003       equivalent memory usage for states (stored*(State-vector + overhead))
    0.291       actual memory usage for states
    128.000       memory used for hash table (-w24)
    0.458       memory used for DFS stack (-m10000)
    128.653       total actual memory usage


unreached in proctype P
        peterson.pml:22, state 16, "-end-"
        (1 of 16 states)

pan: elapsed time 0 seconds
```

So, no errors are reported (and no trail files are generated), and a full state space search has been performed. Hence, no deadlocks were found.

Note that it does report that, in **proctype** P the end state is not reached. The end state here is the valid end state at the closing brace in the definition of P. The reason for this is that process P does not terminate (and is not intended to terminate), so this is a harmless remark by the tool that does not indicate an error.

Solution to Exercise 10.5 Similar to the answer to Exercise 10.4.

Solution to Exercise ??

1. The model is easily obtained by adapting the model of Peterson's algorithm:

```
/* Peterson's mutual exclusion algorithm for 2 processes. */

/* shared variables */
bool flag[2];

active [2] proctype P() /* two processes; */
{
        /* Since we only have two processes, and we do not have the separate
           init process the _pid values of these processes are 0 and 1. */
        pid i = _pid;
        do :: true ->
                flag[i] = true;
                !flag[1-i];
                skip; /* Here the critical section is executed */
                flag[i] = false;
        od;
}
```

2. When executing the following commands, the output shows there is a deadlock. We show the commands and the output together.

```
spin -a mutex-deadlock.pml
gcc -DSAFETY -DNOREDUCE -o pan pan.c
./pan -i
pan:1: invalid end state (at depth 13)
pan: wrote mutex-deadlock.pml.trail

(Spin Version 6.5.2 -- 6 December 2019)
Warning: Search not completed

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 20 byte, depth reached 14, errors: 1
       15 states, stored
        2 states, matched
       17 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.001       equivalent memory usage for states (stored*(State-vector +
        overhead))
    0.292       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.458       memory used for DFS stack (-m10000)
  128.653       total actual memory usage



pan: elapsed time 0 seconds
```

This clearly indicates there is a deadlock.

3. The counterexample can be printed, resulting in:

```
spin -t -p -g mutex-deadlock.pml
1:    proc  1 (P:1) mutex-deadlock.pml:11 (state 1)   [(1)]
2:    proc  1 (P:1) mutex-deadlock.pml:12 (state 2)   [flag[i] = 1]
                flag[0] = 0
                flag[1] = 1
```

```
3:    proc  1 (P:1) mutex-deadlock.pml:13 (state 3)   [(!(flag[(1-i)]))]
4:    proc  1 (P:1) mutex-deadlock.pml:14 (state 4)   [(1)]
5:    proc  0 (P:1) mutex-deadlock.pml:11 (state 1)   [(1)]
6:    proc  1 (P:1) mutex-deadlock.pml:15 (state 5)   [flag[i] = 0]
                flag[0] = 0
                flag[1] = 0
7:    proc  1 (P:1) mutex-deadlock.pml:11 (state 1)   [(1)]
8:    proc  1 (P:1) mutex-deadlock.pml:12 (state 2)   [flag[i] = 1]
                flag[0] = 0
                flag[1] = 1
9:    proc  1 (P:1) mutex-deadlock.pml:13 (state 3)   [(!(flag[(1-i)]))]
10:    proc  0 (P:1) mutex-deadlock.pml:12 (state 2)   [flag[i] = 1]
                flag[0] = 1
                flag[1] = 1
11:    proc  1 (P:1) mutex-deadlock.pml:14 (state 4)   [(1)]
12:    proc  1 (P:1) mutex-deadlock.pml:15 (state 5)   [flag[i] = 0]
                flag[0] = 1
                flag[1] = 0
13:    proc  1 (P:1) mutex-deadlock.pml:11 (state 1)   [(1)]
14:    proc  1 (P:1) mutex-deadlock.pml:12 (state 2)   [flag[i] = 1]
                flag[0] = 1
                flag[1] = 1
spin: trail ends after 14 steps
#processes: 2
                flag[0] = 1
                flag[1] = 1
14:    proc  1 (P:1) mutex-deadlock.pml:13 (state 3)
14:    proc  0 (P:1) mutex-deadlock.pml:13 (state 3)
2 processes created
```

Careful interpretation of the countexample shows that the deadlock appears when both processes have set their flag, and are waiting for the other process to reset its flag to **false**. Note that if we would have modelled the busy waiting loop explicitly, we would not have found a deadlock, instead, both processes would continue looping in their busy waiting loops forever.

Solution to Exercise 10.7

1. The model is as follows.

```
/* Peterson's mutual exclusion algorithm for 2 processes. */

/* shared variables */
bool flag[2];
byte ncrit;

active [2] proctype P() /* two processes; */
{
        /* Since we only have two processes, and we do not have the separate
           init process the _pid values of these processes are 0 and 1. */
        pid i = _pid;
        do :: true ->
                do
                        :: (flag[1-i]) -> skip; /* continue looping */
                        :: else -> break;                        /* stop the loop
                            */
                od;
                flag[i] = true;
                ncrit++;
                assert(ncrit == 1) /* Here the critical section is executed */
                ncrit--;
                flag[i] = false;
        od;
}
```

2. Verification using Spin can be performed as follows.

```
spin -a mutex-violation.pml
gcc -DSAFETY -DNOREDUCE -o pan pan.c
./pan
pan:1: assertion violated (ncrit==1) (at depth 16)
pan: wrote mutex-violation.pml.trail

(Spin Version 6.5.2 -- 6 December 2019)
Warning: Search not completed

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 28 byte, depth reached 27, errors: 1
       49 states, stored
       37 states, matched
       86 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.002       equivalent memory usage for states (stored*(State-vector +
        overhead))
    0.291       actual memory usage for states
  128.000       memory used for hash table (-w24)
    0.458       memory used for DFS stack (-m10000)
  128.653       total actual memory usage



pan: elapsed time 0 seconds
```

So, the output clearly shows that the assertion is violated.

3. We can print the counterexample and interpret the output.

```
spin -t -p -g mutex-violation.pml
  1:    proc  1 (P:1) mutex-violation.pml:12 (state 1)  [(1)]
  2:    proc  1 (P:1) mutex-violation.pml:15 (state 4)  [else]
  3:    proc  1 (P:1) mutex-violation.pml:17 (state 9)  [flag[i] = 1]
                flag[0] = 0
                flag[1] = 1
  4:    proc  1 (P:1) mutex-violation.pml:18 (state 10) [ncrit = (ncrit+1)]
                ncrit = 1
  5:    proc  1 (P:1) mutex-violation.pml:19 (state 11) [assert((ncrit==1))]
  6:    proc  1 (P:1) mutex-violation.pml:20 (state 12) [ncrit = (ncrit-1)]
                ncrit = 0
  7:    proc  0 (P:1) mutex-violation.pml:12 (state 1)  [(1)]
  8:    proc  1 (P:1) mutex-violation.pml:21 (state 13) [flag[i] = 0]
                flag[0] = 0
                flag[1] = 0
  9:    proc  1 (P:1) mutex-violation.pml:12 (state 1)  [(1)]
 10:    proc  1 (P:1) mutex-violation.pml:15 (state 4)  [else]
 11:    proc  0 (P:1) mutex-violation.pml:15 (state 4)  [else]
 12:    proc  1 (P:1) mutex-violation.pml:17 (state 9)  [flag[i] = 1]
                flag[0] = 0
                flag[1] = 1
 13:    proc  1 (P:1) mutex-violation.pml:18 (state 10) [ncrit = (ncrit+1)]
                ncrit = 1
 14:    proc  1 (P:1) mutex-violation.pml:19 (state 11) [assert((ncrit==1))]
 15:    proc  0 (P:1) mutex-violation.pml:17 (state 9)  [flag[i] = 1]
                flag[0] = 1
                flag[1] = 1
 16:    proc  0 (P:1) mutex-violation.pml:18 (state 10) [ncrit = (ncrit+1)]
                ncrit = 2
spin: mutex-violation.pml:19, Error: assertion violated
```

```
spin: text of failed assertion: assert((ncrit==1))
 17:    proc  0 (P:1) mutex-violation.pml:19 (state 11) [assert((ncrit==1))]
spin: trail ends after 17 steps
#processes: 2
                flag[0] = 1
                flag[1] = 1
                ncrit = 2
 17:    proc  1 (P:1) mutex-violation.pml:20 (state 12)
 17:    proc  0 (P:1) mutex-violation.pml:20 (state 12)
2 processes created
```

If we carefully follow the counterexample, in particular tracing back from the assertion violation, we can observe that first, both processes terminate the busy waiting loop, and subsequently set their flag and enter the critical section, violating the assertion.

Solution to Exercise 10.8

1. The message type can be defined as follows.

```
mtype:message = {aBeginTurn, aEndTurn};
```

2. The synchronous channels for both players can be defined as follows.

```
chan comPlayerX = [0] of { mtype:message };
chan comPlayerO = [0] of { mtype:message };
```

3. We extend the process for player × as follows.

