Reactive Systems:
Modelling, Specification and Verification
DRAFT OF September 19, 2006

Luca Aceto\textsuperscript{1, 2}  Anna Ingólfsdóttir\textsuperscript{1, 2}  Kim G. Larsen\textsuperscript{1}
Jiri Srba\textsuperscript{1}

September 19, 2006

\textsuperscript{1}BRICS, Department of Computer Science, Aalborg University, 9220 Aalborg Ø, Denmark.
\textsuperscript{2}Department of Computer Science, School of Science and Engineering, Reykjavik University, Iceland
This draft textbook, to be published by Cambridge University Press in the first half of 2007, is intended to support the courses on Semantics and Verification (Aalborg) and Modelling and Verification (Reykjavik). It is under constant revision, and its most recent version is available at the URL


Please let us know of any comment you may have, or typographical mistake you may find, by sending an email at the addresses

luca@ru.is and srba@cs.aau.dk

with subject line ‘RS Book’.
# Contents

Preface xi

I  A Classic Theory of Reactive Systems 1

1  Introduction 3
   1.1  What are reactive systems? 4
   1.2  Process algebras 7

2  The language CCS 9
   2.1  Some CCS process constructions 9
       2.1.1  The behaviour of processes 17
   2.2  CCS, formally 20
       2.2.1  The model of labelled transition systems 20
       2.2.2  The formal syntax and semantics of CCS 25
       2.2.3  Value passing CCS 32

3  Behavioural equivalence 37
   3.1  Criteria for a good behavioural equivalence 37
   3.2  Trace equivalence: a first attempt 40
   3.3  Strong bisimilarity 42
   3.4  Weak bisimilarity 61
   3.5  Game characterization of bisimilarity 73
       3.5.1  Weak bisimulation games 79
   3.6  Further results on equivalence checking 81

4  Theory of fixed points and bisimulation equivalence 85
   4.1  Posets and complete lattices 85
   4.2  Tarski’s fixed point theorem 89
   4.3  Bisimulation as a fixed point 96
## CONTENTS

5 Hennessy-Milner logic 101

6 Hennessy-Milner logic with recursive definitions 115
  6.1 Examples of recursive properties 120
  6.2 Syntax and semantics of HML with recursion 122
  6.3 Largest fixed points and invariant properties 127
  6.4 A game characterization for HML with recursion 128
    6.4.1 Examples of use 131
  6.5 Mutually recursive equational systems 134
  6.6 Characteristic properties 138
  6.7 Mixing largest and least fixed points 148
  6.8 Further results on model checking 152

7 Modelling and analysis of mutual exclusion algorithms 155
  7.1 Specifying mutual exclusion in HML 160
  7.2 Specifying mutual exclusion using CCS itself 162
  7.3 Testing mutual exclusion 166

II A Theory of Real-time Systems 173

8 Introduction 175

9 CCS with Time Delays 177
  9.1 Intuition 177
  9.2 Timed Labelled Transition Systems 179
  9.3 Syntax and SOS rules 181
  9.4 Parallel composition 184
  9.5 Discussion 188

10 Timed Automata 191
  10.1 Motivation 191
  10.2 Syntax of Timed Automata 192
  10.3 Timed and Untimed Trace Equivalence 200
  10.4 Timed and Untimed Bisimilarity 202
  10.5 Region Graph 206
  10.6 Overview of the Main Results 216
  10.7 Networks of Timed Automata 217
CONTENTS

11 Hennessy-Milner Logic with Time
   11.1 Basic Logic .............................................. 224
   11.2 Timed Bisimilarity vs. Hennessy-Milner Logic with Time .... 233
   11.3 Adding Recursively Defined Formulae to HML with Time .... 238
   11.4 Overview of the Main Results ................................ 247

12 Modelling and Analysis of Fischer’s Algorithm
   12.1 Mutual Exclusion Using Timing ............................. 251
   12.2 Modelling Fischer’s Algorithm .............................. 252
      12.2.1 Proving Mutual Exclusion Using UPPAAL .......... 254
      12.2.2 An Erroneous Version of Fischer’s Algorithm ...... 256
   12.3 Further Exercises on Timing Based Mutual Exclusion Algorithms 258

13 Suggestions for student projects
   13.1 Alternating bit protocol ................................. 263
   13.2 Gossiping girls ........................................... 265

Bibliography ...................................................... 266

Index ............................................................... 278
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The interface for process CS</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>The interface for process CM</td>
<td>CS</td>
</tr>
<tr>
<td>2.3</td>
<td>The interface for process CM</td>
<td>CS</td>
</tr>
<tr>
<td>2.4</td>
<td>The interface for process CM</td>
<td>CS</td>
</tr>
<tr>
<td>2.5</td>
<td>The interface for process SmUni</td>
<td>CS'</td>
</tr>
<tr>
<td>2.6</td>
<td>Labelled transition system with initial state p</td>
<td>22</td>
</tr>
<tr>
<td>3.1</td>
<td>P R Q implies that C[P] R C[Q]</td>
<td>39</td>
</tr>
<tr>
<td>3.2</td>
<td>A bisimulation showing $B^2_0 \sim B^1_0</td>
<td>B^1_0$</td>
</tr>
<tr>
<td>6.1</td>
<td>Two processes</td>
<td>116</td>
</tr>
<tr>
<td>6.2</td>
<td>A process</td>
<td>123</td>
</tr>
<tr>
<td>6.3</td>
<td>The processes p and $p_i$ for $i \leq n$</td>
<td>139</td>
</tr>
<tr>
<td>6.4</td>
<td>The nice coffee machine $gkm$</td>
<td>141</td>
</tr>
<tr>
<td>6.5</td>
<td>Simple infinite process p</td>
<td>143</td>
</tr>
<tr>
<td>7.1</td>
<td>The pseudocode for Hyman’s algorithm</td>
<td>160</td>
</tr>
<tr>
<td>10.1</td>
<td>Light switch</td>
<td>191</td>
</tr>
<tr>
<td>10.2</td>
<td>Clock constraint in the guard vs. in the invariant</td>
<td>199</td>
</tr>
<tr>
<td>12.1</td>
<td>The Timed Automaton $A_i$ for Process $i$</td>
<td>252</td>
</tr>
<tr>
<td>12.2</td>
<td>Erroneous Timed Automaton $A^e_i$ for Process $i$</td>
<td>257</td>
</tr>
</tbody>
</table>
# List of Tables

2.1  An alternative formulation for process CS  . . . . . . . . . . . . . . . . 18  
2.2  SOS rules for CCS $(\alpha \in \text{Act}, a \in \mathcal{L})$  . . . . . . . . . . . 29  
3.1  The behaviour of $(\text{CM}_b \mid \text{CS}) \setminus \{\text{coin, coffee}\}$  . . . . . . 63  
3.2  The sender, receiver and medium in (3.8)  . . . . . . . . . . . . . . . 67  
9.1  SOS Rules for TCCS $(d, d' \in \mathbb{R}_{\geq 0})$  . . . . . . . . . . . . . . . . . 183  
12.1 Program for Process $i$ (Alur and Taubenfeld)  . . . . . . . . . . . . 259  
12.2 Program for Process $i$ (Revised Algorithm)  . . . . . . . . . . . . . . . 260
Preface

This book is based on courses that have been held at Aalborg University and Reykjavik University over the last five-six years. The aim of those semester-long courses was to introduce students at the early stage of their MSc. degrees, or late in their BSc. degree studies, in Computer Science to the theory of concurrency, and to its applications in the modelling and analysis of reactive systems. This is an area of formal methods that is finding increasing application outside academic circles, and allows the students to appreciate how techniques and software tools based on sound theoretical principles are very useful in the design and analysis of non-trivial reactive computing systems.

In order to carry this message across to the students in the most effective way, the courses on which the material in this book is based presented

- some of the prime models used in the theory of concurrency (with special emphasis on state-transition models of computation like labelled transition systems and timed automata),

- languages for describing actual systems and their specifications (with focus on classic algebraic process calculi like Milner’s Calculus of Communicating Systems and logics like modal and temporal logics), and

- their embodiment in tools for the automatic verification of computing systems.

The use of the theory and the associated software tools in the modelling and analysis of computing systems is a very important component in our courses since it gives the students hands-on experience in the application of what they have learned, and reinforces their belief that the theory they are studying is indeed useful and worth mastering. Once we have succeeded in awakening an interest in the theory of concurrency and its applications amongst our students, it will be more likely that at least some of them will decide to pursue a more in-depth study of the more advanced, and mathematically sophisticated, aspects of our field—for instance, during their MSc. thesis work or at a doctoral level.
It has been very satisfying for us to witness a change of attitudes in the students taking our courses over the years. Indeed, we have gone from a state in which most of the students saw very little point in taking the course on which this material is based, to one in which the relevance of the material we cover is uncontroversial to many of them! At the time when an early version of our course was elective at Aalborg University, and taken only by a few mathematically inclined individuals, one of our students remarked in his course evaluation form that ‘This course ought to be mandatory for Computer Science students.’ Now the course is mandatory, it is attended by all of the MSc. students in Computer Science at Aalborg University, and most of them happily play with the theory and tools we introduce in the course.

How did this change in attitude come about? And why do we believe that this is an important change? In order to answer these questions, it might be best to describe first the general area of Computer Science this textbook aims at contributing to.

The correctness problem and its importance  Computer scientists build artifacts (implemented in hardware, software or, as is the case in the fast-growing area of embedded and interactive systems, using a combination of both) that are supposed to offer some well defined services to their users. Since these computing systems are deployed in very large numbers, and often control crucial, if not safety critical, industrial processes, it is vital that they correctly implement the specification of their intended behaviour. The problem of ascertaining whether a computing system does indeed offer the behaviour described by its specification is called the correctness problem, and is one of the most fundamental problems in Computer Science. The field of Computer Science that studies languages for the description of (models of) computer systems and their specifications, and (possibly automated) methods for establishing the correctness of systems with respect to their specifications is called algorithmic verification.

Despite their fundamental scientific and practical importance, however, 20th century computer and communication technology has not paid sufficient attention to issues related to correctness and dependability of systems in its drive toward faster and cheaper products. (See the editorial (Patterson, 2005) by David Patterson, former president of the ACM, for forceful arguments to this effect.) As a result, system crashes are commonplace, sometimes leading to very costly, when not altogether spectacular, system failures like Intel’s Pentium-II bug in the floating-point division unit (Pratt, 1995) and the crash of the Ariane-5 rocket due to a conversion of a 64-bit real number to a 16-bit integer (Lions, n.d.).

Classic engineering disciplines have a time-honoured and effective approach to building artifacts that meet their intended specifications: before actually construct-
ing the artifacts, engineers build models of the design to be built and subject it to a thorough analysis. Surprisingly, such an approach has only recently been used extensively in the development of computing systems.

This textbook, and the courses we have given over the years based on the material it presents, stem from our deep conviction that each well educated 21st century computer scientist should be well versed in the technology of algorithmic, model-based verification. Indeed, as recent advances in algorithmic verification and applications of model checking (Clarke, Gruemberg and Peled, 1999) have shown, the tools and ideas developed within these fields can be used to analyze designs of considerable complexity that, until a few years ago, were thought to be intractable using formal analysis and modelling tools. (Companies such as AT&T, Cadence, Fujitsu, HP, IBM, Intel, Motorola, NEC, Siemens and Sun—to mention but a few—are using these tools increasingly on their own designs to reduce time to market and ensure product quality.)

We believe that the availability of automatic software tools for model-based analysis of systems is one of the two main factors behind the increasing interest amongst students and practitioners alike in model-based verification technology. Another is the realization that even small reactive systems—for instance, relatively short concurrent algorithms—exhibit very complex behaviours due to their interactive nature. Unlike in the setting of sequential software, it is therefore not hard for the students to realize that systematic and formal analysis techniques are useful, when not altogether necessary, to obtain some level of confidence in the correctness of our designs. The tool support that is now available to explore the behaviour of models of systems expressed as collections of interacting state machines of some sort makes the theory presented in this textbook very appealing for many students at several levels of their studies.

It is our firmly held belief that only by teaching the beautiful theory of concurrent systems, together with its applications and associated verification tools, to our students, we shall be able to transfer the available technology to industry, and improve the reliability of embedded software and other reactive systems. We hope that this textbook will offer a small contribution to this pedagogical endeavour.

**Why this book?** This book is by no means the first one devoted to aspects of the theory of reactive systems. Some of the books that have been published in this area over the last twenty years or so are the references (Baeten and Weijland, 1990; Fokkink, 2000; Hennessy, 1988; Hoare, 1985; Magee and Kramer, 1999; Milner, 1989; Roscoe, 1999; Schneider, 1999; Stirling, 2001) to mention but a few. However, unlike all the aforementioned books but (Fokkink, 2000; Magee and Kramer, 1999; Schneider, 1999), the present book was explicitly written to...
serve as a textbook, and offers a distinctive pedagogical approach to the material that derives from our extensive use of the material presented here in book form in the classroom. In writing this textbook we have striven to transfer on paper the spirit of the lectures on which this text is based. Our readers will find that the style in which this book is written is often colloquial, and attempts to mimic the Socratic dialogue with which we try to entice our student audience to take active part in the lectures and associated exercise sessions. Explanations of the material presented in this textbook are interspersed with questions to our readers and exercises that invite the readers to check straight away whether they understand the material as it is being presented. We believe that this makes this book suitable for self-study as well as for use as the main reference text in courses ranging from advanced BSc. courses to MSc. courses in Computer Science and related subjects.