```
active proctype PlayerX()
{
        do
        ::
                comPlayerX?aBeginTurn;
                if
                :: (grid[0] == cNone) -> grid[0] = cX;
                :: (grid[1] == cNone) -> grid[1] = cX;
                :: (grid[2] == cNone) -> grid[2] = cX;
                :: (grid[3] == cNone) -> grid[3] = cX;
                :: (grid[4] == cNone) -> grid[4] = cX;
                :: (grid[5] == cNone) -> grid[5] = cX;
                :: (grid[6] == cNone) -> grid[6] = cX;
                :: (grid[7] == cNone) -> grid[7] = cX;
                :: (grid[8] == cNone) -> grid[8] = cX;
                fi
                comPlayerX!aEndTurn;
        od
}
```

So, we start the turn by receiving `aBeginTurn` along channel `comPlayerX`, and end the turn by sending `aEndTurn` along the same channel.

4. The process for player ○ is completely analogous to that of player ×.

```
active proctype PlayerO()
{
        do
        ::
                comPlayerO?aBeginTurn;
                if
                :: (grid[0] == cNone) -> grid[0] = cO;
                :: (grid[1] == cNone) -> grid[1] = cO;
                :: (grid[2] == cNone) -> grid[2] = cO;
                :: (grid[3] == cNone) -> grid[3] = cO;
                :: (grid[4] == cNone) -> grid[4] = cO;
                :: (grid[5] == cNone) -> grid[5] = cO;
                :: (grid[6] == cNone) -> grid[6] = cO;
                :: (grid[7] == cNone) -> grid[7] = cO;
                :: (grid[8] == cNone) -> grid[8] = cO;
                fi
                comPlayerO!aEndTurn;
        od;
}
```

5. We model a process `Game()` with local variables `turn` and `won`. Then, we iterate. In each iteration, we first check if either of the players wins, if so, we indicate that no player has the turn any more by setting `turn = cNone`, we set `won` to the winner, and we **`break`** from the loop. If it is a draw, we also indicate no player has the turn, and set `won = cNone` to indicate there is no winner.
   If the game has not ended yet, we look at the `turn` variable, and send the turn to the appropriate player, wait for the player to end the turn, and then set the turn to the other player.
   The entire process then becomes the following.

```
active proctype Game()
{
        mtype turn = cX;
        mtype won = cNone;

        do
        :: (WINS(grid, cX)) ->
                turn = cNone;
                won = cX;
                printf("Player␣X␣wins")
                break;

        :: (WINS(grid, cO)) ->
                turn = cNone;
                won = cO;
                printf("Player␣Y␣wins")
                break;

        :: (DRAWS(grid)) ->
                turn = cNone;
                won = cNone;
                printf("It's␣a␣draw")
                break;

        :: else ->
                if
                :: (turn == cX) ->
                        comPlayerX!aBeginTurn;
                        comPlayerX?aEndTurn;
                        turn = cO;

                :: (turn == cO) ->
                        comPlayerO!aBeginTurn;
                        comPlayerO?aEndTurn;
                        turn = cX;
                fi
        od
}
```

Solution to Exercise 10.9

1. Spin reports an `pan:1: invalid end state (at depth 66)`. What this error message means is that we are effectively in a deadlock state, where not all processes are at the end of the body of their definition (at the closing `}`). We can indicate to spin that this is ok, by considering the beginning of the loop in each of the player processes a correct end state as well. For this, prepend the **`do`** lines in both player processes with `end:`

If we make this change and reverify, the tool reports no errors.

2. We can add the following assertion to the `Game()` process, after the **`od`**.

```
assert(WINS(grid,cX) || WINS(grid,cO));
```

If we then verify the property using Spin, according to the same commands provided before, the tool reports `pan:1: assertion violated`, with the assertion that is violated. It also shows exactly the grid that violates the assertion upon termination. If we study the grid, we see that indeed, the grid is full but there is no winner.

3. Since there can be no winner, we need to account for a draw as well at the end of execution, so we need to change the assertion to the weaker

```
assert(WINS(grid,cX) || WINS(grid,cO) || DRAWS(grid));
```

If we make this change and reverify, the tool reports no errors. Of course, with our knowledge of tic-tac-toe this is exactly what we expected.

Solution to Exercise 10.10 We explain the steps that need to be taken in Spin. We assume the model of the alternating bit protocol is in `abp-unreliable.pml`.

1. If we assume the analyzer was generated following the example in the previous section, the output reports no errors, so there are no deadlock states, and no assertion failures.
Furthermore, the tool will report that there are unreachable states `-end-` in each of the processes. This is to be expected: we model a protocol that keeps running indefinitely, so it is undesirable for the processes to terminate, and we can, in fact, be happy that no end states are reached. In other words, the unreachable end states that are reported are nothing to worry about.

2. The tool reports an assertion violation `pan:1: assertion violated (m == ((last_m+1)%2) )(at depth 22)`. So, we accept a message that we did not expect to accept.
We can give a sequence diagram-like representation as follows.

```
spin -t -c abp-unreliable.pml
proc 0 = :init:
proc 1 = Sender
proc 2 = Receiver
proc 3 = unreliable_channel_bit
proc 4 = unreliable_channel_data
q\p   0   1   2   3   4
  2   .   out!MSG,0,0
  2   .   .   .   .   in?MSG,0,0
  3   .   .   .   .   out!MSG,0,0
  3   .   .   in?MSG,0,0
  4   .   .   out!ACK,0
  4   .   .   .   in?ACK,0
  1   .   .   .   out!ACK,0
  1   .   in?ACK,0
  2   .   out!MSG,1,1
  2   .   .   .   .   in?MSG,1,1
  3   .   .   .   .   out!MSG,1,1
  3   .   .   in?MSG,1,1
  4   .   .   out!ACK,1
              ACCEPT 1
spin: abp-unreliable.pml:46, Error: assertion violated
spin: text of failed assertion: assert((m==((last_m+1)%2)))
spin: trail ends after 21 steps
-------------
final state:
-------------
#processes: 5
        queue 2 (fromS):
        queue 3 (toR):
        queue 4 (fromR): [ACK,1]
        queue 1 (toS):
 21:        proc  4 (unreliable_channel_data:1) abp-unreliable.pml:71 (state
     6)
 21:        proc  3 (unreliable_channel_bit:1) abp-unreliable.pml:58 (state 6)
 21:        proc  2 (Receiver:1) abp-unreliable.pml:38 (state 14)
 21:        proc  1 (Sender:1) abp-unreliable.pml:17 (state 13)
 21:        proc  0 (:init::1) abp-unreliable.pml:85 (state 5) <valid end
     state>
5 processes created
```

This gives us the processes and the messages they send and receive. In this case, for instance, the first message that is sent is `out!MSG,0,0` by the `Sender` process. Using this information, we can try to understand the execution that leads to a failure, and debug the model.

If we analyze this counterexample, the first accepted message is 1, but the first sent message is 0, so what happened to the first message?

The problem now is that, in the first round, `last_recvbit == recvbit` in the receiver, so the receiver will believe this is an old message, and does not update the successfully received message, and does not accept the message, but it sends the acknowledgement that the sender expects. So the next message the sender sends is message 1, which is then accepted by the receiver. However, since the first message that should be accepted is 0, the assertion fails.

Solution to Exercise 11.1

1. $\mathsf{F}\,\mathit{1coin}$
2. $\mathsf{G}\,\mathsf{F}\,\mathit{servingTea}$
3. $\mathsf{G}\,\mathsf{F}(\mathit{servingTea} \vee \mathit{servingCoffee})$
4. $\mathsf{G}(\mathit{2coins} \rightarrow (\mathit{2coins}\,\mathsf{U}\,\mathit{servingCoffee}))$
5. $\mathsf{G}(\mathit{1coin} \rightarrow (\mathit{1coin} \vee \mathit{2coins})\,\mathsf{U}(\mathit{servingCoffee} \vee \mathit{servingTea}))$

Solution to Exercise 11.2

1. $\mathsf{G}\,\neg(o \wedge \mathsf{X}\,o)$
2. $\mathsf{G}(i \rightarrow (o \vee \mathsf{X}\,o \vee \mathsf{X}\,\mathsf{X}\,o))$
3. $\mathsf{G}(i \rightarrow (o \leftrightarrow \mathsf{X}\,o))$
4. $\mathsf{G}\,\mathsf{F}\,o$

Solution to Exercise 11.3 The following formula is a solution:

$$\mathsf{G}(\mathit{green} \rightarrow \mathsf{X}(\mathit{green}\,\mathsf{U}(\mathit{yellow} \wedge \mathsf{X}(\mathit{yellow}\,\mathsf{U}\,\mathit{red}))))$$

Solution to Exercise 11.4

1. $\mathsf{X}\,a$: $\{s_0, s_1, s_2, s_3, s_4\}$
2. $\mathsf{X}(\mathsf{X}\,a)$: $\{s_0, s_1, s_2, s_3, s_4\}$
3. $\mathsf{G}\,b$: $\emptyset$
4. $\mathsf{G}\,\mathsf{F}\,b$: $\{s_0, s_1, s_2, s_3, s_4\}$
5. $\mathsf{F}\,\mathsf{G}\,a$: $\{s_0, s_1, s_2, s_3, s_4\}$
6. $\mathsf{G}(a\,\mathsf{U}\,b)$: $\{s_1, s_2, s_3, s_4\}$
7. $\mathsf{G}(b\,\mathsf{U}\,a)$: $\{s_1, s_2, s_3, s_4\}$
8. $\mathsf{F}(b\,\mathsf{U}\,a)$: $\{s_0, s_1, s_2, s_3, s_4\}$

Solution to Exercise 11.5

1. $\varphi_1 = \mathsf{F}\,\mathsf{G}\,c$: false. One counter-example is the path $s_1(s_4 s_3)^\omega$. (Note that there more paths)
2. $\varphi_2 = \mathsf{G}\,\mathsf{F}\,c$: true. All infinite paths from s_1 have a suffix in which s_4 alternates with one of the states s_2, s_3, and s_5 where c holds. You cannot avoid c.
3. $\varphi_3 = \mathsf{X}\,\neg c \rightarrow \mathsf{X}\,\mathsf{X}\,c$: true. The property considers only the first steps of a path. Starting from s_1, the left hand side of the implication requires that the first transition is to s_4. Then, in one step we must reach a state where c holds. In all states reachable in one step from s_4 c holds, so the property is satisfied.
4. $\varphi_4 = \mathsf{G}\,a$: false. We only need a finite prefix as counter-example. For instance, $s_1 s_4$, etc.
5. $\varphi_5 = a\,\mathsf{U}\,\mathsf{G}(b \vee c)$: true. Starting from s_1: a holds in s_1. After that, $b \vee c$ holds in all the other states. Therefore, property φ_5 holds.
6. $\varphi_6 = (\mathsf{X}\,\mathsf{X}\,b)\,\mathsf{U}(b \vee c)$: false. Consider the path (prefix) $s_1 s_4 s_2$. At position 0, $b \vee c$ does not hold, so $\mathsf{X}\,\mathsf{X}\,b$ must hold, but b does not hold in s_2 and $s_1 s_4 s_2$ is a counter-example.

Solution to Exercise 11.6

1. Mutual exclusion: $\mathsf{G}\,\neg(\mathit{Peter.use} \wedge \mathit{Betsy.use})$
2. Finite time of usage: $\mathsf{G}(\mathit{Peter.use} \rightarrow \mathsf{F}\,\neg \mathit{Peter.use})$. Same for Betsy.
3. Absence of individual starvation: $\mathsf{G}(\mathit{Peter.request} \rightarrow \mathsf{F}\,\mathit{Peter.use})$
4. Absence of blocking: $\mathsf{G}\,\mathsf{F}\,\mathit{Peter.request}$

Solution to Exercise 11.7

1. At next: $\mathsf{X}((\neg\varphi\,\mathsf{U}(\varphi \wedge \psi)) \vee \mathsf{G}\,\neg\varphi)$
2. While: $(\varphi\,\mathsf{U}\,\neg\psi) \vee \mathsf{G}\,\varphi$
3. Before: $\mathsf{F}(\varphi \wedge (\mathsf{X}\,\mathsf{F}\,\psi)) \vee \mathsf{G}\,\neg\psi$

Solution to Exercise 11.8

1. $\mathsf{F}(\mathsf{win}_\times \vee \mathsf{win}_\bigcirc \vee \mathsf{draw})$
2. $\mathsf{G}(\neg(\mathsf{win}_\times \wedge \neg\mathsf{win}_\bigcirc))$
3. $\mathsf{G}((\mathsf{turn}_\times \rightarrow \neg\,\mathsf{X}\,\mathsf{turn}_\times) \wedge (\mathsf{turn}_\bigcirc \rightarrow \neg\,\mathsf{X}\,\mathsf{turn}_\bigcirc))$
4. For all $1 \leq i, j \leq 3$ we need the formula $\mathsf{G}(\neg(\mathsf{marked}_\times(i,j) \wedge \mathsf{marked}_\bigcirc(i,j)))$.
5. For all $1 \leq i, j \leq 3$ we need the formula

$$\mathsf{G}(\mathsf{marked}_\times(i,j) \rightarrow \mathsf{G}(\mathsf{marked}_\times(i,j))) \wedge \ G(\mathsf{marked}_\bigcirc(i,j) \rightarrow \mathsf{G}(\mathsf{marked}_\bigcirc(i,j)))$$

Solution to Exercise 11.9 The equivalence does not hold. A counter example is path $\pi = s_0 s_1$ such that $L(s_0) = \{a\}$ and $L(s_1) = \{c\}$. That is, proposition a holds and then proposition c holds. Formula $\psi = (a\,\mathsf{U}\,b)\,\mathsf{U}\,c$ does not hold on that path. The first part of the formula requires that if a holds then b must hold in the future. Along path π, b never holds. Therefore, ψ does not hold. By definition of the operator until, $\varphi = a\,\mathsf{U}(b\,\mathsf{U}\,c)$ holds on path π, since $b\,\mathsf{U}\,c$ holds immediately in state s_1.

Solution to Exercise 11.10

1. $\mathsf{G}\,\varphi \to \mathsf{F}\,\psi \equiv \varphi\,\mathsf{U}(\psi \vee \neg\varphi)$. We first derive:

$$\begin{aligned} \mathsf{G}\,\varphi \to \mathsf{F}\,\psi &\equiv \neg\,\mathsf{G}\,\varphi \vee \mathsf{F}\,\psi && \{\text{Def. of} \to\} \\ &\equiv \mathsf{F}\,\neg\varphi \vee \mathsf{F}\,\psi && \{\text{Du-G}\} \\ &\equiv \mathsf{F}(\neg\varphi \vee \psi) && \{\text{Di-F}\vee\} \\ &\equiv \text{true}\,\mathsf{U}(\neg\varphi \vee \psi) && \{\text{F-U}\} \end{aligned}$$

It remains to be proven that true $\mathsf{U}(\neg\varphi \vee \psi) \equiv \varphi\,\mathsf{U}(\psi \vee \neg\varphi)$. In other words, for all infinite paths π, $\pi \models$ true $\mathsf{U}(\neg\varphi \vee \psi)$ if and only if $\pi \models \varphi\,\mathsf{U}(\psi \vee \neg\varphi)$. Note that, since $\phi \implies$ true, it follows immediately from M-U1 that $\varphi\,\mathsf{U}(\psi \vee \neg\varphi) \implies$ true $\mathsf{U}(\psi \vee \neg\varphi)$.