Of course, it is not up to us to say whether we have succeeded in conveying the spirit of the lectures in the book you now hold in your hands, but we sincerely hope that our readers will experience some of the excitement that we still have in teaching our courses based on this material, and in seeing our students appreciate it, and enjoy working with concurrency theory and the tools it offers to analyze reactive systems.

For the instructor  We have used some of the material presented in this textbook in several one semester courses at Aalborg University and at Reykjavík University. These courses usually consist of about thirty hours of lectures and a similar number of hours of exercise sessions where the students solve exercises and work on projects related to the material in the course. As we already stated above, we strongly believe that these practical sessions play a very important role in making the students appreciate the importance of the theory they are learning, and understand it in depth. Examples of recent courses based on this book may be found at the following URLs:


- [http://www.cs.auc.dk/~annai/HI/SV04/plan.html](http://www.cs.auc.dk/~annai/HI/SV04/plan.html) (course given at the University of Iceland in the spring 2004 by Anna Ingólsdóttir),

- [http://www.cs.aau.dk/~srba/courses/SV-05/](http://www.cs.aau.dk/~srba/courses/SV-05/) (course given at Aalborg University in the spring 2005 by Jiri Srba) and

There the instructor will find suggested schedules for his/her courses, exercises that can be used to supplement those in the textbook, links to other useful teaching resources available on the web, further suggestions for student projects and electronic slides that can be used for the lectures. (As an example, we usually supplement lectures covering the material in this textbook with a series of four-six 45 minute lectures on Binary Decision Diagrams (Bryant, 1992) and their use in verification based on Henrik Reif Andersen’s excellent lecture notes (Andersen, n.d.) that are freely available on the web and on Randel Bryant’s survey paper (Bryant, 1992).)

We strongly recommend that the teaching of the material covered in this book be accompanied by the use of software tools for verification and validation. In our courses, we usually employ the Edinburgh Concurrency Workbench\(^1\) for the part of the course devoted to classic reactive systems, and, not surprisingly, UPPAAL\(^2\) for the lectures on real-time systems. Both of these tools are freely available, and their use makes the theoretical material covered during the lectures come alive for the students. Using the tools, the students will be able to analyze systems of considerable complexity, and we suggest that courses based upon this book be accompanied by two practical projects involving the use of these, or similar, tools for verification and validation.

We plan to maintain a page with all of the supporting material, and other useful resources for students and instructors alike, at the URL

\[\text{http://www.ru.is/MV-BOOK/}.\]

In writing this book, we have tried to be at once pedagogical, careful and precise. However, despite our efforts, we are sure that there is still room for improving this text, and correcting any mistake that may have escaped our attention. We shall use the aforementioned web page to inform the reader about additions and modifications to this book.

We welcome corrections (typographical or otherwise), comments and suggestions from our readers. You can contact us by sending an email at the addresses 

\[\text{luca@ru.is and srba@cs.aau.dk}\]

with subject line ‘RS Book’.

Acknowledgments This book was partly written while Luca Aceto was on leave from Aalborg University at Reykjavík University, Anna Ingólfsdóttir was working at deCODE Genetics, and Jiri Srba was visiting the University of Stuttgart sponsored by a grant from the Alexander von Humboldt Foundation. They thank these

\(^{1}\text{http://homepages.inf.ed.ac.uk/perdita/cwb/}\)

\(^{2}\text{http://www.uppaal.com/}\)
institutions for their hospitality and excellent working conditions. Luca Aceto and Anna Ingólfsdóttir were partly supported by the project ‘The Equational Logic of Parallel Processes’ (nr. 060013021) of The Icelandic Research Fund.

We thank Silvio Capobianco, Pierre-Louis Curien, Gudmundur Hreidarson, Rocco De Nicola, Ralph Leibmann and the students of the Concurrency Course (Concurrence) (number 2–3) 2004–2005, Master Parisien de Recherche en Informatique, for useful comments and corrections on drafts of this text.

The authors used drafts of the book in courses taught in the spring of 2004, 2005 and 2006, and in the autumn 2006, at Aalborg University, Reykjavík University and the University of Iceland. The students who took those courses offered valuable feedback on the text, and gave us detailed lists of errata. We thank Claus Brabrand for using a draft of the first part of this book in his course Semantics (Q1, 2005) at Aarhus University. The suggestions from Claus and his students helped us improve the text further. Moreover, Claus and one of his students designed and implemented an excellent CCS simulator\footnote{http://www.brics.dk/~martinm/CCSVisualizer.html} that our students can use to experiment with the behaviour of processes written in this language.

Last, but not least, we are thankful to David Tranah at Cambridge University Press for his enthusiasm for our project, and to the three anonymous reviewers that provided useful comments on a draft of this book.

Any remaining infelicity is solely our responsibility.

Luca Aceto and Anna Ingólfsdóttir dedicate this book to their son Róbert.

Luca Aceto and Anna Ingólfsdóttir, Reykjavík, Iceland
Kim G. Larsen and Jiri Srba, Aalborg, Denmark
Part I

A Classic Theory of Reactive Systems
Chapter 1

Introduction

The aim of the first part of this book is to introduce three of the basic notions that we shall use to describe, specify and analyze reactive systems, namely

- Milner’s Calculus of Communicating Systems (CCS) (Milner, 1989),
- the model of Labelled Transition Systems (LTSs) (Keller, 1976), and

We shall present a general theory of reactive systems and its applications. In particular, we intend to show how

1. to describe actual systems using terms in our chosen models (that is, either as terms in the process description language CCS or as labelled transition systems),
2. to offer specifications of the desired behaviour of systems either as terms of our models or as formulae in HML, and
3. to manipulate these descriptions, possibly (semi-)automatically, in order to analyze the behaviour of the model of the system under consideration.

In the second part of the book, we shall introduce a similar trinity of basic notions that will allow us to describe, specify and analyze real-time systems—that is, systems whose behaviour depends crucially on timing constraints. There we shall present the formalisms of timed automata (Alur and Dill, 1994) and Timed CCS (Wang, 1990; Wang, 1991a; Wang, 1991b) to describe real-time systems, the model of timed labelled transition systems and a real-time version of Hennessy-Milner Logic (Laroussinie, Larsen and Weise, 1995).
CHAPTER 1. INTRODUCTION

After having worked through the material in this book, you will be able to describe non-trivial reactive systems and their specifications using the aforementioned models, and verify the correctness of a model of a system with respect to given specifications either manually or by using automatic verification tools like the Edinburgh Concurrency Workbench (Cleaveland, Parrow and Steffen, 1993) and UPPAAL (Behrmann, David and Larsen, 2004).

Our, somewhat ambitious, aim is therefore to present a model of reactive systems that supports their design, specification and verification. Moreover, since many real-life systems are hard to analyze manually, we should like to have computer support for our verification tasks. This means that all the models and languages that we shall use in this book need to have a formal syntax and semantics. (The syntax of a language consists of the rules governing the formation of statements, whereas its semantics assigns meaning to each of the syntactically correct statements in the language.) These requirements of formality are not only necessary in order to be able to build computer tools for the analysis of systems’ descriptions, but are also fundamental in agreeing upon what the terms in our models are actually intended to describe in the first place. Moreover, as Donald Knuth once wrote:

A person does not really understand something until after teaching it to a computer, i.e. expressing it as an algorithm... An attempt to formalize things as algorithms leads to a much deeper understanding than if we simply try to comprehend things in the traditional way.

The pay-off of using formal models with an explicit formal semantics to describe our systems will therefore be the possibility of devising algorithms for the animation, simulation and verification of system models. These would be impossible to obtain if our models were specified only in an informal notation.

Now that we know what to expect from this book, it is time to get to work. We shall begin our journey through the beautiful land of Concurrency Theory by introducing a prototype description language for reactive systems and its semantics. However, before setting off on such an enterprise, we should describe in more detail what we actually mean with the term ‘reactive system’.

1.1 What are reactive systems?

The ‘standard’ view of computing systems is that, at a high level of abstraction, these may be considered as black boxes that take inputs and provide appropriate outputs. This view agrees with the description of algorithmic problems. An algorithmic problem is specified by a collection of legal inputs, and, for each legal
input, its expected output. In an imperative setting, an abstract view of a computing system may therefore be given by describing how it transforms an initial state—that is, a function from variables to their values—to a final state. This function will, in general, be partial—that is, it may be undefined for some initial states—to capture that the behaviour of a computing system may be non-terminating for some input states. For example, the effect of the program

\[ S = z \leftarrow x; x \leftarrow y; y \leftarrow z \]

is described by the function \([S]\) from states to states defined thus:

\[ [S] = \lambda s. s[x \mapsto s(y), y \mapsto s(x), z \mapsto s(x)] , \]

where the state \(s[x \mapsto s(y), y \mapsto s(x), z \mapsto s(x)]\) is the one in which the value of variable \(x\) is the value of \(y\) in state \(s\) and that of variables \(y\) and \(z\) is the value of \(x\) in state \(s\). The values of all of the other variables are those they had in state \(s\). This state transformation is a way of formally describing that the intended effect of \(S\) is essentially to swap the values of the variables \(x\) and \(y\).

On the other hand, the effect of the program

\[ U = \textbf{while} \; \text{true} \; \textbf{do} \; \textbf{skip} \]

where we use \textbf{skip} to stand for a ‘no operation’, is described by the partial function from states to states given by

\[ [U] = \lambda s. \text{undefined} , \]

that is the always undefined function. This captures the fact that the computation of \(U\) never produces a result (final state) irrespective of the initial state.

In this view of computing systems, non-termination is a highly undesirable phenomenon. An algorithm that fails to terminate on some inputs is not one the users of a computing system would expect to have to use. A moment of reflection, however, should make us realize that we already use many computing systems whose behaviour cannot be readily described as a function from inputs to outputs—not least because, at some level of abstraction, these systems are inherently meant to be non-terminating. Examples of such computing systems are

- operating systems,
- communication protocols,
- control programs, and
• software running in embedded system devices like mobile telephones.

At a high level of abstraction, the behaviour of a control program can be seen to be governed by the following pseudo-code algorithm skeleton.

```
loop
    read the sensors’ values at regular intervals
    depending on the sensors’ values trigger the relevant actuators
forever
```

The aforementioned examples, and many others, are examples of computing systems that interact with their environment by exchanging information with it. Like the neurons in a human brain, these systems react to stimuli from their computing environment (in the example control program above these are variations in the values of the sensors) by possibly changing their state or mode of computation, and in turn influence their environment by sending back some signals to it, or initiating some operations whose effect it is to affect the computing environment (this is the role played by the actuators in the example control program). David Harel and Amir Pnueli coined the term reactive system in (Harel and Pnueli, 1985) to describe a system that, like the aforementioned ones, computes by reacting to stimuli from its environment.

As the above examples and discussion indicate, reactive systems are inherently parallel systems, and a key role in their behaviour is played by communication and interaction with their computing environment. A ‘standard’ computing system can also be viewed as a reactive system in which interaction with the environment only takes place at the beginning of the computation (when inputs are fed to the computing device) and at the end (when the output is received). On the other hand, all the example systems given before maintain a continuous interaction with their environment, and we may think of both the computing system and its environment as parallel processes that communicate one with the other. In addition, as again nicely exemplified by the skeleton of a control program given above, non-termination is a desirable feature of some reactive systems. In contrast to the setting of ‘standard’ computing systems, we certainly do not expect the operating systems running on our computers or the control program monitoring a nuclear reactor to terminate!

Now that we have an idea of what reactive systems are, and of the key aspects of their behaviour, we can begin to consider what an appropriate abstract model for this class of systems should offer. In particular, such a model should allow us to describe the behaviour of collections of (possibly non-terminating) parallel processes that may compute independently and interact with one another. It should provide us with facilities for the description of well-known phenomena that appear in the presence of concurrency and are familiar to us from the world of operating
systems and parallel computation in general (e.g., deadlock, livelock, starvation and so on). Finally, in order to abstract from implementation dependent issues having to do with, e.g., scheduling policies, the chosen model should permit a clean description of non-determinism—a most useful modelling tool in Computer Science.

Our aim in the remainder of this book will be to present a general purpose theory that can be used to describe, and reason about, any collection of interacting processes. The approach we shall present will make use of a collection of models and formal techniques that is often referred to as Process Theory. The key ingredients in this approach are

- (Process) Algebra,
- Automata/labelled transition systems,
- Structural Operational Semantics, and
- Logic.

These ingredients give the foundations for the development of (semi-)automatic verification tools for reactive systems that support various formal methods for validation and verification that can be applied to the analysis of highly non-trivial computing systems. The development of these tools requires in turn advances in algorithmics, and via the increasing complexity of the analyzed designs feeds back to the theory development phase by suggesting the invention of new languages and models for the description of reactive systems.