For the other implication, fix an arbitrary infinite path π such that $\pi \models$ true $\mathsf{U}(\neg\varphi \vee \psi)$. So, there must be some $j \geq 0$ such that $\pi[j\ldots] \models \neg\varphi \vee \psi$. Consider the smallest such j. Then for all $0 \leq i < j$, $\pi[i\ldots] \not\models \neg\varphi \vee \psi$, so in particular, $\pi[i\ldots] \models \varphi$. Therefore, $\pi \models \varphi\,\mathsf{U}(\psi \vee \neg\varphi)$. □

2. $\mathsf{F}\,\mathsf{G}\,\varphi \to \mathsf{G}\,\mathsf{F}\,\psi \equiv \mathsf{G}(\varphi\,\mathsf{U}(\psi \vee \neg\varphi))$. We first prove a lemma. Although in general G does not distribute over $\vee$, it sometimes does.

Lemma. *For any two formulas p and q,* $\mathsf{G}(\mathsf{F}\,p \vee \mathsf{F}\,q) \equiv \mathsf{G}\,\mathsf{F}\,p \vee \mathsf{G}\,\mathsf{F}\,q$.

Proof. Consider the contraposition $\neg\,\mathsf{G}(\mathsf{F}\,p \vee \mathsf{F}\,q) \equiv \neg\,\mathsf{G}\,\mathsf{F}\,p \wedge \neg\,\mathsf{G}\,\mathsf{F}\,q$. We first derive the following.

$$\begin{aligned} \neg\,\mathsf{G}(\mathsf{F}\,p \vee \mathsf{F}\,q) &\equiv \neg\,\mathsf{G}\,\mathsf{F}(p \vee q) && \text{Di-F} \\ &\equiv \mathsf{F}\,\neg\,\mathsf{F}(p \vee q) && \text{Du-G} \\ &\equiv \mathsf{F}\,\mathsf{G}(\neg p \wedge \neg q) && \text{Du-F} \end{aligned}$$

So, if we apply this to the left-hand side of our proof obligation, it remains to show that $\mathsf{F}\,\mathsf{G}(\neg p \wedge \neg q) \equiv \mathsf{F}\,\mathsf{G}\,\neg p \wedge \mathsf{F}\,\mathsf{G}\,\neg q$.

We now prove both directions of the equivalence separately.

$\Longrightarrow$ Consider an arbitrary infinite path π such that $\pi \models \mathsf{F}\,\mathsf{G}(\neg p \wedge \neg q)$. So, there is some $i \geq 0$ such that for all $j \geq i$, $\pi[j\ldots] \models \neg p \wedge \neg q$. Informally, from position i onwards, $\neg q \wedge \neq q$ is true forever. It then follows immediately that for all $j \geq i$, $\pi[j\ldots] \models \neg p$, and $\pi[j\ldots] \models \neg q$. Therefore, i witnesses $\pi \models \mathsf{F}\,\mathsf{G}\,\neg p$ and $\pi \models \mathsf{F}\,\mathsf{G} \neq q$.

$\Longleftarrow$ Consider an arbitrary infinite path π such that $\pi \models \mathsf{F}\,\mathsf{G}\,\neg p \wedge \mathsf{F}\,\mathsf{G}\,\neg q$. Therefore, $\pi \models \mathsf{F}\,\mathsf{G}\,\neg p$ and $\pi \models \mathsf{F}\,\mathsf{G}\,\neg q$. So, there is some $i \geq 0$ such that for all $j \geq i$, $\pi[j\ldots] \models \neg p$, and also there is some $k \geq 0$ such that for all $j \geq k$, $\pi[j\ldots] \models \neg q$. Then for all $j \geq \max(i,k)$ it holds that $\pi[j\ldots] \models \neg qland \neq q$, and therefore, $\max(i,k)$ witnesses that $\pi \models \mathsf{F}\,\mathsf{G}(\neg p \wedge \neg q)$. □

We now prove the equivalence of the exercise by a sequence of equivalences.

$$\begin{aligned} \mathsf{F}\,\mathsf{G}\,\varphi \to \mathsf{G}\,\mathsf{F}\,\psi &\equiv \neg\,\mathsf{F}\,\mathsf{G}\,\phi \vee \mathsf{G}\,\mathsf{F}\,\psi && \{\text{Definition of} \to\} \\ &\equiv \mathsf{G}\,\mathsf{F}\,\neg\varphi \vee \mathsf{G}\,\mathsf{F}\,\psi && \{\text{Du-F}, \text{Du-G}\} \\ &\equiv \mathsf{G}(\mathsf{F}\,\neg\varphi \vee \mathsf{F}\,\psi) && \{\text{Lemma}\} \\ &\equiv \mathsf{G}(\neg\,\mathsf{G}\,\varphi \vee \mathsf{F}\,\psi) && \{\text{Du-G}\} \\ &\equiv \mathsf{G}(\mathsf{G}\,\varphi \to \mathsf{F}\,\psi) && \{\text{Definition of} \to\} \\ &\equiv \mathsf{G}(\varphi\,\mathsf{U}(\psi \vee \neg\varphi)) && \{\text{Previous exercise}\} \end{aligned}$$

□

3. $\mathsf{G}\,\mathsf{G}(\varphi \vee \neg\psi) \equiv \neg F(\neg\varphi \wedge \psi)$. The proof is as follows.

$$\begin{aligned} \mathsf{G}\,\mathsf{G}(\varphi \vee \neg\psi) &\equiv \mathsf{G}(\varphi \vee \neg\psi) && \{\text{Id-G}\} \\ &\equiv \neg\neg\,\mathsf{G}(\varphi \vee \neg\psi) && \{\text{Double negation}\} \\ &\equiv \neg\,\mathsf{F}(\neg\varphi \wedge \psi) && \{\text{Du-G}\} \end{aligned}$$

□

4. $\mathsf{F}(\varphi \wedge \psi) \equiv \mathsf{F}\,\varphi \wedge \mathsf{F}\,\psi$. This equivalence does not hold. The counter-example is as follows. Consider an infinite path $\pi = s_0 s_1 (s_2)^\omega$ such that $s_0 \models \neg(\varphi \wedge \psi)$, $s_1 \models (\varphi \wedge \neg\psi)$ and $s_2 \models (\neg\varphi \wedge \psi)$. Observe that $\pi \models \mathsf{F}\,\varphi \wedge \mathsf{F}\,\psi$, i.e., the right hand side formula holds, but $\pi \not\models \mathsf{F}(\varphi \wedge \psi)$, i.e., the left hand side formula does not hold.
5. $\mathsf{G}\,\varphi \wedge \mathsf{X}\,\mathsf{F}\,\varphi \equiv \mathsf{G}\,\varphi$. This holds. We prove both directions of the equivalence separately.

   ⇒ Fix arbitrary infinite path π such that $\pi \models \mathsf{G}\,\varphi \wedge \mathsf{X}\,\mathsf{F}\,\varphi$, from the semantics of $\wedge$ it follows immediately that $\pi \models \mathsf{G}\,\varphi$, which is what we have to prove.

   ⇐ Fix arbitrary infinite path π such that $\pi \models \mathsf{G}\,\varphi$. We need to prove that We need to derive $\mathsf{X}\,\mathsf{F}\,\varphi$ from $\mathsf{G}\,\varphi$. By definition of $\mathsf{G}\,\varphi$ we know that $\pi \models \mathsf{G}\,\varphi \wedge \mathsf{X}\,\mathsf{F}\,\varphi$. For this, we have to show that $\pi \models \mathsf{G}\,\varphi$, which follows immediately from the assumption, and that $\pi \models \mathsf{X}\,\mathsf{F}\,\varphi$. Since $\pi \models \mathsf{G}\,\varphi$, for all $i \geq 0$, $\pi[i \ldots] \models \varphi$. Since $\pi[1 \ldots] \models \varphi$, $\pi[1 \ldots] \models \mathsf{F}\,\varphi$, hence, as $\pi = \pi[0 \ldots]$, $\pi \models \mathsf{X}\,\mathsf{F}\,\varphi$. □

6. $\mathsf{F}\,\varphi \wedge \mathsf{X}\,\mathsf{G}\,\varphi \equiv \mathsf{F}\,\varphi$. This does not hold. The counter-example is as follows. Consider the path $\pi = s_0 (s_1)^\omega$ such that $s_0 \models \varphi$ and $s_1 \models \neg\varphi$. Note that $\pi \models \mathsf{F}\,\varphi$ (since $\pi[0 \ldots] \models \varphi$. However, since for all $i > 0$, $\pi[i \ldots] \not\models \varphi$, $\pi[1 \ldots] \not\models \mathsf{G}\,\varphi$, hence $\pi \not\models \mathsf{F}\,\varphi \wedge \mathsf{X}\,\mathsf{G}\,\varphi$, so the equivalence does not hold.
7. $\mathsf{G}\,\mathsf{F}\,\varphi \to \mathsf{G}\,\mathsf{F}\,\psi \equiv \mathsf{G}(\varphi \to \mathsf{F}\,\psi)$. This does not hold. Consider the path $\pi = s_0 (s_1)^\omega$ such that $s_0 \models \varphi \wedge \neg\psi$ and $s_1 \models \neg\varphi \wedge \neg\psi$. First, observe that $\pi \not\models \mathsf{G}\,\mathsf{F}\,\varphi$, as, after the first step, φ will never hold. Hence, according to the semantics of $\to$, $\pi \models \mathsf{G}\,\mathsf{F}\,\varphi \to \mathsf{G}\,\mathsf{F}\,\psi$. However, $\pi[0 \ldots] \models \varphi$, and for all $i \geq 0$, $\pi[i \ldots] \not\models \psi$, so for all $i \geq 0$, $\pi[i \ldots] \not\models \mathsf{F}\,\psi$. Therefore, $\pi \not\models \mathsf{G}(\varphi \to \mathsf{F}\,\psi)$. Hence, the formulas are not equivalent.
8. $\mathsf{X}\,\mathsf{F}\,\varphi \equiv \mathsf{F}\,\mathsf{X}\,\varphi$. This equivalence holds. We prove this as follows.

$$\begin{aligned}
\mathsf{X}\,\mathsf{F}\,\varphi &\equiv \mathsf{X}(\text{true}\,\mathsf{U}\,\varphi) && \{\text{F-U}\}\\
&\equiv \mathsf{X}\ \text{true}\,\mathsf{U}\,\mathsf{X}\,\varphi && \{\text{Di-XU}\}\\
&\equiv \text{true}\,\mathsf{U}\,\mathsf{X}\,\varphi && \{\text{Paths are infinite (total transition relation); all states satisfies true}\}\\
&\equiv \mathsf{F}\,\mathsf{X}\,\varphi && \{\text{F-U}\}
\end{aligned}$$

□

Solution to Exercise 11.11 The value of `last_m` is `1` initially, so property `msg_one` holds trivially. If you carefully study the counterexample of property `msg_zero` you will observe that the message that the sender sends can get lost infinitely often, so it never actually reaches the receiver.

Solution to Exercise 11.12 We can extend the model with the following LTL property:

```
ltl p1 { <> WINS(grid, cX) || <> WINS(grid, cO) || <> DRAWS(grid)) };
```

Note that to satisfy the parser of the LTL formulas, we need to slightly modify the definition of `DRAWS` to the following:

```
#define losesX (!WINS(grid, cX))
#define losesO (!WINS(grid, cO))

#define DRAWS(grid) \
    (   (grid[0] != cNone && grid[1] != cNone && grid[2] != cNone \
    &&  grid[3] != cNone && grid[4] != cNone && grid[5] != cNone \
    &&  grid[6] != cNone && grid[7] != cNone && grid[8] != cNone) \
    &&  losesX && losesO )
```

If we verify this property using

```
  spin -a abp-unreliable-ltl.pml
  gcc -DNOREDUCE -o pan pan.c
  ./pan -a
```

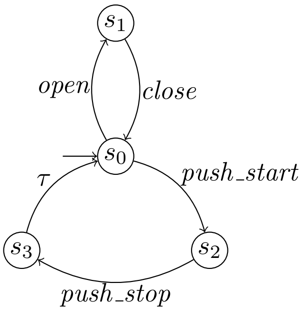
the tool will report no errors.

Note that the equivalent property

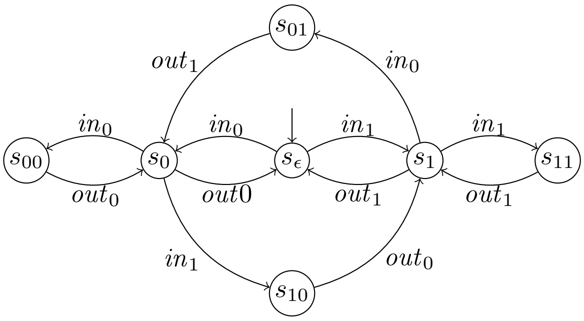
```
ltl p2 { <> (WINS(grid, cX) || WINS(grid, cO) || DRAWS(grid)) };
```

will also lead to complaints of the tool because it fails to properly transform the formula internally. This indicates some restrictions in how we can formulate requirements in Spin, that we sometimes need to workaround by using an equivalent formula.

Solution to Exercise 12.1 This could, for instance, be modelled using the following LTS.



Solution to Exercise 12.2 As mnemonic, we use state names that resemble the content of the queue, e.g., s_ϵ is the state with the empty queue, which is the initial state of the system. State s_{01} is the state in which the queue contains values 0 and 1, where 1 was the value that was inserted first. The LTS we get is then the following.



Solution to Exercise 12.3

1. Some example trances are:
   - ϵ
   - *turn_on*
   - *turn_on malfunction*
   - *malfunction repair*
   - *turn_on malfunction repair*
   - *turn_on turn_off turn_on malfunction repair*
2. As the labelled transition system contains cycles, there are infinitely many traces.

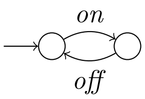
Solution to Exercise 12.4

1. *ignition_on engine_on*
2. *ignition_on engine_on ignition_off engine_off*

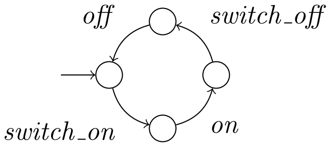
Solution to Exercise 12.5

1. $\mathit{Reach}(s_0) = \{s_0, s_1, s_2, s_3\}$
2. s **after** $b\ a = \{s_1\}$
3. s **after** $a\ a = \{s_1, s_2\}$
4. s **after** $a\ a\ b = \{s_3\}$
5. s **after** $a\ b = \emptyset$

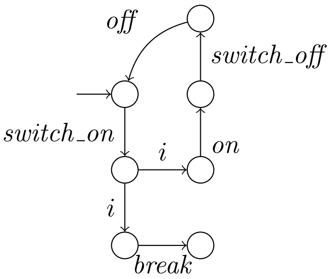
Solution to Exercise 12.6

1. 

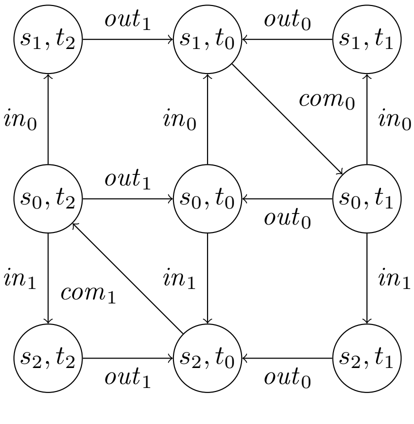
2. We change the LTS from the previous exercise such that switching on/off becomes a two-step process, as shown in the following LTS.



3. We next add the non-deterministic choice after switching on.



Solution to Exercise 12.7



Solution to Exercise 13.1

1. The transitions are:
   - $(\mathsf{P1}, \mathsf{a}, \emptyset, \emptyset, \mathsf{P2})$

- $(\mathsf{P2}, \mathsf{b}, \emptyset, \emptyset, \mathsf{Q1})$
- $(\mathsf{P2}, \mathsf{a}, \emptyset, \emptyset, \mathsf{U})$
- $(\mathsf{Q1}, \mathsf{e}, \emptyset, \emptyset, \mathsf{U})$
- $(\mathsf{U}, \mathsf{e}, \emptyset, \emptyset, \mathsf{P2})$
- $(\mathsf{U}, \mathsf{c}, \mathsf{S2}, \emptyset, \mathsf{Q1})$
- $(\mathsf{S1}, \mathsf{d}, \emptyset, \emptyset, \mathsf{S2})$
- $(\mathsf{S2}, \mathsf{f}, \emptyset, \emptyset, \mathsf{S1})$
- $(\mathsf{T1}, \mathsf{c}, \emptyset, \emptyset, \mathsf{T2})$
- $(\mathsf{T2}, \mathsf{c}, \emptyset, \emptyset, \mathsf{T1})$
- $(\mathsf{Q}, \mathsf{f}, \emptyset, \emptyset, \mathsf{P1})$

2. We calculate the scope for all transitions:

   - $\mathit{scope}((\mathsf{P1}, \mathsf{a}, \emptyset, \emptyset, \mathsf{P2})) = \mathit{lcsoa}(\{\mathsf{P1}, \mathsf{P2}\}) = \mathsf{root}$
   - $\mathit{scope}((\mathsf{P2}, \mathsf{b}, \emptyset, \emptyset, \mathsf{Q1})) = \mathit{lcsoa}(\{\mathsf{P2}, \mathsf{Q1}\}) = \mathsf{root}$
   - $\mathit{scope}((\mathsf{P2}, \mathsf{a}, \emptyset, \emptyset, \mathsf{U})) = \mathit{lcsoa}(\{\mathsf{P2}, \mathsf{U}\}) = \mathsf{root}$
   - $\mathit{scope}((\mathsf{Q1}, \mathsf{e}, \emptyset, \emptyset, \mathsf{U})) = \mathit{lcsoa}(\{\mathsf{Q1}, \mathsf{U}\}) = \mathsf{Q}$
   - $\mathit{scope}((\mathsf{U}, \mathsf{e}, \emptyset, \emptyset, \mathsf{P2})) = \mathit{lcsoa}(\{\mathsf{U}, \mathsf{P2}\}) = \mathsf{root}$
   - $\mathit{scope}((\mathsf{U}, \mathsf{c}, \mathsf{S2}, \emptyset, \mathsf{Q1})) = \mathit{lcsoa}(\{\mathsf{U}, \mathsf{Q1}\}) = \mathsf{Q}$
   - $\mathit{scope}((\mathsf{S1}, \mathsf{d}, \emptyset, \emptyset, \mathsf{S2})) = \mathit{lcsoa}(\{\mathsf{S1}, \mathsf{S2}\}) = \mathsf{S}$
   - $\mathit{scope}((\mathsf{S2}, \mathsf{f}, \emptyset, \emptyset, \mathsf{S1})) = \mathit{lcsoa}(\{\mathsf{S2}, \mathsf{S1}\}) = \mathsf{S}$
   - $\mathit{scope}((\mathsf{T1}, \mathsf{c}, \emptyset, \emptyset, \mathsf{T2})) = \mathit{lcsoa}(\{\mathsf{T1}, \mathsf{T2}\}) = \mathsf{T}$
   - $\mathit{scope}((\mathsf{T2}, \mathsf{c}, \emptyset, \emptyset, \mathsf{T1})) = \mathit{lcsoa}(\{\mathsf{T2}, \mathsf{T1}\}) = \mathsf{T}$
   - $\mathit{scope}((\mathsf{Q}, \mathsf{f}, \emptyset, \emptyset, \mathsf{P1})) = \mathit{lcsoa}(\{\mathsf{Q}, \mathsf{P1}\}) = \mathsf{root}$

3. We summarize the information in the following table. For convenience we also include the scope.

Transition	Configuration	Event	Scope	*UExt*	*Ext*	*UEnt*	*Ent*
$(\mathsf{P1}, \mathsf{a}, \emptyset, \emptyset, \mathsf{P2})$	{P1}	a	root	P1	{P1}	P2	{P2}
$(\mathsf{P2}, \mathsf{b}, \emptyset, \emptyset, \mathsf{Q1})$	{P2}	b	root	P2	{P2}	Q	{Q, Q1}