Unlike in the setting of sequential programs, where we often kid ourselves into believing that the development of correct programs can be done without any recourse to 'formalism', it is a well-recognized fact of life that the behaviour of even very short parallel programs may be very hard to analyze and understand. Indeed, analyzing these programs requires a careful consideration of issues related to the interactions amongst their components, and even imagining all of these is often a mind-boggling task. As a result, the techniques and tools that we shall present in this book are becoming widely accepted in the academic and industrial communities that develop reactive systems.

1.2 Process algebras

The first ingredient in the approach to the theory of reactive systems presented in this book is a prototypical example of a process algebra. Process algebras are
CHAPTER 1. INTRODUCTION

prototype specification languages for reactive systems. They evolved from the insights of many outstanding researchers over the last thirty years, and a brief history of the evolution of the original ideas that led to their development may be found in (Baeten, 2004). (For an accessible, but more advanced, discussion of the role that algebra plays in process theory you may consult the survey paper (Luttik, 2006).) A crucial initial observation that is at the heart of the notion of process algebra is due to Milner, who noticed that concurrent processes have an algebraic structure. For example, once we have built two processes $P$ and $Q$, we can form a new process by combining $P$ and $Q$ sequentially or in parallel. The result of these combinations will be a new process whose behaviour depends on that of $P$ and $Q$ and on the operation that we have used to compose them. This is the first sense in which these description languages are algebraic: they consist of a collection of operations for building new process descriptions from existing ones.

Since these languages aim at specifying parallel processes that may interact with one another, a key issue that needs to be addressed is how to describe communication/interaction between processes running at the same time. Communication amounts to information exchange between a process that produces the information (the sender), and a process that consumes it (the receiver). We often think of this communication of information as taking place via some medium that connects the sender and the receiver. If we are to develop a theory of communicating systems based on this view, it looks as if we have to decide upon the communication medium used in inter-process communication. Several possible choices immediately come to mind. Processes may communicate via, e.g., (un)bounded buffers, shared variables, some unspecified ether, or the tuple spaces used by Linda-like languages (Gelernter, 1985). Which one do we choose? The answer is not at all clear, and each specific choice may in fact reduce the applicability of our language and the models that support it. A language that can properly describe processes that communicate via, say, FIFO buffers may not readily allow us to specify situations in which processes interact via shared variables, say.

The solution to this riddle is both conceptually simple and general. One of the crucial original insights of figures like Hoare and Milner is that we need not distinguish between active components like senders and receivers, and passive ones like the aforementioned kinds of communication media. All of these may be viewed as processes—that is, as systems that exhibit behaviour. All of these processes can interact via message-passing modelled as synchronized communication, which is the only basic mode of interaction. This is the key idea underlying Hoare’s Communicating Sequential Processes (CSP) (Hoare, 1978; Hoare, 1985), a highly influential proposal for a programming language for parallel programs, and Milner’s Calculus of Communicating Systems (CCS) (Milner, 1989), the paradigmatic process algebra.
Chapter 2

The language CCS

We shall now introduce the language CCS. We begin by informally presenting the process constructions allowed in this language and their semantics in Section 2.1. We then proceed to put our developments on a more formal footing in Section 2.2.

2.1 Some CCS process constructions

It is useful to begin by thinking of a CCS process as a black box. This black box may have a name that identifies it, and has a process interface. This interface describes the collection of communication ports, also referred to as channels, that the process may use to interact with other processes that reside in its environment, together with an indication of whether it uses these ports for inputting or outputting information. For example, the drawing in Figure 2.1 pictures the interface for a process whose name is CS (for Computer Scientist). This process may interact with its environment via three ports, namely coffee, coin and pub. The port coffee is used for input, whereas the ports coin and pub are used by process CS for output. In general, given a port name $a$, we use $\bar{a}$ for output on port $a$. We shall often refer to labels as coffee or coin as actions.

A description like the one given in Figure 2.1 only gives static information about a process. What we are most interested in is the behaviour of the process being specified. The behaviour of a process is described by giving a ‘CCS program’. The idea being that, as we shall see soon, the process constructions that are used in building the program allow us to describe both the structure of a process and its behaviour.

Inaction, prefixing and recursive definitions

Let us begin by introducing the constructs of the language CCS by means of examples. The most basic process of
all is the process $0$ (read ‘nil’). This is the most boring process imaginable, as it performs no action whatsoever. The process $0$ offers the prototypical example of a deadlocked behaviour—one that cannot proceed any further in its computation.

The most basic process constructor in CCS is *action prefixing*. Two example processes built using $0$ and action prefixing are a match and a complex match, described by the expressions

$$\text{strike.}0\text{ and take.strike.}0\ ,$$

respectively. Intuitively, a match is a process that dies when stricken (i.e., that becomes the process $0$ after executing the action strike), and a complex match is one that needs to be taken before it can behave like a match. More in general, the formation rule for action prefixing says that:

If $P$ is a process and $a$ is a label, then $a.P$ is a process.

The idea is that a label, like strike or pub, will denote an input or output action on a communication port, and that the process $a.P$ is one that begins by performing action $a$ and behaves like $P$ thereafter.

We have already mentioned that processes can be given names, very much like procedures can. This means that we can introduce names for (complex) processes, and that we can use these names in defining other process descriptions. For instance, we can give the name Match to the complex match thus:

$$\text{Match} \stackrel{\text{def}}{=} \text{take.strike.}0\ .$$

The introduction of names for processes allows us to give recursive definitions of process behaviours—compare with the recursive definition of procedures or methods in your favourite programming language. For instance, we may define the
2.1. SOME CCS PROCESS CONSTRUCTIONS

behaviour of an everlasting clock thus:

\[ \text{Clock} \overset{\text{def}}{=} \text{tick.Clock} . \]

Note that, since the process name Clock is a short-hand for the term on the right-hand side of the above equation, we may repeatedly replace the name Clock with its definition to obtain that

\[
\begin{align*}
\text{Clock} & \overset{\text{def}}{=} \text{tick.Clock} \\
& = \text{tick.tick.Clock} \\
& = \text{tick.tick.tick.Clock} \\
& \quad \vdots \\
& = \underbrace{\text{tick.\ldots.tick}}_{\text{n-times}} \text{.Clock} ,
\end{align*}
\]

for each positive integer \( n \).

As another recursive process specification, consider that of a simple coffee vending machine:

\[ \text{CM} \overset{\text{def}}{=} \text{coin.coffee.CM} . \quad (2.1) \]

This is a machine that is willing to accept a coin as input, deliver coffee to its customer, and thereafter return to its initial state.

**Choice** The CCS constructs that we have presented so far would not allow us to describe the behaviour of a vending machine that allows its paying customer to choose between tea and coffee, say. In order to allow for the description of processes whose behaviour may follow different patterns of interaction with their environment, CCS offers the *choice operator*, which is written ‘+’. For example, a vending machine offering either tea or coffee may be described thus:

\[ \text{CTM} \overset{\text{def}}{=} \text{coin.}(\text{coffee.CTM + tea.CTM}) . \quad (2.2) \]

The idea here is that, after having received a coin as input, the process CTM is willing to deliver either coffee or tea, depending on its customer’s choice. In general, the formation rule for choice states that:

\[ \text{If } P \text{ and } Q \text{ are processes, then so is } P + Q. \]

The process \( P + Q \) is one that has the initial capabilities of both \( P \) and \( Q \). However, choosing to perform initially an action from \( P \) will pre-empt the further execution of actions from \( Q \), and vice versa.
Exercise 2.1 Give a CCS process that describes a clock that ticks at least once, and that may stop ticking after each clock tick.

Exercise 2.2 Give a CCS process that describes a coffee machine that may behave like that given by (2.1), but may also steal the money it receives and fail at any time.

Exercise 2.3 A finite process graph $T$ is a quadruple $(Q, A, \delta, q_0)$, where

- $Q$ is a finite set of states,
- $A$ is a finite set of labels,
- $q_0 \in Q$ is the start state and
- $\delta : Q \times A \rightarrow 2^Q$ is the transition function.

Using the operators introduced so far, give a CCS process that ‘describes $T$’.

Parallel composition It is well-known that a computer scientist working in a research university is a machine for turning coffee into publications. The behaviour of such an academic may be described by the CCS process

$$CS \triangleq \text{pub.coin.coffee.CS}.$$ (2.3)

As made explicit by the above description, a computer scientist is initially keen to produce a publication—possibly straight out of her doctoral dissertation—, but she needs coffee to produce her next publication. Coffee is only available through interaction with the departmental coffee machine CM. In order to describe systems consisting of two or more processes running in parallel, and possibly interacting with each other, CCS offers the parallel composition operation, which is written ‘$|$’. For example, the CCS expression $CM | CS$ describes a system consisting of two processes—the coffee machine CM and the computer scientist CS—that run in parallel one with the other. These two processes may communicate via the communication ports they share and use in complementary fashion, namely coffee and coin. By complementary, we mean that one of the processes uses the port for input and the other for output. Potential communications are represented in Figure 2.2 by the solid lines linking complementary ports. The port pub is instead used by the computer scientist to communicate with her research environment, or, more prosaically, with other processes that may be present in her environment and that are willing to accept input along that port. One important thing to note is that the link between complementary ports in Figure 2.2 denotes that it is possible for the
computer scientist and the coffee machine to communicate in the parallel composition CM \| CS. However, we do not require that they must communicate with one another. Both the computer scientist and the coffee machine could use their complementary ports to communicate with other reactive systems in their environment. For example, another computer scientist CS’ can use the coffee machine CM, and, in so doing, make sure that he can produce publications to beef up his curriculum vitae, and thus be a worthy competitor for CS in the next competition for a tenured position. (See Figure 2.3.) Alternatively, the computer scientist may have access to another coffee machine in her environment, as pictured in Figure 2.4.

In general, given two CCS expressions P and Q, the process P \| Q describes a system in which

- P and Q may proceed independently or
- may communicate via complementary ports.

**Restriction and relabelling** Since academics like the computer scientist often live in a highly competitive ‘publish or perish’ environment, it may be fruitful for her to make the coffee machine CM private to her, and therefore inaccessible to her competitors. To make this possible, the language CCS offers an operation called restriction, whose aim is to delimit the scope of channel names in much the same way as variables have scope in block structured programming languages. For instance, using the operations \(\backslash\text{coin}\) and \(\backslash\text{coffee}\), we may hide the coin and coffee ports from the environment of the processes CM and CS. Define the process SmUni (for ‘Small University’) thus:

\[
\text{SmUni} \overset{\text{def}}{=} (\text{CM} \| \text{CS}) \backslash \text{coin} \backslash \text{coffee}.
\] (2.4)
Figure 2.3: The interface for process $CM \parallel CS \parallel CS'$
Figure 2.4: The interface for process CM | CS | CM'

As pictured in Figure 2.5, the restricted coin and coffee ports may now only be used for communication between the computer scientist and the coffee machine, and are not available for interaction with their environment. Their scope is restricted to the process SmUni. The only port of SmUni that is visible to its environment, e.g., to the competing computer scientist CS', is the one via which the computer scientist CS outputs her publications. In general, the formation rule for restriction is as follows:

If $P$ is a process and $L$ is a set of port names, then $P \setminus L$ is a process.

In $P \setminus L$, the scope of the port names in $L$ is restricted to $P$—those port names can only be used for communication within $P$.

Since a computer scientist cannot live on coffee alone, it is beneficial for her to have access to other types of vending machines offering, say, chocolate, dried figs and crisps. The behaviour of these machines may be easily specified by means of minor variations on equation 2.1 on page 11. For instance, we may define the
Figure 2.5: The interface for process SmUni | CS'}
processes

\[
\begin{align*}
\text{CHM} & \overset{\text{def}}{=} \text{coin}.\text{choc}.\text{CHM} \\
\text{DFM} & \overset{\text{def}}{=} \text{coin}.\text{figs}.\text{DFM} \\
\text{CRM} & \overset{\text{def}}{=} \text{coin}.\text{crisps}.\text{CRM}.
\end{align*}
\]

Note, however, that all of these vending machines follow a common behavioural pattern, and may be seen as specific instances of a \textit{generic} vending machine that receives a coin as input, dispenses an item and restarts, namely the process

\[
\text{VM} \overset{\text{def}}{=} \text{coin}.\text{item}.\text{VM}.
\]

All of the aforementioned specific vending machines may be obtained as appropriate ‘renamings’ of \text{VM}. For example,

\[
\text{CHM} \overset{\text{def}}{=} \text{VM}[\text{choc}/\text{item}],
\]

where \text{VM}[\text{choc}/\text{item}] is a process that behaves like \text{VM}, but outputs chocolate whenever \text{VM} dispenses the generic item. In general,

\[
\text{If } P \text{ is a process and } f \text{ is a function from labels to labels satisfying certain requirements that will be made precise in Section 2.2, then } P[f] \text{ is a process.}
\]

By introducing the relabelling operation, we have completed our informal tour of the operations offered by the language CCS for the description of process behaviours. We hope that this informal introduction has given our readers a feeling for the language, and that our readers will agree with us that CCS is indeed a language based upon very few operations with an intuitively clear semantic interpretation. In passing, we have also hinted at the fact that CCS processes may be seen as defining automata which describe their behaviour—see Exercise 2.3. We shall now expand a little on the connection between CCS expressions and the automata describing their behaviour. The presentation will again be informal, as we plan to highlight the main ideas underlying this connection rather than to focus immediately on the technicalities. The formal connection between CCS expressions and labelled transition systems will be presented in Section 2.2 using the tools of Structural Operational Semantics (Plotkin, 1981; Plotkin, 2004b).