$(\mathsf{P2}, \mathsf{a}, \emptyset, \emptyset, \mathsf{U})$	{P2}	a	root	P2	{P2}	Q	{Q, U, S, T, S1, T1}
$(\mathsf{Q1}, \mathsf{e}, \emptyset, \emptyset, \mathsf{U})$	{Q, Q1}	e	Q	Q1	{Q1}	U	{U, S, T, S1, T1}
$(\mathsf{U}, \mathsf{e}, \emptyset, \emptyset, \mathsf{P2})$	{Q, U, S, T, S1, T1}	e	root	Q	{Q, U, S, T, S1, T1}	P2	{P2}
$(\mathsf{U}, \mathsf{c}, \mathsf{S2}, \emptyset, \mathsf{Q1})$	{Q, U, S, T, S2, T1}	c	Q	U	{U, S, T, S2, T1}	Q1	{Q1}
$(\mathsf{S1}, \mathsf{d}, \emptyset, \emptyset, \mathsf{S2})$	{Q, U, S, T, S1, T1}	d	S	S1	{S1}	S2	{S2}
$(\mathsf{S2}, \mathsf{f}, \emptyset, \emptyset, \mathsf{S1})$	{Q, U, S, T, S2, T1}	f	S	S2	{S2}	S1	{S1}
$(\mathsf{T1}, \mathsf{c}, \emptyset, \emptyset, \mathsf{T2})$	{Q, U, S, T, S1, T1}	c	T	T1	{T1}	T2	{T2}
$(\mathsf{T2}, \mathsf{c}, \emptyset, \emptyset, \mathsf{T1})$	{Q, U, S, T, S1, T2}	c	T	T2	{T2}	T1	{T1}
$(\mathsf{Q}, \mathsf{f}, \emptyset, \emptyset, \mathsf{P1})$	{Q, U, S, T, S1, T2}	f	root	Q	{Q, U, S, T, S1, T2}	P1	{P1}

4. In configuration $\mathit{conf} = \{\mathsf{Q}, \mathsf{U}, \mathsf{S}, \mathsf{T}, \mathsf{S2}, \mathsf{T1}\}$ the two transitions $\mathit{tr}_1 = (\mathsf{U}, \mathsf{c}, \mathsf{S2}, \emptyset, \mathsf{Q1})$ and $\mathit{tr}_2 = (\mathsf{T1}, \mathsf{c}, \emptyset, \emptyset, \mathsf{T2})$ are enabled. For tr_1 we have $\mathit{UExt}(\mathit{tr}_1) = \mathsf{U}$, and $\mathit{Ext}(\mathit{tr}_1, \mathit{conf}) = \{\mathsf{U}, \mathsf{S}, \mathsf{T}, \mathsf{S2}, \mathsf{T1}\}$. For tr_2 we have $\mathit{UExt}(\mathit{tr}_2) = \mathsf{T1}$, and $\mathit{Ext}(\mathit{tr}_1, \mathit{conf}) = \{\mathsf{T1}\}$. The exit sets overlap, so these to transitions conflict with each other.

Solution to Exercise 13.2

1. The scope of a transition is the least common strict or-ancestor of the source and the target state.

   - $\mathit{scope}((\mathsf{P1}, \mathsf{a}, \emptyset, \emptyset, \mathsf{P2})) = \mathit{lcsoa}(\{\mathsf{P1}, \mathsf{P2}\}) = \mathsf{P}$
   - $\mathit{scope}((\mathsf{Q1}, \mathsf{c}, \emptyset, \{\mathsf{d}\}, \mathsf{U})) = \mathsf{Q}$
   - $\mathit{scope}((\mathsf{S1}, \mathsf{d}, \emptyset, \{\mathsf{e}\}, \mathsf{S2})) = \mathsf{S}$
   - $\mathit{scope}((\mathsf{U}, \mathsf{e}, \mathsf{S2}, \{\mathsf{f}\}, \mathsf{P2})) = \mathsf{root}$

2. The scope of a transition is the least common or-ancestor of the source and the target state.

   - Situation $(\{\mathsf{P}, \mathsf{P1}\}, \emptyset)$ with dispatched event a. The unique exit state is $\mathsf{P1}$, the exit set is $\{\mathsf{P1}\}$, the unique enter state is $\mathsf{P2}$, the enter set is $\{\mathsf{P2}\}$.
   - Situation $(\{\mathsf{Q}, \mathsf{Q1}\}, \emptyset)$ with dispatched event c. The unique exit state is $\mathsf{Q1}$, the exit set is $\{\mathsf{Q1}\}$, the unique enter state is U, the enter set is $\{\mathsf{U}, \mathsf{S}, \mathsf{T}, \mathsf{S1}, \mathsf{T1}\}$.

- Situation $(\{Q, U, S, T, S1, T1\}, \emptyset)$ with dispatched event d. The unique exit state is S1, the exit set is $\{S1\}$, the unique enter state is S2, the enter set is $\{S2\}$.
- Situation $(\{Q, U, S, T, S2, T1\}, \emptyset)$ with dispatched event e. The unique exit state is Q, the exit set is $\{Q, U, S, T, S2, T1\}$, the unique enter state is P, the enter set is $\{P, P2\}$.

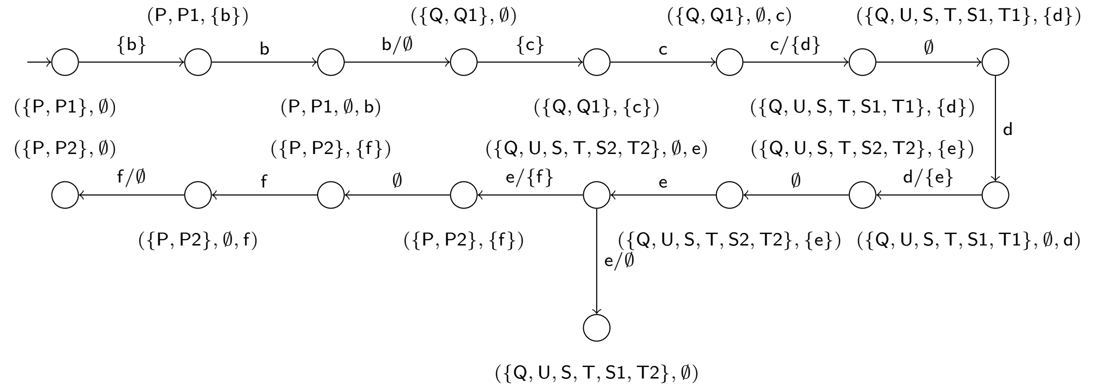
Solution to Exercise 13.3

1. For configuration $(\{P, P1\}, \{a\})$ with dispatched event b we have the following enabled transitions: $(P, b, \emptyset, \emptyset, Q)$ with exit set $\{P, P1\}$, and enter set $\{Q, Q1\}$. This transition is fireable (there are no conflicts). There is only one maximal micro-step, which consists of this single transitions.
2. For configuration $(\{P, P1\}, \{b\})$ with dispatched event a we have the following enabled transitions: $(P1, a, \emptyset, \emptyset, P2)$ with exit set $\{P1\}$, and enter set $\{P2\}$. This transition is fireable (there are no conflicts). There is only one maximal micro-step, which consists of this single transitions.
3. For configuration $(\{Q, U, S, T, S1, T1\}, \emptyset)$ with dispatched event d, we have two enabled transitions:
   - $(S1, d, \emptyset, \{e\}, S2)$ with exit set $\{S1\}$, and enter set $\{S2\}$.
   - $(T1, d, \{S1\}, \emptyset, T2)$ with exit set $\{T1\}$, and enter set $\{T2\}$.

   Both transitions are fireable, the transitions do not conflict. This results in one maximal micro-step containing both transitions.
4. For configuration $(\{Q, U, S, T, S2, T2\}, \emptyset)$ with dispatched event e we have two enabled transitions:
   - $(S2, e, \emptyset, \emptyset, S1)$ with exit set $\{S2\}$, and enter set $\{S1\}$.
   - $(U, e, \emptyset, \{f\}, P2)$ with exit set $\{Q, U, S, T, S2, T2\}$, and enter set $\{P, P2\}$.

   Since S2 is a strict descendant of U, only the first transition is fireable. Hence, this results in one maximal micro-step, which consists of the first transition.

Solution to Exercise 13.4 The environment provides triggers $\{b\}$, $\{c\}$, and subsequently nothing. The corresponding part of the LTS is the following. From the bottom three states, the execution could continue if the environment provides additional triggers. Note that the individual steps have been precomputed in the previous part of the exercise. We left out root in the configurations in the LTS for the sake of conciseness.