\subsection*{2.1.1 The behaviour of processes}

The key idea underlying the semantics of CCS is that a process passes through states during its execution; processes change their state by performing actions. For
instance, for the purpose of notational convenience in what follows, let us redefine the process CS (originally defined in equation 2.3 on page 12) as in Table 2.1. (This is the definition of the process CS that we shall use from now on, both when discussing its behaviour in isolation and in the context of other processes—for instance, as a component of the process SmUni.) Process CS can perform action \texttt{pub} and evolve into a process whose behaviour is described by the CCS expression CS\textsubscript{1} in doing so. Process CS\textsubscript{1} can then output a coin, thereby evolving into a process whose behaviour is described by the CCS expression CS\textsubscript{2}. Finally, this process can receive coffee as input, and behave like our good old CS all over again. Thus the processes CS, CS\textsubscript{1} and CS\textsubscript{2} are the only possible states of the computation of process CS. Note, furthermore, that there is really no conceptual difference between processes and their states! By performing an action, a process evolves to another process that describes what remains to be executed of the original one.

In CCS, processes change state by performing transitions, and these transitions are labelled by the action that caused them. An example state transition is

\[
\text{CS} \xrightarrow{\text{pub}} \text{CS}_1,
\]

which says that CS can perform action \texttt{pub} and become CS\textsubscript{1} in doing so. The operational behaviour of our computer scientist CS is therefore completely described by the following labelled transition system.

In much the same way, we can make explicit the set of states of the coffee machine described in equation 2.1 on page 11 by rewriting that equation thus:

\[
\begin{align*}
\text{CM} & \overset{\text{def}}{=} \text{coin.CM}_1 \\
\text{CM}_1 & \overset{\text{def}}{=} \text{coffee.CM}.
\end{align*}
\]
Note that the computer scientist is willing to output a coin in state $CS_1$, as witnessed by the transition

$$CS_1 \xrightarrow{\text{coin}} CS_2,$$

and the coffee machine is willing to accept that coin in its initial state, because of the transition

$$CM \xrightarrow{\text{coin}} CM_1.$$ 

Therefore, when put in parallel with one another, these two processes may communicate and change state simultaneously. The result of the communication should be described as a state transition of the form

$$CM \parallel CS_1 \tau \rightarrow CM_1 \parallel CS_2.$$ 

However, we are now faced with an important design decision—namely, we should decide what label to use in place of the ‘?’ labelling the above transition. Should we decide to use a standard label denoting input or output on some port, then a third process might be able to synchronize further with the coffee machine and the computer scientist, leading to multi-way synchronization. The choice made by Milner in his design of CCS is different. In CCS, communication is via handshake, and leads to a state transition that is unobservable, in the sense that it cannot synchronize further. This state transition is labelled by a new label $\tau$. So the above transition is indicated by

$$CM \parallel CS_1 \xrightarrow{\tau} CM_1 \parallel CS_2.$$ 

In this way, the behaviour of the process $SmUni$ defined by equation 2.4 on page 13 can be described by the following labelled transition system.
Since \( \tau \) actions are supposed to be unobservable, the following process seems to be an appropriate high level specification of the behaviour exhibited by process SmUni:

\[
\text{Spec} \overset{\text{def}}{=} \text{pub.Spec}.
\]

Indeed, we expect that SmUni and Spec describe the same observable behaviour, albeit at different levels of abstraction. We shall see in the remainder of this book that one of the big questions in process theory is to come up with notions of ‘behavioural equivalence’ between processes that will allow us to establish formally that, for instance, SmUni and Spec do offer the same behaviour. But this is getting ahead of our story.

### 2.2 CCS, formally

Having introduced CCS by example, we now proceed to present formal definitions for its syntax and semantics.

#### 2.2.1 The model of labelled transition systems

We have already indicated in our examples how the operational semantics for CCS can be given in terms of automata—which we have called labelled transition systems, as customary in concurrency theory. These we now proceed to define, for the sake of clarity. We first introduce the ingredients in the model of labelled transition systems informally, and then provide its formal definition.

In the model of labelled transition systems, processes are represented by vertices of certain edge-labelled directed graphs (the labelled transition systems themselves) and a change of process state caused by performing an action is understood as moving along an edge, labelled by the action name, that goes out of that state.

A labelled transition system consists therefore of a set of states (also referred to as processes or configurations), a set of labels (or actions), and a transition relation \( \rightarrow \) describing changes in process states: if a process \( p \) can perform an action \( a \) and become a process \( p' \), we write \( p \xrightarrow{a} p' \). Sometimes a state is singled out as the start state in the labelled transition system under consideration. In that case, we say that the labelled transition system is rooted.

#### Example 2.1

Let us start with a variation on the classic example of a tea/coffee vending machine. The very simplified behaviour of the process which determines the interaction of the machine with a customer can be described as follows. From the initial state—say, \( p \)—representing the situation ‘waiting for a request’, two possible actions are enabled. Either the tea button or the coffee button can be pressed.
(the corresponding action ‘tea’ or ‘coffee’ is executed) and the internal state of the machine changes accordingly to $p_1$ or $p_2$. Formally, this can be described by the transitions

\[ p \xrightarrow{\text{tea}} p_1 \text{ and } p \xrightarrow{\text{coffee}} p_2. \]

The target state $p_1$ records that the customer has requested tea, whereas $p_2$ describes the situation in which coffee has been selected.

Now the customer is asked to insert the corresponding amount of money, let us say one euro for a cup of tea and two euros for a cup of coffee. This is reflected by corresponding changes in the control state of the vending machine. These state changes can be modelled by the transitions

\[ p_1 \xrightarrow{1\mathcal{E}} p_3 \text{ and } p_2 \xrightarrow{2\mathcal{E}} p_3, \]

whose target state $p_3$ records that the machine has received payment for the chosen drink.

Finally, the drink is collected and the machine returns to its initial state $p$, ready to accept the request of another customer. This corresponds to the transition

\[ p_3 \xrightarrow{\text{collect}} p. \]

It is often convenient and suggestive to use a graphical representation for labelled transition systems. The following picture represents the tea/coffee machine described above.

![Diagram of tea/coffee machine](image)

Sometimes, when referring only to the process $p$, we do not have to give names to the other process states (in our example $p_1$, $p_2$ and $p_3$) and it is sufficient to provide the following labelled transition system for the process $p$. 
Remark 2.1 The definition of a labelled transition system permits situations like that in Figure 2.6 (where \( p \) is the initial state). In that labelled transition system, the state \( p_2 \), where the action \( c \) can be performed in a loop, is irrelevant for the behaviour of the process \( p \) since, as you can easily check, \( p_2 \) can never be reached from \( p \). This motivates us to introduce the notion of reachable states. We say that a state \( p' \) in the transition system representing a process \( p \) is reachable from \( p \) iff there exists an directed path from \( p \) to \( p' \). The set of all such states is called the set of reachable states. In our example this set contains exactly two states, namely \( p \) and \( p_1 \).

Definition 2.1 [Labelled transition system] A labelled transition system (LTS) (at times also called a transition graph) is a triple \((\text{Proc}, \text{Act}, \{a \rightarrow \mid a \in \text{Act}\})\), where:

- \( \text{Proc} \) is a set of states (or processes);
- \( \text{Act} \) is a set of actions (or labels);
- \( \rightarrow \subseteq \text{Proc} \times \text{Proc} \) is a transition relation, for every \( a \in \text{Act} \). As usual, we shall use the more suggestive notation \( s \xrightarrow{a} s' \) in lieu of \( (s, s') \in \rightarrow \), and write \( s \not\xrightarrow{a} \) (read ‘s refuses a’) iff \( s \xrightarrow{a} s' \) for no state \( s' \).
A labelled transition system is finite if its sets of states and actions are both finite.

For example, the LTS for the process SmUni defined by equation 2.4 on page 13 (see page 19) is formally specified thus:

\[
\begin{align*}
\text{Proc} & = \{ \text{SmUni}, (\text{CM} \mid \text{CS}_1) \setminus \text{coin} \setminus \text{coffee}, (\text{CM}_1 \mid \text{CS}_2) \setminus \text{coin} \setminus \text{coffee}, \\
& \qquad \quad (\text{CM} \mid \text{CS}) \setminus \text{coin} \setminus \text{coffee} \} \\
\text{Act} & = \{ \text{pub}, \tau \} \\
\text{pub} \rightarrow & = \{ (\text{SmUni}, (\text{CM} \mid \text{CS}_1) \setminus \text{coin} \setminus \text{coffee}), \\
& \quad (\text{CM} \mid \text{CS}) \setminus \text{coin} \setminus \text{coffee}, (\text{CM} \mid \text{CS}_1) \setminus \text{coin} \setminus \text{coffee} \} , \\
\tau \rightarrow & = \{ ((\text{CM} \mid \text{CS}_1) \setminus \text{coin} \setminus \text{coffee}, (\text{CM} \mid \text{CS}_2) \setminus \text{coin} \setminus \text{coffee}), \\
& \quad ((\text{CM} \mid \text{CS}) \setminus \text{coin} \setminus \text{coffee}, (\text{CM} \mid \text{CS}) \setminus \text{coin} \setminus \text{coffee}) \} .
\end{align*}
\]

As mentioned above, we shall often distinguish a so called start state (or initial state), which is one selected state in which the system initially starts. For example, the start state for the process SmUni presented above is, not surprisingly, the process SmUni itself.

**Remark 2.2** Sometimes the transition relations \( a \rightarrow \) are presented as a ternary relation \( \rightarrow \subseteq \text{Proc} \times \text{Act} \times \text{Proc} \) and we write \( s \xrightarrow{a} s' \) whenever \( (s, a, s') \in \rightarrow \). This is an alternative way to describe a labelled transition system and it defines the same notion as Definition 2.1.

**Notation 2.1** Let us now recall a few useful notations that will be used in connection with labelled transitions systems.

- We can extend the transition relation to the elements of \( \text{Act}^* \) (the set of all finite strings over \( \text{Act} \) including the empty string \( \varepsilon \)). The definition is as follows:
  
  \begin{itemize}
  \item \( s \xrightarrow{\varepsilon} s \) for every \( s \in \text{Proc} \), and
  \item \( s \xrightarrow{aw} s' \) iff there is a state \( t \in \text{Proc} \) such that \( s \xrightarrow{a} t \) and \( t \xrightarrow{w} s' \), for every \( s, s', t \in \text{Proc}, a \in \text{Act} \) and \( w \in \text{Act}^* \).
  \end{itemize}

In other words, if \( w = a_1a_2\cdots a_n \) for \( a_1, a_2, \ldots, a_n \in \text{Act} \) then we write \( s \xrightarrow{w} s' \) whenever there exist states \( s_0, s_1, \ldots, s_{n-1}, s_n \in \text{Proc} \) such that

\[
s = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} s_3 \xrightarrow{a_4} \cdots s_{n-1} \xrightarrow{a_n} s_n = s'.
\]

For the transition system in Figure 2.6 we have, for example, that \( p \xrightarrow{\varepsilon} p \), \( p \xrightarrow{ab} p \) and \( p \xrightarrow{bab} p \).
We write $s \rightarrow s'$ whenever there is an action $a \in \text{Act}$ such that $s \xrightarrow{a} s'$.

For the transition system in Figure 2.6 we have, for instance, that $p \rightarrow p_1$, $p_1 \rightarrow p$, $p_2 \rightarrow p_1$ and $p_2 \rightarrow p_2$.

• We use the notation $s \xrightarrow{a}$ meaning that there is some $s' \in \text{Proc}$ such that $s \xrightarrow{a} s'$.

For the transition system in Figure 2.6 we have, for instance, that $p \xrightarrow{a}$ and $p_1 \xrightarrow{b}$.

• We write $s \rightarrow^* s'$ iff $s \xrightarrow{w} s'$ for some $w \in \text{Act}^*$. In other words, $\rightarrow^*$ is the reflexive and transitive closure of the relation $\rightarrow$.

For the transition system in Figure 2.6 we have, for example, that $p \rightarrow^* p$, $p \rightarrow^* p_1$, and $p_2 \rightarrow^* p$.

**Exercise 2.4** Consider the following labelled transition system.

\[
\begin{array}{c}
s \xrightarrow{a} s_1 \\
\downarrow \quad a \\
\downarrow \\
s_3 \xrightarrow{a} s_2
\end{array}
\]

• Define the labelled transition system as a triple $(\text{Proc}, \text{Act}, \{a \rightarrow | a \in \text{Act}\})$.

• What is the reflexive closure of the binary relation $\xrightarrow{a}$? (A drawing is fine.)

• What is the symmetric closure of the binary relation $\xrightarrow{a}$? (A drawing is fine.)

• What is the transitive closure of the binary relation $\xrightarrow{a}$? (A drawing is fine.)

**Definition 2.2** [Reachable states] Let $T = (\text{Proc}, \text{Act}, \{a \rightarrow | a \in \text{Act}\})$ be a labelled transition system, and let $s \in \text{Proc}$ be its initial state. We say that $s' \in \text{Proc}$ is reachable in the transition system $T$ iff $s \rightarrow^* s'$. The set of reachable states contains all states reachable in $T$.

In the transition system from Figure 2.6, where $p$ is the initial state, the set of reachable states is equal to $\{p, p_1\}$. 

\[\boxdot\]
Exercise 2.5 What would the set of reachable states in the labelled transition system in Figure 2.6 be if its start state were $p_2$?

The step from a process denoted by a CCS expression to the LTS describing its operational behaviour is taken using the framework of Structural Operational Semantics (SOS) as pioneered by Plotkin in (Plotkin, 2004b). (The history of the development of the ideas that led to SOS is recounted by Plotkin himself in (Plotkin, 2004a).) The key idea underlying this approach is that the collection of CCS process expressions will be the set of states of a (large) labelled transition system, whose actions will be either input or output actions on communication ports or $\tau$, and whose transitions will be exactly those that can be proven to hold by means of a collection of syntax-driven rules. These rules will capture the informal semantics of the CCS operators presented above in a very simple and elegant way. The operational semantics of a CCS expression is then obtained by selecting that expression as the start state in the LTS for the whole language, and restricting ourselves to the collection of CCS expressions that are reachable from it by following transitions.

2.2.2 The formal syntax and semantics of CCS

The next step in our formal developments is to offer the formal syntax for the language CCS. Since the set of ports plays a crucial role in the definition of CCS processes, we begin by assuming a countably infinite collection $\mathcal{A}$ of (channel) names. (‘Countably infinite’ means that we have as many names as there are natural numbers.) The set

$$\bar{\mathcal{A}} = \{\bar{a} \mid a \in \mathcal{A}\}$$

is the set of complementary names (or co-names for short). In our informal introduction to the language, we have interpreted names as input actions and co-names as output actions. We let

$$\mathcal{L} = \mathcal{A} \cup \bar{\mathcal{A}}$$

be the set of labels, and

$$\text{Act} = \mathcal{L} \cup \{\tau\}$$

be the set of actions. In our formal developments, we shall use $a, b$ to range over $\mathcal{L}$ and $\alpha$ as a typical member of $\text{Act}$, but, as we have already done in the previous section, we shall often use more suggestive names for channels in applications and examples. By convention, we assume that $\bar{\bar{a}} = a$ for each label $a$. (This also makes sense intuitively because the complement of output is input.) We also assume a given countably infinite collection $\mathcal{K}$ of process names (or constants). (This ensures that we never run out of names for processes.)
Definition 2.3 The collection $\mathcal{P}$ of CCS expressions is given by the following grammar:

$$P, Q ::= K \mid \alpha.P \mid \sum_{i \in I} P_i \mid P \parallel Q \mid P[f] \mid P \setminus L,$$

where

- $K$ is a process name in $\mathcal{K}$;
- $\alpha$ is an action in $\text{Act}$;
- $I$ is a possibly infinite index set;
- $f : \text{Act} \to \text{Act}$ is a relabelling function satisfying the following constraints:
  
  $$f(\tau) = \tau \text{ and } f(\bar{a}) = \overline{f(a)} \text{ for each label } a;$$

- $L$ is a set of labels from $\mathcal{L}$.

We write $0$ for an empty sum of processes, i.e.,

$$0 = \sum_{i \in \emptyset} P_i,$$

and $P_1 + P_2$ for a sum of two processes, i.e.,

$$P_1 + P_2 = \sum_{i \in \{1,2\}} P_i.$$

Moreover, we assume that the behaviour of each process constant $K \in \mathcal{K}$ is given by a defining equation

$$K \overset{\text{def}}{=} P,$$

where $P \in \mathcal{P}$. As it was already made clear by the previous informal discussion, the constant $K$ may appear in $P$. ♦

We sometimes write $[b_1/a_1, \ldots, b_n/a_n]$, where $n \geq 1$, $a_i, b_i \in \mathcal{A}$ for each $i \in \{1, \ldots, n\}$ and the $a_i$ are distinct channel names, for the relabelling $[f]$, where $f$ is the relabelling function mapping each $a_i$ to $b_i$, each $\overline{a_i}$ to $\overline{b_i}$ ($i \in \{1, \ldots, n\}$) and acting like the identity function on all of the other actions. For each label $a$, we also often write $\backslash a$ in lieu of $\{a\}$.

To avoid the use of too many parentheses in writing CCS expressions, we use the convention that the operators have decreasing binding strength in the following
order: restriction and relabelling (tightest binding), action prefixing, parallel composition and summation. For example, the expression $a.0 \parallel b.P \setminus L + c.0$ stands for

$((a.0) \parallel (b.(P \setminus L))) + (c.0)$.

**Exercise 2.6** Which of the following expressions are syntactically correct CCS expressions? Why? Assume that $A, B$ are process constants and $a, b$ are channel names.

- $a.b.A + B$
- $(a.0 \parallel \tau.A) \setminus \{a, b\}$
- $(a.0 \parallel \tau.A) \setminus \{a, \tau\}$,
- $a.B + [a/b]$
- $\tau.\tau.B + 0$
- $(a.B + b.B)[a/b, b/a]$
- $(a.B + \tau.B)[a/\tau, b/a]$
- $(a.b.A + \tau.0) \parallel B$
- $(a.b.A + \tau.0).B$
- $(a.b.A + \tau.0) + B$
- $(0 \parallel 0) + 0$

Our readers can easily check that all of the processes presented in the previous section are indeed CCS expressions. Another example of a CCS expression is given by a counter, which is defined thus:

Counter$_0 \overset{\text{def}}{=} \text{up.Counter}_1$ \hfill (2.5)
Counter$_n \overset{\text{def}}{=} \text{up.Counter}_{n+1} + \text{down.Counter}_{n-1} \quad (n > 0)$. \hfill (2.6)

The behaviour of such a process is intuitively clear. For each non-negative integer $n$, the process Counter$_n$ behaves like a counter whose value is $n$; the ‘up’ actions increase the value of the counter by one, and the ‘down’ actions decrease it by one. It would also be easy to construct the (infinite state) LTS for this process based on its syntactic description, and on the intuitive understanding of process behaviour.
we have so far developed. However, intuition alone can lead us to wrong conclusions, and most importantly cannot be fed to a computer! To capture formally our understanding of the semantics of the language CCS, we therefore introduce the collection of SOS rules in Table 2.2. These rules are used to generate an LTS whose states are CCS expressions. In that LTS, a transition \( P \xrightarrow{\alpha} Q \) holds for CCS expressions \( P, Q \) and action \( \alpha \) if, and only if, it can be proven using the rules in Table 2.2.

A rule like

\[
\alpha.P \xrightarrow{\alpha} P
\]

is an axiom, as it has no premises—that is, it has no transition above the solid line. This means that proving that a process of the form \( \alpha.P \) affords the transition \( \alpha.P \xrightarrow{\alpha} P \) (the conclusion of the rule) can be done without establishing any further sub-goal. Therefore each process of the form \( \alpha.P \) affords the transition \( \alpha.P \xrightarrow{\alpha} P \). As an example, we have that the following transition

\[
\text{pub}.CS_1 \xrightarrow{\text{pub}} CS_1
\]

(2.7)

is provable using the above rule for action prefixing.

On the other hand, a rule like

\[
\frac{P \xrightarrow{\alpha} P'}{K \xrightarrow{\alpha} P'} \quad K \overset{\text{def}}{=} P
\]

has a non-empty set of premises. This rule says that to establish that constant \( K \) affords the transition mentioned in the conclusion of the rule, we have to prove first that the body of the defining equation for \( K \), namely the process \( P \), affords the transition \( P \xrightarrow{\alpha} P' \). Using this rule, pattern matching and transition (2.7), we can prove the transition

\[
CS \xrightarrow{\text{pub}} CS_1
\]

which we had informally derived before for the version of process CS given in Table 2.1 on page 18.

The aforementioned rule for constants has a side condition, namely \( K \overset{\text{def}}{=} P \), that describes a constraint that must be met in order for the rule to be applicable. In that specific example, the side condition states intuitively that the rule may be used to derive an initial transition for constant \( K \) if ‘\( K \) is declared to have body \( P \).’

Another example of a rule with a side condition is that for restriction.

\[
\frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L} \quad \alpha, \bar{\alpha} \notin L
\]
### Table 2.2: SOS rules for CCS ($\alpha \in \text{Act}$, $a \in \mathcal{L}$)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ACT</strong></td>
<td>$\alpha.P \xrightarrow{\alpha} P$</td>
</tr>
<tr>
<td><strong>SUM</strong></td>
<td>$\frac{P_j \xrightarrow{\alpha} P'<em>j}{\sum</em>{j \in I} P_i \xrightarrow{\alpha} P'_j}$ where $j \in I$</td>
</tr>
<tr>
<td><strong>COM1</strong></td>
<td>$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$</td>
</tr>
<tr>
<td><strong>COM2</strong></td>
<td>$\frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$</td>
</tr>
<tr>
<td><strong>COM3</strong></td>
<td>$\frac{P \xrightarrow{\alpha} P', Q \xrightarrow{\bar{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$</td>
</tr>
<tr>
<td><strong>RES</strong></td>
<td>$\frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L}$ where $\alpha, \bar{\alpha} \notin L$</td>
</tr>
<tr>
<td><strong>REL</strong></td>
<td>$\frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$</td>
</tr>
<tr>
<td><strong>CON</strong></td>
<td>$\frac{P \xrightarrow{\alpha} P'}{K \xrightarrow{\alpha} P'}$ where $K \overset{\text{def}}{=} P$</td>
</tr>
</tbody>
</table>
This rule states that every transition of a term $P$ determines a transition of the expression $P \setminus L$, provided that neither the action producing the transition nor its complement are in $L$. For example, as you can check, this side condition prevents us from proving the existence of the transition

$$(\text{coffee.CS}) \setminus \text{coffee} \xrightarrow{\text{coffee}} \text{CS} \setminus \text{coffee}.$$ 

Finally, note that, when considering the binary version of the summation operator, the family of rules $\text{SUM}_j$ reduces to the following two rules.

\[
\begin{align*}
\text{SUM}_1 & \quad \frac{P_1 \xrightarrow{\alpha} P'_1}{P_1 + P_2 \xrightarrow{\alpha} P'_1} \\
\text{SUM}_2 & \quad \frac{P_2 \xrightarrow{\alpha} P'_2}{P_1 + P_2 \xrightarrow{\alpha} P'_2}
\end{align*}
\]

To get a feeling for the power of recursive definitions of process behaviours, consider the process $C$ defined thus:

$$C \stackrel{\text{def}}{=} \text{up}.(C \mid \text{down}.0). \quad (2.8)$$

What are the transitions that this process affords? Using the rules for constants and action prefixing, you should have little trouble in arguing that the only initial transition for $C$ is

$$C \xrightarrow{\text{up}} C \mid \text{down}.0. \quad (2.9)$$

What next? Observing that $\text{down}.0 \xrightarrow{\text{down}} 0$, using rule COM2 in Table 2.2 we can infer that

$$C \mid \text{down}.0 \xrightarrow{\text{down}.0} C \mid 0.$$ 

Since it is reasonable to expect that the process $C \mid 0$ exhibits the same behaviour as $C$—and we shall see later on that this does hold true—, the above transition effectively brings our process back to its initial state, at least up to behavioural equivalence. However, this is not all, because, as we have already proven (2.9), using rule COM1 in Table 2.2 we have that the transition

$$C \mid \text{down}.0 \xrightarrow{\text{up}} (C \mid \text{down}.0) \mid \text{down}.0$$

is also possible. You might find it instructive to continue building a little more of the transition graph for process $C$. As you may begin to notice, the LTS giving the operational semantics of the process expression $C$ looks very similar to that for $\text{Counter}_0$, as given in (2.5). Indeed, we shall prove later on that these two processes exhibit the same behaviour in a very strong sense.

**Exercise 2.7** Use the rules of the SOS semantics for CCS to derive the LTS for the process $\text{SmUni}$ defined by equation 2.4 on page 13. (Use the definition of CS in Table 2.1.)

\[\Diamond\]
Exercise 2.8 Assume that \( A \overset{\text{def}}{=} b.a.B \). By using the SOS rules for CCS prove the existence of the following transitions:

- \((A | \overline{b}.0) \setminus \{b\} \xrightarrow{\tau} (a.B | 0) \setminus \{b\}\),
- \((A | \overline{b}a.B) + (\overline{b}.A) [a/b] \xrightarrow{\tau} (A | a.B)\), and
- \((A | \overline{b}a.B) + (\overline{b}.A) [a/b] \xrightarrow{\tau} A[a/b]\).