(P, P1, {b})
({Q, Q1}, ∅)
({Q, Q1}, ∅, c)
({Q, U, S, T, S1, T1}, {d})
{b}
b
b/∅
{c}
c
c/{d}
∅
({P, P1}, ∅)
(P, P1, ∅, b)
({Q, Q1}, {c})
({Q, U, S, T, S1, T1}, {d})
d
({P, P2}, ∅)
({P, P2}, {f})
({Q, U, S, T, S2, T2}, ∅, e)
({Q, U, S, T, S2, T2}, {e})
f/∅
f
∅
e/{f}
e
∅
d/{e}
({P, P2}, ∅, f)
({P, P2}, {f})
({Q, U, S, T, S2, T2}, {e})
({Q, U, S, T, S1, T1}, ∅, d)
e/∅
({Q, U, S, T, S1, T2}, ∅)

Solution to Exercise 14.1

1. $traces(q) = traces(q_0) = \{\epsilon, but, but \cdot liq, but \cdot choc\}$
2. $\{r_1, r_2\}$
3. $\{r_4\}$

4. $\{v_0, v_1\}$
5. $\{v_0, v_1\}$

Solution to Exercise 14.2

1. *true*, since r_2 cannot do *liq*, and does not have an outgoing τ-transition.
2. *false*
3. r_0 **after** *but* $= \{r_1, r_2\}$, since in r_1 we can do *liq*, and in r_2 we can do *but*, this is *false*.
4. This is a tricky one. Recall that v_0 **after** *but* $= \{v_0, v_1\}$. Since in v_0 it is not possible to do a τ or a *liq* transition, v_0 **refuses** $\{liq\}$, and hence $\{v_0, v_1\}$ **refuses** $\{liq\}$, so this is *true*.

Solution to Exercise 14.3 Only LTS v is an IOTS. In every reachable state $?but$ is enabled, which is the only input in L_I. All other LTSs have at least one state in which $?but$ is not enabled.

Solution to Exercise 14.4 States p_0, p_2, q_0, q_2, q_3, r_0, r_2, r_3, r_5, u_0, and v_0 are quiescent, since they do not have an output, nor τ.

Solution to Exercise 14.5 Recall that we identify $STraces(r) = STraces(r_0)$. $STraces(r) = \{\epsilon, \delta, ?but, ?but \cdot \delta, ?but \cdot !liq, ?but \cdot ?but, ?but \cdot !liq \cdot \delta, ?but \cdot ?but \cdot !choc, ?but \cdot ?but \cdot !choc \cdot \delta\}$. Note that, technically, this is not the full set of suspension traces, since each occurrence of δ can be repeated infinitely many times.

Solution to Exercise 14.6

1. $out(s_0) = \{\delta\}$
2. $out(s_1) = \{!liq\}$
3. $out(s_2) = \{!choc\}$
4. $out(s_3) = \{\delta\}$
5. $out(\{s_0, s_1\}) = \{\delta, !liq\}$
6. $out(\{s_1, s_2\}) = \{!liq, !choc\}$
7. $out(\{s_1, s_2, s_3\}) = \{\delta, !liq, !choc\}$
8. $out(\{t_0\}$ **after** $?but) = out(\{t_1, t_2, t_3\}) = \{!liq, !choc, \delta\}$
9. $out(\{t_0\}$ **after** $?but \cdot ?but) = out(\{t_1, t_3\}) = \{!liq, \delta\}$
10. $out(\{t_0\}$ **after** $?but \cdot \delta) = out(\{t_3\}) = \{\delta\}$
11. $out(\{t_0\}$ **after** $?but \cdot !choc) = out(\{t_4\}) = \{!choc\}$

Solution to Exercise 14.7 No, the implementation does not conform to the specification. For t **ioco** s to hold, we need to have $\forall \sigma \in Straces(s).out(t$ **after** $\sigma) \subseteq out(s$ **after** $\sigma)$. Now consider for instance the suspension trace $\sigma = ?but \cdot ?but$. We compute the following output sets:

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

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

Solution to Exercise 14.8 No, the implementation does not conform to the specification. Recall that i **ioco** $s = \forall \sigma \in STraces(s).out(i$ **after** $\sigma) \subseteq out(s$ **after** $\sigma)$. The trace $?but2 \cdot ?but1$ is a suspension trace of the specification, i.e., $?but2 \cdot ?but1 \in Straces(s)$. We have:

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

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

Solution to Exercise 14.9

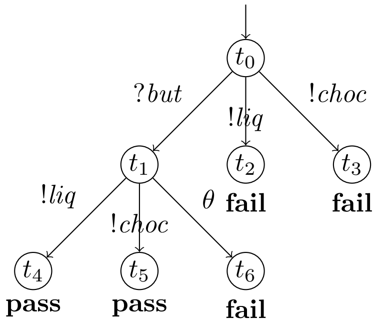
1. - p is a TTS, it satisfies all requirements for test cases.
   - q is not a TTS. From q_0 it can provide an input, but it also checks for quiescence. This combination is not allowed.
   - r is not a TTS. From r_0 it provides an input, but is does not specify what to do for any of the outputs. The latter should be specified.
   - s is not a TTS. From s_1 it cannot provide 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.
2. 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.
3. 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.

Solution to Exercise 14.10 A test run is an execution starting in $t_0 \rceil\!\lceil i_0$. We have the following executions:

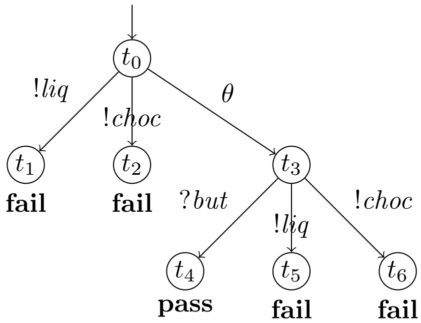
- $t_0 \rceil\!\lceil i_0 \xrightarrow{but} t_1 \rceil\!\lceil i_1 \xrightarrow{liq} t_4 = \textbf{pass} \rceil\!\lceil i_3$
- $t_0 \rceil\!\lceil i_0 \xrightarrow{but} t_1 \rceil\!\lceil i_2 \xrightarrow{\tau} t_1 \rceil\!\lceil i_3 \xrightarrow{\delta} t_6 = \textbf{fail} \rceil\!\lceil i_3$
- $t_0 \rceil\!\lceil i_0 \xrightarrow{but} t_1 \rceil\!\lceil i_2 \xrightarrow{choc} t_5 = \textbf{fail} \rceil\!\lceil i_4$

Solution to Exercise 14.11 The question implies that we need to apply 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.



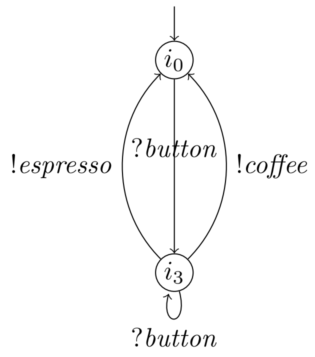
Solution to Exercise 14.12 Observe that the specification is similar to the specification in Figure 14.5. The difference is in the moment the choice is made. In order to determine the valid continuation, in the test generation algorithm we keep track of all states the specification can be in. For instance, when starting with 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 14.11 are hence also valid for this specification.

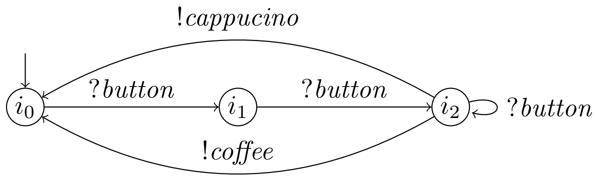
Solution to Exercise 14.13

1. - Implementation 1 is not **ioco**-conforming to the specification *spec*, since $out(spec$ **after** $?button \cdot \delta \cdot ?button) = \{!cappucino\}$, while $out(impl_1$ **after** $?button \cdot \delta \cdot ?button) = \{!cappucino, !coffee, !espresso\}$. So $\{!cappucino, !coffee, !espresso\} \not\subseteq \{!cappucino\}$.
   - Implementation 2 is not **ioco**-conforming to *spec*, since $out(impl_2$ **after** $?button \cdot ?button) = \{!cappuccino, !coffee, !espresso, \delta\}$, while $out(spec$ **after** $?button \cdot ?button) = \{!cappuccino, !coffee, !espresso\}$. So clearly $\{!cappuccino, !coffee, !espresso, \delta\} \not\subseteq \{!cappuccino, !coffee, !esp$
   - Implementation 3 is **ioco**-conforming to *spec*. Note that the graph of $impl_3$ is a sub-graph of *spec* and the quiescent states of $impl_3$ are a subset of the quiescent states of *spec*.
   - Implementation 4 is not **ioco**-conforming to *spec*, since $out(impl_4$ **after** $?button \cdot ?button) = \{!espresso, !coffee, !cappuccino, \delta\}$, while $out(spec$ **after** $?button \cdot ?button) = \{!cappuccino, !coffee, !espresso\}$. So $\{!espresso, !coffee, !cappuccino, \delta\} \not\subseteq \{!cappuccino, !coffee, !espresso\}$.
   - Implementation 5 is **ioco**-conforming to *spec*. Its graph is a sub-graph of that of *spec* and no new quiescent states are added.

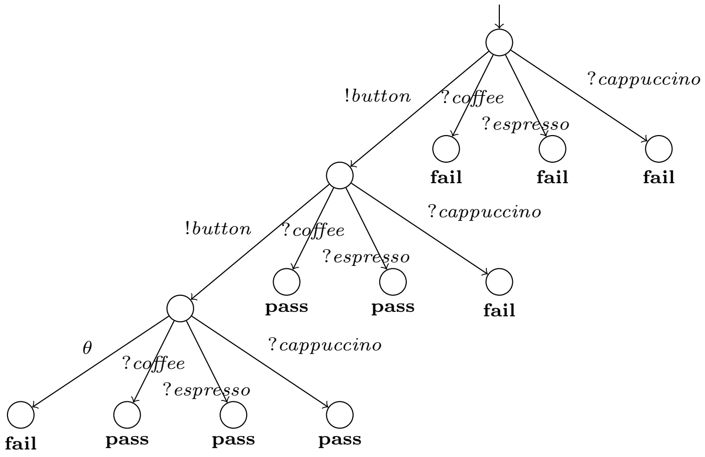
2. The following defines a sub-graph of *spec* without new quiescent states, and is **ioco**-conforming to the specification.



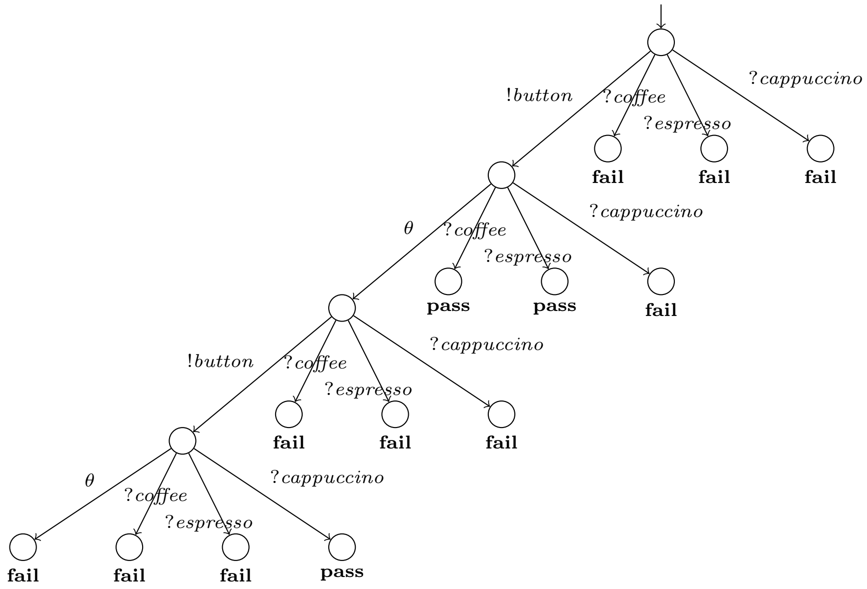
3. The following implementation is not **ioco**-conforming to the specification because after the trace $?button$, the output set of the implementation contains quiescence δ which is not in the output set of the specification after $?button$.



4. We can cover the **ioco**-incorrect implementations using the following two test cases. The first test case covers $impl_2$ and $impl_4$.



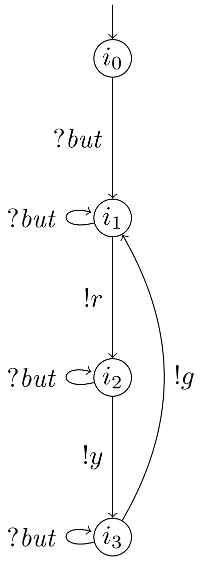
The second test case covers $impl_1$.



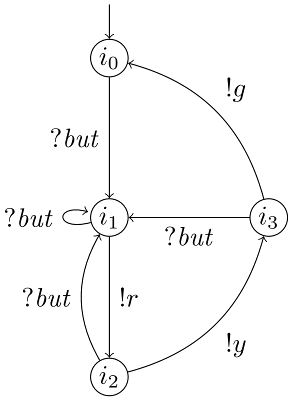
5. Test cases are *sound* w.r.t. to the **ioco**-relation, if **fail** results are only obtained in case the implementation is **not ioco**-correct.
6. The test suite composed of the two tests described previously is *sound* w.r.t. to the **ioco**-relation, since **pass** results are obtained only in case of correct **ioco**-implementations. In case of incorrect implementations, **fail** results are given.
7. *Exhaustiveness* of a test suite implies that any non-conforming implementation can be detected (which means that there are no errors that can never be detected).
8. Test suite composed of the tests we generated is not *exhaustive* (there might be non-conforming implementations, which cannot be tackled by the test suite). In our case, an *exhaustive* test suite requires infinitely many test-cases.

Solution to Exercise 14.14

1. The implementation is not **ioco**-conforming to the specification. In the specification, $out(spec\ \textbf{after}\ ?but{\cdot}!r!y) = \{!g\}$, in the implementation, $out(impl\ \textbf{after}\ ?but{\cdot}!r!y) = \{!g, \delta\}$.
2. An LTS implementing the described system is the following:



3. Such an implementation i is an **ioco**-conforming to the specification, since $out(i\ \textbf{after}\ ?but)$ is a subset of $out(spec\ \textbf{after}\ ?but)$. In a similar way, we can easily verify $out(i\ \textbf{after}\ ?but{\cdot}!r) \subseteq out(spec\ \textbf{after}\ ?but{\cdot}!r)$, $out(i\ \textbf{after}\ ?but{\cdot}!y) \subseteq out(spec\ \textbf{after}\ ?but{\cdot}!y)$, etc. An alternative way to look at this is to observe that the implementation is a subgraph of the specification (but with each state a self-loop labelled ?*but* to ensure input-enabledness); and there are no additional quiescent states.
4. The LTS described is the following.



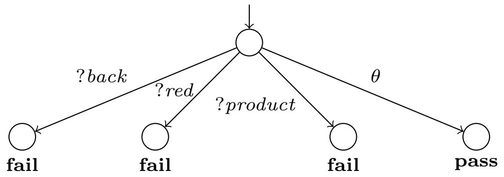
To determine whether the implementation conforms to the specification, we only need to consider the suspension traces of the *specification.* Any of the suspension traces $?but{\cdot}!r{\cdot}!y{\cdot}!g \cdots$ interspersed with ?*but* at arbitrary intermediate places, or terminated after any of the actions, are not suspension traces of the specification. We therefore do not need to check the corresponding outputs. All other suspension traces are part of the implementation, and the outputs of the implementation are allowed by the specification. The implementation therefore conforms to the specification.

Solution to Exercise 14.15

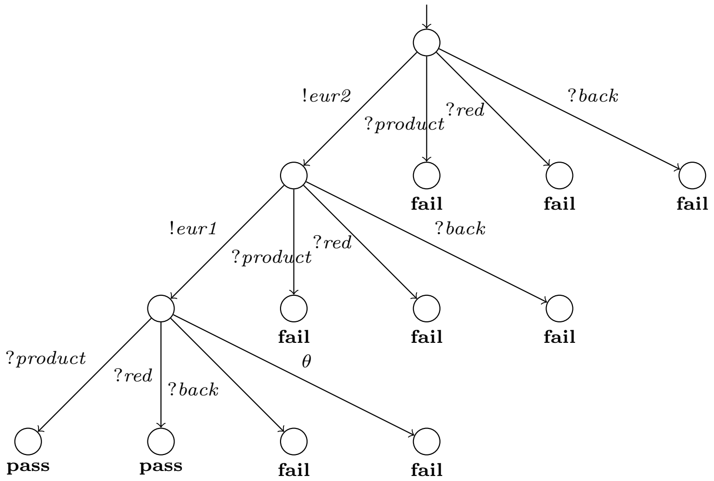
1. $\{s_0, s_1\}$
2. Only implementation j conforms to the specification. For implementation i consider the trace $?eur2 \cdot ?eur1 \cdot !red$, after this trace we will have an output of δ, no output, whereas in the specification we would have the output !*back*.

For implementation j we only have to consider traces starting with ?*eur2*, since the specification cannot start with ?*eur1* (it is underspecified in that regard). For the remaining states we do not have new quiescent states. And for the other traces we will not have new other outputs.

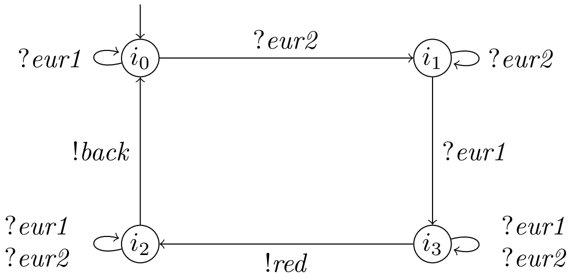
3. The test case is the following:



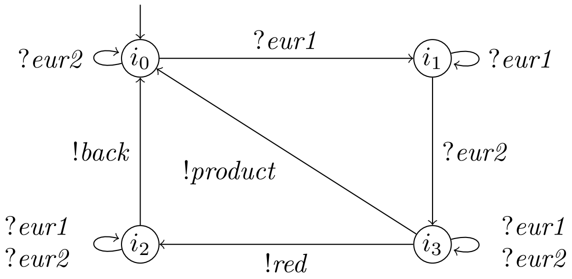
4. The test case is the following:



5. One such implementation is the following.



6. One such implementation is the following.


?eur2
i_0
?eur1
i_1
?eur1
!back
!product
?eur2
?eur1
?eur2
i_2
!red
i_3
?eur1
?eur2