Exercise 2.9 Draw (part of) the transition graph for the process name \( A \) whose behaviour is given by the defining equation

\[ A \overset{\text{def}}{=} (a.A) \setminus b. \]

The resulting transition graph should have infinitely many states. Can you think of a CCS term that generates a finite labelled transition system that should intuitively have the same behaviour as \( A \)?

Exercise 2.10 Draw (part of) the transition graph for the process name \( A \) whose behaviour is given by the defining equation

\[ A \overset{\text{def}}{=} (a_0.A) [f] \]

where we assume that the set of channel names is \( \{a_0, a_1, a_2, \ldots\} \), and \( f(a_i) = a_{i+1} \) for each \( i \).

The resulting transition graph should (again!) have infinitely many states. Can you give an argument showing that there is no finite state labelled transition system that could intuitively have the same behaviour as \( A \)?

Exercise 2.11

1. Draw the transition graph for the process name \( \text{Mutex}_1 \) whose behaviour is given by the defining equation

\[ \text{Mutex}_1 \overset{\text{def}}{=} (\text{User} | \text{Sem}) \setminus \{p, v\} \]

\[ \text{User} \overset{\text{def}}{=} \overline{p}.\text{enter} \cdot \overline{v}.\text{exit} \cdot \overline{\text{User}} \]

\[ \text{Sem} \overset{\text{def}}{=} p.v.\text{Sem} . \]
2. Draw the transition graph for the process name Mutex whose behaviour is given by the defining equation

\[ \text{Mutex}_2 \overset{\text{def}}{=} ((\text{User} | \text{Sem}) | \text{User}) \setminus \{p, v\}, \]

where User and Sem are defined as before.

Would the behaviour of the process change if User was defined as

\[ \text{User} \overset{\text{def}}{=} \overline{p}.\text{enter} \cdot \overline{v}.\text{exit} \cdot \overline{\text{User}}? \]

3. Draw the transition graph for the process name FMutex whose behaviour is given by the defining equation

\[ \text{FMutex} \overset{\text{def}}{=} ((\text{User} | \text{Sem}) | \text{FUser}) \setminus \{p, v\}, \]

where User and Sem are defined as before, and the behaviour of FUser is given by the defining equation

\[ \text{FUser} \overset{\text{def}}{=} \overline{p}.\text{enter} \cdot (\overline{v}.\text{exit} \cdot \overline{\text{FUser}} + \overline{\text{exit} \cdot \overline{\text{FUser}}} + \overline{\text{exit} \cdot \overline{0}}). \]

Do you think that Mutex_2 and FMutex are offering the same behaviour? Can you argue informally for your answer?

♦

2.2.3 Value passing CCS

This section may be skipped on first reading as it is meant mainly as a pointer for further reading and self-study.

So far, we have only introduced the so-called pure CCS—that is, the fragment of CCS where communication is pure synchronization and involves no exchange of data. In many applications, however, processes exchange data when they communicate. To allow for a natural modelling of these examples, it is convenient, although theoretically unnecessary as argued in (Milner, 1989, Section 2.8), to extend our language to what is usually called value passing CCS. We shall now introduce the new features in this language, and their operational semantics, by means of examples. In what follows, we shall assume for simplicity that the only data type is the set of non-negative integers.

Assume that we wish to define a one-place buffer B which has the following behaviour.
2.2. CCS, FORMALLY

- If B is empty, then it is only willing to accept one datum as input along a channel called ‘in’. The received datum is stored for further output.

- If B is full, then it is only willing to output the successor of the value it stores, and empties itself in doing so.

This behaviour of B can be modelled in value passing CCS thus:

\[
\begin{align*}
B & \overset{\text{def}}{=} \text{in}(x).B(x) \\
B(x) & \overset{\text{def}}{=} \text{out}(x + 1).B.
\end{align*}
\]

Note that the input prefix ‘in’ now carries a parameter that is a variable—in this case \(x\)—whose scope is the process that is prefixed by the input action—in this example, \(B(x)\). The intuitive idea is that process B is willing to accept a non-negative integer \(n\) as input, bind the received value to \(x\) and thereafter behave like \(B(n)\)—that is, like a full one-place buffer storing the value \(n\). The behaviour of the process \(B(n)\) is then described by the second equation above, where the scope of the formal parameter \(x\) is the whole right-hand side of the equation. Note that output prefixes, like \(\text{out}(x + 1)\) above, may carry expressions—the idea being that the value being output is the one that results from the evaluation of the expression.

The general SOS rule for input prefixing now becomes

\[
\alpha(x).P \xrightarrow{a(n)} P[n/x] \quad n \geq 0
\]

where we write \(P[n/x]\) for the expression that results by replacing each free occurrence of the variable \(x\) in \(P\) with \(n\). The general SOS rule for output prefixing is instead the one below.

\[
\bar{\alpha}(e).P \xrightarrow{\alpha(n)} P \quad n \text{ is the result of evaluating } e
\]

In value passing CCS, as we have already seen in our definition of the one place buffer B, process names may be parameterized by value variables. The general form that these parameterized constants may take is \(A(x_1, \ldots, x_n)\), where \(A\) is a process name, \(n \geq 0\) and \(x_1, \ldots, x_n\) are distinct value variables. The operational semantics for these constants is given by the following rule.

\[
P[v_1/x_1, \ldots, v_n/x_n] \xrightarrow{\alpha} P' \quad A(e_1, \ldots, e_n) \overset{\text{def}}{=} P \text{ and each } e_i \text{ has value } v_i
\]
To become familiar with these rules, you should apply them to the one-place buffer B, and derive its possible transitions.

In what follows, we shall restrict ourselves to CCS expressions that have no free occurrences of value variables—that is, to CCS expressions in which each occurrence of a value variable, say $y$, is within the scope of an input prefix of the form $a(y)$ or of a parameterized constant $A(x_1, \ldots, x_n)$ with $y = x_i$ for some $1 \leq i \leq n$. For instance, the expression

$$a(x).\overline{b}(y + 1).0$$

is disallowed because the single occurrence of the value variable $y$ is bound neither by an input prefixing nor by a parameterized constant.

Since processes in value passing CCS may manipulate data, it is natural to add an if $bexp$ then $P$ else $Q$ construct to the language, where $bexp$ is a boolean expression. Assume, by way of example, that we wish to define a one-place buffer $Pred$ that computes the predecessor function on the non-negative integers. This may be defined thus:

$$Pred \overset{\text{def}}{=} \text{in}(x).Pred(x)$$

$$Pred(x) \overset{\text{def}}{=} \text{if } x = 0 \text{ then } \overline{\text{out}}(0).Pred \text{ else } \overline{\text{out}}(x - 1).Pred .$$

We expect $Pred(0)$ to output the value 0 on channel ‘out’, and $Pred(n + 1)$ to output $n$ on the same channel for each non-negative integer $n$. The SOS rules for if $bexp$ then $P$ else $Q$ will allow us to prove this formally. They are the expected ones, namely

$$\frac{P \overset{\alpha}{\Rightarrow} P'}{\text{if } bexp \text{ then } P \text{ else } Q \overset{\alpha}{\Rightarrow} P'} \quad \text{bexp is true}$$

and

$$\frac{Q \overset{\alpha}{\Rightarrow} Q'}{\text{if } bexp \text{ then } P \text{ else } Q \overset{\alpha}{\Rightarrow} Q'} \quad \text{bexp is false} .$$

Exercise 2.12 Consider a one place buffer defined by

$$\text{Cell} \overset{\text{def}}{=} \text{in}(x).\text{Cell}(x)$$

$$\text{Cell}(x) \overset{\text{def}}{=} \overline{\text{out}}(x).\text{Cell} .$$

Use the Cell to define a two-place bag and a two-place FIFO queue. (Recall that a bag, also known as multiset, is a set whose elements have multiplicity.) Give specifications of the expected behaviour of these processes, and use the operational rules given above to convince yourselves that your implementations are correct. ♦
2.2. **CCS, FORMALLY**

**Exercise 2.13** Consider the process $B$ defined thus:

\[
B \overset{\text{def}}{=} \text{push}(x).(C(x) \triangleright B) + \text{empty}.B
\]

\[
C(x) \overset{\text{def}}{=} \text{push}(y).(C(y) \triangleright C(x)) + \overline{\text{pop}}(x).D
\]

\[
D \overset{\text{def}}{=} o(x).C(x) + \overline{e}.B
\]

where the linking combinator $P \triangleright Q$ is as follows:

\[
P \triangleright Q = (P[p'/p, e'/e, o'/o] \mid Q[p'/\text{push}, e'/\text{empty}, o'/\text{pop}]) \setminus \{p', o', e'\}.
\]

Draw an initial fragment of the transition graph for this process. What behaviour do you think $B$ implements?

**Exercise 2.14 (For the theoretically minded)** Prove that the operational semantics for value passing CCS we have given above is in complete agreement with the semantics for this language via translation into the pure calculus given by Milner in *Milner, 1989, Section 2.8*.
Chapter 3

Behavioural equivalence

We have previously remarked that CCS, like all other process algebras, can be used to describe both implementations of processes and specifications of their expected behaviours. A language like CCS therefore supports the so-called *single language approach* to process theory—that is, the approach in which a single language is used to describe both actual processes and their specifications. An important ingredient of these languages is therefore a notion of behavioural equivalence or behavioural approximation between processes. One process description, say SYS, may describe an implementation, and another, say SPEC, may describe a specification of the expected behaviour. To say that SYS and SPEC are equivalent is taken to indicate that these two processes describe essentially the same behaviour, albeit possibly at different levels of abstraction or refinement. To say that, in some formal sense, SYS is an approximation of SPEC means roughly that every aspect of the behaviour of this process is allowed by the specification SPEC, and thus that nothing unexpected can happen in the behaviour of SYS. This approach to program verification is also sometimes called *implementation verification* or *equivalence checking*.

3.1 Criteria for a good behavioural equivalence

We have already informally argued that some of the processes that we have met so far ought to be considered behaviourally equivalent. For instance, we claimed that the behaviour of the process SmUni defined in equation 2.4 on page 13 should be considered equivalent to that of the specification

\[ \text{Spec} \overset{\text{def}}{=} \text{pub.Spec} , \]
and that the process C in equation 2.8 on page 30 behaves like a counter. Our order of business now will be to introduce a suitable notion of behavioural equivalence that will allow us to establish these expected equalities and many others.

Before doing so, it is however instructive to consider the criteria that we expect a suitable notion of behavioural equivalence for processes to meet. First of all, we have already used the term ‘equivalence’ several times, and since this is a mathematical notion that some of you may not have met before, it is high time to define it precisely.

**Definition 3.1** Let $X$ be a set. A **binary relation** over $X$ is a subset of $X \times X$, the set of pairs of elements of $X$. If $R$ is a binary relation over $X$, we often write $x R y$ instead of $(x, y) \in R$.

An **equivalence relation** over $X$ is a binary relation $R$ that satisfies the following constraints:

- $R$ is **reflexive**—that is, $x R x$ for each $x \in X$;
- $R$ is **symmetric**—that is, $x R y$ implies $y R x$, for all $x, y \in X$; and
- $R$ is **transitive**—that is, $x R y$ and $y R z$ imply $x R z$, for all $x, y, z \in X$.

A reflexive and transitive relation is a **preorder**.

An equivalence relation is therefore a more abstract version of the notion of equality that we are familiar with since elementary school.

**Exercise 3.1** Which of the following relations over the set of non-negative integers $\mathbb{N}$ is an equivalence relation?

- The identity relation $I = \{(n, n) \mid n \in \mathbb{N}\}$.
- The universal relation $U = \{(n, m) \mid n, m \in \mathbb{N}\}$.
- The standard $\leq$ relation.
- The parity relation $M_2 = \{(n, m) \mid n, m \in \mathbb{N}, \ n \ \text{mod} \ 2 = m \ \text{mod} \ 2\}$.

Can you give an example of a preorder over the set $\mathbb{N}$ that is not an equivalence relation?

Since we expect that each process is a correct implementation of itself, a relation used to support implementation verification should certainly be reflexive. Moreover, as we shall now argue, it should also be transitive—at least if it is to support stepwise derivation of implementations from specifications. In fact, assume that we wish to derive a correct implementation from a specification via a sequence of
3.1. CRITERIA FOR A GOOD BEHAVIOURAL EQUIVALENCE

refinement steps which are known to preserve some behavioural relation $R$. In this approach, we might begin from our specification Spec and transform it into our implementation Imp via a sequence of intermediate stages Spec$_i$ ($0 \leq i \leq n$) thus:

$$\text{Spec} = \text{Spec}_0 R \text{Spec}_1 R \text{Spec}_2 R \cdots R \text{Spec}_n = \text{Imp}.$$  

Since each of the steps above preserves the relation $R$, we would like to conclude that Imp is a correct implementation of Spec with respect to $R$—that is, that

$$\text{Spec} R \text{Imp}$$

holds. This is guaranteed to be true if the relation $R$ is transitive.

From the above discussion, it follows that a relation supporting implementation verification should at least be a preorder. The relations considered in the classic theory of CCS, and in the main body of this book, are also symmetric, and are therefore equivalence relations.

Another intuitively desirable property that an equivalence relation $R$ that supports implementation verification should have is that it is a congruence. This means that process descriptions that are related by $R$ can be used interchangeably as parts of a larger process description without affecting its overall behaviour. More precisely, if $P R Q$ and $C[]$ is a program fragment with ‘a hole’, then

$$C[P] R C[Q].$$

This is pictorially represented in Figure 3.1.

Finally, we expect our notion of relation supporting implementation verification to be based on the observable behaviour of processes, rather than on their structure, the actual name of their states or the number of transitions they afford. Ideally, we should like to identify two processes unless there is some sequence of ‘interactions’ that an ‘observer’ may have with them leading to different ‘outcomes’.
The lack of consensus on what constitutes an appropriate notion of observable behaviour for reactive systems has led to a large number of proposals for behavioural equivalences for concurrent processes. (See the study (Glabbeek, 2001), where van Glabbeek presents the linear time-branching time spectrum—a lattice of known behavioural equivalences and preorders over labelled transition systems, ordered by inclusion.) In our search for a reasonable notion of behavioural relation to support implementation verification, we shall limit ourselves to presenting a tiny sample of these.

So let’s begin our search!

3.2 Trace equivalence: a first attempt

Labelled transition systems (LTSs) (Keller, 1976) are a fundamental model of concurrent computation, which is widely used in light of its flexibility and applicability. In particular, they are the prime model underlying Plotkin’s Structural Operational Semantics (Plotkin, 2004b) and, following Milner’s pioneering work on CCS (Milner, 1989), are by now the standard semantic model for various process description languages.

As we have already seen, LTSs model processes by explicitly describing their states and their transitions from state to state, together with the actions that produced them. Since this view of process behaviours is very detailed, several notions of behavioural equivalence and preorder have been proposed for LTSs. The aim of such behavioural semantics is to identify those (states of) LTSs that afford the same ‘observations’, in some appropriate technical sense.

Now, LTSs are essentially (possibly infinite state) automata, and the classic theory of automata suggests a ready made notion of equivalence for them, and thus for the CCS processes that denote them.

Let us say that a trace of a process $P$ is a sequence $\alpha_1 \cdots \alpha_k \in \text{Act}^* (k \geq 0)$ such that there exists a sequence of transitions

$$P = P_0 \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} P_2 \cdots P_{k-1} \xrightarrow{\alpha_k} P_k,$$

for some $P_1, \ldots, P_k$. We write $\text{Traces}(P)$ for the collection of all traces of $P$. Since $\text{Traces}(P)$ describes all the possible finite sequences of interactions that we may have with process $P$, it is reasonable to require that our notion of behavioural equivalence only relates processes that afford the same traces, or else we should have a very good reason for telling them apart—namely a sequence of actions that can be performed with one, but not with the other. This means that, for all processes $P$ and $Q$, we require that

$$\text{if } P \text{ and } Q \text{ are behaviourally equivalent, then } \text{Traces}(P) = \text{Traces}(Q). \quad (3.1)$$
Taking the point of view of standard automata theory, and abstracting from the notion of ‘accept state’ that is missing altogether in our treatment, an automaton may be completely identified by its set of traces, and thus two processes are equivalent if, and only if, they afford the same traces.

This point of view is totally justified and natural if we view our LTSs as nondeterministic devices that may generate or accept sequences of actions. However, is it still a reasonable one if we view our automata as reactive machines that interact with their environment?

To answer this question, consider the coffee and tea machine CTM defined in equation 2.2 on page 11, and compare it with the following one:

\[
CTM' \overset{\text{def}}{=} \text{coin.coffee.CTM' + coin.tea.CTM' .} \tag{3.2}
\]

You should be able to convince yourselves that CTM and CTM' afford the same traces. (Do so!) However, if you were a user of the coffee and tea machine who wants coffee and hates tea, which machine would you like to interact with? We certainly would prefer to interact with CTM as that machine will give us coffee after receiving a coin, whereas CTM' may refuse to deliver coffee after having accepted our coin!

This informal discussion may be directly formalized within CCS by assuming that the behaviour of the coffee starved user is described by the process

\[
CA \overset{\text{def}}{=} \text{coin.coffee.CA} .
\]

Consider now the terms

\[
(CA \mid CTM) \setminus \{\text{coin, coffee, tea}\}
\]

and

\[
(CA \mid CTM') \setminus \{\text{coin, coffee, tea}\}
\]

that we obtain by forcing interaction between the coffee addict CA and the two vending machines. Using the SOS rules for CCS, you should convince yourselves that the former term can only perform an infinite computation consisting of \(\tau\)-labelled transitions, whereas the second term can deadlock thus:

\[
(CA \mid CTM') \setminus \{\text{coin, coffee, tea}\} \xrightarrow{\tau} (\text{coffee.CA} \mid \overline{\text{tea}}.CTM') \setminus \{\text{coin, coffee, tea}\} .
\]

Note that the target term of this transition captures precisely the deadlock situation that we intuitively expected to have, namely that the user only wants coffee, but the machine is only willing to deliver tea. So trace equivalent terms may exhibit
different deadlock behaviour when made to interact with other parallel processes—a highly undesirable state of affairs.

In light of the above example, we are forced to reject the law
\[ \alpha(P + Q) = \alpha P + \alpha Q, \]
which is familiar from the standard theory of regular languages, for our desired notion of behavioural equivalence. (Can you see why?) Therefore we need to refine our notion of equivalence in order to differentiate processes that, like the two vending machines above, exhibit different reactive behaviour while still having the same traces.

**Exercise 3.2 (Recommended)** A completed trace of a process \( P \) is a sequence \( \alpha_1 \cdots \alpha_k \in \text{Act}^* \) \((k \geq 0)\) such that there exists a sequence of transitions

\[ P = P_0 \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} P_2 \cdots P_{k-1} \xrightarrow{\alpha_k} P_k \not\rightarrow, \]

for some \( P_1, \ldots, P_k \). The completed traces of a process may be seen as capturing its deadlock behaviour, as they are precisely the sequences of actions that may lead the process into a state from which no further action is possible.

1. Do the processes

\[ (CA \mid CTM) \setminus \{\text{coin, coffee, tea}\} \]

and

\[ (CA \mid CTM') \setminus \{\text{coin, coffee, tea}\} \]

defined above have the same completed traces?

2. Is it true that if \( P \) and \( Q \) are two CCS processes affording the same completed traces and \( L \) is a set of labels, then \( P \setminus L \) and \( Q \setminus L \) also have the same completed traces?

You should, of course, argue for your answers.

3.3 Strong bisimilarity

Our aim in this section will be to present one of the key notions in the theory of processes, namely strong bisimulation. In order to motivate this notion intuitively, let us reconsider once more the two processes \( CTM \) and \( CTM' \) that we used above to argue that trace equivalence is not a suitable notion of behavioural equivalence for reactive systems. The problem was that, as fully formalized in Exercise 3.2, the
3.3. STRONG BISIMILARITY

trace equivalent processes CTM and CTM’ exhibited different deadlock behaviour when made to interact with a third parallel process, namely CA. In hindsight, this is not overly surprising. In fact, when looking purely at the (completed) traces of a process, we focus only on the sequences of actions that the process may perform, but do not take into account the communication capabilities of the intermediate states that the process traverses as it computes. As the above example shows, the communication potential of the intermediate states does matter when we may interact with the process at all times. In particular, there is a crucial difference in the capabilities of the states reached by CTM and CTM’ after these processes have received a coin as input. Indeed, after accepting a coin the machine CTM always enters a state in which it is willing to output both coffee and tea, depending on what its user wants, whereas the machine CTM’ can only enter a state in which it is willing to deliver either coffee or tea, but not both.

The lesson that we may learn from the above discussion is that a suitable notion of behavioural relation between reactive systems should allow us to distinguish processes that may have different deadlock potential when made to interact with other processes. Such a notion of behavioural relation must take into account the communication capabilities of the intermediate states that processes may reach as they compute. One way to ensure that this holds is to require that in order for two processes to be equivalent, not only should they afford the same traces, but, in some formal sense, the states that they reach should still be equivalent. You can easily convince yourselves that trace equivalence does not meet this latter requirement, as the states that CTM and CTM’ may reach after receiving a coin as input are not trace equivalent.

The classic notion of strong bisimulation equivalence, introduced by David Park in (Park, 1981) and widely popularized by Robin Milner in (Milner, 1989), formalizes the informal requirements introduced above in a very elegant way.

Definition 3.2 [Strong bisimulation] A binary relation $\mathcal{R}$ over the set of states of an LTS is a bisimulation iff whenever $s_1 \mathcal{R} s_2$ and $\alpha$ is an action:

- if $s_1 \xrightarrow{\alpha} s'_1$, then there is a transition $s_2 \xrightarrow{\alpha} s'_2$ such that $s'_1 \mathcal{R} s'_2$;
- if $s_2 \xrightarrow{\alpha} s'_2$, then there is a transition $s_1 \xrightarrow{\alpha} s'_1$ such that $s'_1 \mathcal{R} s'_2$.

Two states $s$ and $s'$ are bisimilar, written $s \sim s'$, iff there is a bisimulation that relates them. Henceforth the relation $\sim$ will be referred to as strong bisimulation equivalence or strong bisimilarity. ♦

Since the operational semantics of CCS is given in terms of an LTS whose states are CCS process expressions, the above definition applies equally well to CCS
processes. Intuitively, a strong bisimulation is a kind of invariant relation between processes that is preserved by transitions in the sense of Definition 3.2.

Before beginning to explore the properties of strong bisimilarity, let us remark one of its most appealing features, namely a proof technique that it supports to show that two processes are strongly bisimilar. Since two processes are strongly bisimilar if there is a strong bisimulation that relates them, to prove that they are related by $\sim$ it suffices only to exhibit a strong bisimulation that relates them.

Example 3.1 Consider the labelled transition system

$$(\text{Proc}, \text{Act}, \{\overset{a}{\rightarrow} \mid a \in \text{Act}\})$$

where

- $\text{Proc} = \{s, s_1, s_2, t, t_1\}$,
- $\text{Act} = \{a, b\}$,
- $\overset{a}{\rightarrow} = \{(s, s_1), (s, s_2), (t, t_1)\}$, and
- $\overset{b}{\rightarrow} = \{(s_1, s_2), (s_2, s_2), (t_1, t_1)\}$.

Here is a graphical representation of this labelled transition system.

We will show that $s \sim t$. In order to do so, we have to define a strong bisimulation $\mathcal{R}$ such that $(s, t) \in \mathcal{R}$. Let us define it as

$$\mathcal{R} = \{(s, t), (s_1, t_1), (s_2, t_1)\}.$$ 

The binary relation $\mathcal{R}$ can be graphically depicted by dotted lines like in the following picture.
3.3. STRONG BISIMILARITY

Obviously, \((s, t) \in R\). We have to show that \(R\) is a strong bisimulation, i.e., that it meets the requirements stated in Definition 3.2. To this end, for each pair of states from \(R\), we have to investigate all the possible transitions from both states and see whether they can be matched by corresponding transitions from the other state. Note that a transition under some label can be matched only by a transition under the same label. We will now present a complete analysis of all of the steps needed to show that \(R\) is a strong bisimulation, even though they are very simple and tedious.

- Let us consider first the pair \((s, t)\):
  - transitions from \(s\):
    * \(s \xrightarrow{a} s_1\) can be matched by \(t \xrightarrow{a} t_1\) and \((s_1, t_1) \in R\),
    * \(s \xrightarrow{a} s_2\) can be matched by \(t \xrightarrow{a} t_1\) and \((s_2, t_1) \in R\), and
    * these are all the transitions from \(s\);
  - transitions from \(t\):
    * \(t \xrightarrow{a} t_1\) can be matched, e.g., by \(s \xrightarrow{a} s_2\) and \((s_2, t_1) \in R\) (another possibility would be to match it by \(s \xrightarrow{a} s_1\) but finding one matching transition is enough), and
    * this is the only transition from \(t\).

- Next we consider the pair \((s_1, t_1)\):
  - transitions from \(s_1\):
    * \(s_1 \xrightarrow{b} s_2\) can be matched by \(t_1 \xrightarrow{b} t_1\) and \((s_2, t_1) \in R\), and
    * this is the only transition from \(s_1\);
  - transitions from \(t_1\):
    * \(t_1 \xrightarrow{b} t_1\) can be matched by \(s_1 \xrightarrow{b} s_2\) and \((s_2, t_1) \in R\), and
    * this is the only transition from \(t_1\).

- Finally we consider the pair \((s_2, t_1)\):
  - transitions from \(s_2\):
    * \(s_2 \xrightarrow{b} s_2\) can be matched by \(t_1 \xrightarrow{b} t_1\) and \((s_2, t_1) \in R\), and
    * this is the only transition from \(s_2\);
  - transitions from \(t_1\):
    * \(t_1 \xrightarrow{b} t_1\) can be matched by \(s_2 \xrightarrow{b} s_2\) and \((s_2, t_1) \in R\), and
    * this is the only transition from \(t_1\).
This completes the proof that $\mathcal{R}$ is a strong bisimulation and, since $(s, t) \in \mathcal{R}$, we get that $s \sim t$.

In order to prove that, e.g., $s_1 \sim s_2$ we can use the following relation

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

The reader is invited to verify that $\mathcal{R}$ is indeed a strong bisimulation. ♦

**Example 3.2** In this example we shall demonstrate that it is possible for the initial state of a labelled transition system with infinitely many reachable states to be strongly bisimilar to a state from which only finitely many states are reachable. Consider the labelled transition system $(\text{Proc}, \text{Act}, \{ \overset{a}{\rightarrow} \mid a \in \text{Act} \})$ where

- $\text{Proc} = \{s_i \mid i \geq 1\} \cup \{t\}$,
- $\text{Act} = \{a\}$, and
- $\overset{a}{\rightarrow} = \{(s_i, s_{i+1}) \mid i \geq 1 \} \cup \{(t, t)\}$.

Here is a graphical representation of this labelled transition system.

![Graphical Representation](image)

We can now observe that $s_1 \sim t$ because the relation

$$\mathcal{R} = \{(s_i, t) \mid i \geq 1\}$$

is a strong bisimulation and it contains the pair $(s_1, t)$. The reader is invited to verify this simple fact. ♦

Consider now the two coffee and tea machines in our running example. We can argue that $\text{CTM}$ and $\text{CTM}'$ are not strongly bisimilar thus. Assume, towards a contradiction, that $\text{CTM}$ and $\text{CTM}'$ are strongly bisimilar. This means that there is a strong bisimulation $\mathcal{R}$ such that

$$\text{CTM} \mathcal{R} \text{CTM}' .$$

Recall that

$$\text{CTM}' \overset{\text{coin}}{\rightarrow} \text{tea}.\text{CTM}' .$$
So, by the second requirement in Definition 3.2, there must be a transition

\[ \text{CTM} \xrightarrow{\text{coin}} P \]

for some process \( P \) such that \( P \sim \text{tea}.\text{CTM} \). A moment of thought should be enough to convince yourselves that the only process that CTM can reach by receiving a coin as input is \( \text{coffee}.\text{CTM} + \text{tea}.\text{CTM} \). So we are requiring that

\[(\text{coffee}.\text{CTM} + \text{tea}.\text{CTM}) \sim \text{tea}.\text{CTM} \]

However, now a contradiction is immediately reached. In fact,

\[ \text{coffee}.\text{CTM} + \text{tea}.\text{CTM} \xrightarrow{\text{coffee}} \text{CTM} \]

but \( \text{tea}.\text{CTM} \) cannot output coffee. Thus the first requirement in Definition 3.2 cannot be met. It follows that our assumption that the two machines were strongly bisimilar leads to a contradiction. We may therefore conclude that, as claimed, the processes CTM and CTM’ are not strongly bisimilar.

**Example 3.3** Consider the processes \( P \) and \( Q \) defined thus:

\[
\begin{align*}
P & \overset{\text{def}}{=} a.P_1 + b.P_2 \\
P_1 & \overset{\text{def}}{=} c.P \\
P_2 & \overset{\text{def}}{=} c.P
\end{align*}
\]

and

\[
\begin{align*}
Q & \overset{\text{def}}{=} a.Q_1 + b.Q_2 \\
Q_1 & \overset{\text{def}}{=} c.Q_3 \\
Q_2 & \overset{\text{def}}{=} c.Q_3 \\
Q_3 & \overset{\text{def}}{=} a.Q_1 + b.Q_2
\end{align*}
\]

We claim that \( P \sim Q \). To prove that this does hold, it suffices to argue that the following relation is a strong bisimulation

\[ \mathcal{R} = \{(P, Q), (P, Q_3), (P_1, Q_1), (P_2, Q_2)\} \]

We encourage you to check that this is indeed the case. ♦
Exercise 3.3 Consider the processes $P$ and $Q$ defined thus:

\[
P \overset{\text{def}}{=} a.P_1 \quad P_1 \overset{\text{def}}{=} b.P + c.P
\]

and

\[
Q \overset{\text{def}}{=} a.Q_1 \\
Q_1 \overset{\text{def}}{=} b.Q_2 + c.Q \\
Q_2 \overset{\text{def}}{=} a.Q_3 \\
Q_3 \overset{\text{def}}{=} b.Q + c.Q_2 .
\]

Show that $P \sim Q$ holds by exhibiting an appropriate strong bisimulation.

Exercise 3.4 Consider the processes

\[
P \overset{\text{def}}{=} a.(b.0 + c.0) \quad \text{and} \quad Q \overset{\text{def}}{=} a.b.0 + a.c.0 .
\]

Show that $P$ and $Q$ are not strongly bisimilar.

Exercise 3.5 Consider the following labelled transition system.

Show that $s \sim t$ by finding a strong bisimulation $R$ containing the pair $(s, t)$.

Before looking at a few more examples, we now proceed to present some of the general properties of strong bisimilarity. In particular, we shall see that $\sim$ is an equivalence relation, and that it is preserved by all of the constructs in the CCS language.

The following result states the most basic properties of strong bisimilarity, and is our first theorem in this book.
Theorem 3.1  For all LTSs, the relation $\sim$ is

1. an equivalence relation,
2. the largest strong bisimulation, and
3. satisfies the following property:

$s_1 \sim s_2$ iff for each action $\alpha$,
- if $s_1 \xrightarrow{\alpha} s'_1$, then there is a transition $s_2 \xrightarrow{\alpha} s'_2$ such that $s'_1 \sim s'_2$;
- if $s_2 \xrightarrow{\alpha} s'_2$, then there is a transition $s_1 \xrightarrow{\alpha} s'_1$ such that $s'_1 \sim s'_2$.

Proof: Consider an LTS $(\text{Proc}, \text{Act}, \{\xrightarrow{\alpha} | \alpha \in \text{Act}\})$. We prove each of the statements in turn.

1. In order to show that $\sim$ is an equivalence relation over the set of states $\text{Proc}$, we need to argue that it is reflexive, symmetric and transitive. (See Definition 3.1.)

To prove that $\sim$ is reflexive, it suffices only to provide a bisimulation that contains the pair $(s, s)$, for each state $s \in \text{Proc}$. It is not hard to see that the identity relation

$I = \{(s, s) | s \in \text{Proc}\}$

is such a relation.

We now show that $\sim$ is symmetric. Assume, to this end, that $s_1 \sim s_2$ for some states $s_1$ and $s_2$ contained in $\text{Proc}$. We claim that $s_2 \sim s_1$ also holds. To prove this claim, recall that, since $s_1 \sim s_2$, there is a bisimulation $R$ that contains the pair of states $(s_1, s_2)$. Consider now the relation

$R^{-1} = \{(s', s) | (s, s') \in R\}$.

You should now be able to convince yourselves that the pair $(s_2, s_1)$ is contained in $R^{-1}$, and that this relation is indeed a bisimulation. Therefore $s_2 \sim s_1$, as claimed.

We are therefore left to argue that $\sim$ is transitive. Assume, to this end, that $s_1 \sim s_2$ and $s_2 \sim s_3$ for some states $s_1$, $s_2$, and $s_3$ contained in $\text{Proc}$. We claim that $s_1 \sim s_3$ also holds. To prove this, recall that, since $s_1 \sim s_2$ and $s_2 \sim s_3$, there are two bisimulations $R$ and $R'$ that contain the pairs of states $(s_1, s_2)$ and $(s_2, s_3)$, respectively. Consider now the relation

$S = \{(s'_1, s'_3) | (s'_1, s'_2) \in R \text{ and } (s'_2, s'_3) \in R', \text{ for some } s'_2\}$.
The pair \((s_1, s_3)\) is contained in \(S\). (Why?) Moreover, using that \(R\) and \(R'\) are bisimulations, you should be able to show that so is \(S\). Therefore \(s_1 \sim s_3\), as claimed.

2. We aim at showing that \(\sim\) is the largest strong bisimulation over the set of states \(\text{Proc}\). To this end, observe, first of all, that the definition of \(\sim\) states that

\[
\sim = \bigcup \{R \mid R \text{ is a bisimulation}\}.
\]

This yields immediately that each bisimulation is included in \(\sim\). We are therefore left to show that the right-hand side of the above equation is itself a bisimulation. This we now proceed to do.

Since we have already shown that \(\sim\) is symmetric, it is sufficient to prove that if

\[
(s_1, s_2) \in \bigcup \{R \mid R \text{ is a bisimulation}\} \quad \text{and} \quad s_1 \xrightarrow{\alpha} s'_1,
\]

then there is a state \(s'_2\) such that \(s_2 \xrightarrow{\alpha} s'_2\) and

\[
(s'_1, s'_2) \in \bigcup \{R \mid R \text{ is a bisimulation}\}.
\]

Assume, therefore, that (3.3) holds. Since

\[
(s_1, s_2) \in \bigcup \{R \mid R \text{ is a bisimulation}\},
\]

there is a bisimulation \(R\) that contains the pair \((s_1, s_2)\). As \(R\) is a bisimulation and \(s_1 \xrightarrow{\alpha} s'_1\), we have that there is a state \(s'_2\) such that \(s_2 \xrightarrow{\alpha} s'_2\) and \((s'_1, s'_2) \in R\). Observe now that the pair \((s'_1, s'_2)\) is also contained in

\[
\bigcup \{R \mid R \text{ is a bisimulation}\}.
\]

Hence, we have argued that there is a state \(s'_2\) such that \(s_2 \xrightarrow{\alpha} s'_2\) and

\[
(s'_1, s'_2) \in \bigcup \{R \mid R \text{ is a bisimulation}\},
\]

which was to be shown.

3. We now aim at proving that \(\sim\) satisfies the following property:

\(s_1 \sim s_2\) iff for each action \(\alpha\),

- if \(s_1 \xrightarrow{\alpha} s'_1\), then there is a transition \(s_2 \xrightarrow{\alpha} s'_2\) such that \(s'_1 \sim s'_2\);
- if \(s_2 \xrightarrow{\alpha} s'_2\), then there is a transition \(s_1 \xrightarrow{\alpha} s'_1\) such that \(s'_1 \sim s'_2\).
The implication from left to right is an immediate consequence of the fact that, as we have just shown, \( \sim \) is itself a bisimulation. We are therefore left to prove the implication from right to left. To this end, assume that \( s_1 \) and \( s_2 \) are two states in \( \text{Proc} \) having the following property:

\[
(*) \text{ for each action } \alpha, \\
\begin{align*}
\text{- if } s_1 & \xrightarrow{\alpha} s'_1, \text{ then there is a transition } s_2 \xrightarrow{\alpha} s'_2 \text{ such that } s'_1 \sim s'_2; \\
\text{- if } s_2 & \xrightarrow{\alpha} s'_2, \text{ then there is a transition } s_1 \xrightarrow{\alpha} s'_1 \text{ such that } s'_1 \sim s'_2.
\end{align*}
\]

We shall now prove that \( s_1 \sim s_2 \) holds by constructing a bisimulation that contains the pair \( (s_1, s_2) \).

How can we build the desired bisimulation \( R \)? First of all, we must add the pair \( (s_1, s_2) \) to \( R \) because we wish to use that relation to prove \( s_1 \sim s_2 \). Since \( R \) should be a bisimulation, each transition \( s_1 \xrightarrow{\alpha} s'_1 \) from \( s_1 \) should be matched by a transition \( s_2 \xrightarrow{\alpha} s'_2 \) from \( s_2 \), for some state \( s'_2 \) such that \( (s'_1, s'_2) \in \sim \). In light of the aforementioned property, this can be easily achieved by adding to the relation \( R \) all of the pairs of states contained in \( \sim \)!

Since we have already shown that \( \sim \) is itself a bisimulation, no more pairs of states need be added to \( R \).

The above discussion suggests that we consider the relation

\[
R = \{(s_1, s_2)\} \cup \sim.
\]

Indeed, by construction, the pair \( (s_1, s_2) \) is contained in \( R \). Moreover, using property \( (*) \) and statement 2 of the theorem, it is not hard to prove that \( R \) is a bisimulation. This shows that \( s_1 \sim s_2 \), as claimed.

The proof is now complete.

Exercise 3.6 Prove that the relations we have built in the proof of Theorem 3.1 are indeed bisimulations.

Exercise 3.7 In the proof of Theorem 3.1(2), we argued that the union of all of the bisimulation relations over an LTS is itself a bisimulation. Use the argument we adopted in the proof of that statement to show that the union of an arbitrary family of bisimulations is always a bisimulation.

Exercise 3.8 Is it true that any strong bisimulation must be reflexive, transitive and symmetric? If yes then prove it, if not then give counter-examples—that is
• define an LTS and a binary relation over states that is not reflexive but is a strong bisimulation;

• define an LTS and a binary relation over states that is not symmetric but is a strong bisimulation; and

• define an LTS and a binary relation over states that is not transitive but is a strong bisimulation.

Are the relations you have constructed the largest strong bisimulations over your labelled transition systems?

Exercise 3.9 (Recommended) A binary relation $R$ over the set of states of an LTS is a string bisimulation iff whenever $s_1 R s_2$ and $\sigma$ is a sequence of actions in $\text{Act}$:

- if $s_1 \xrightarrow{\sigma} s_1'$, then there is a transition $s_2 \xrightarrow{\sigma} s_2'$ such that $s_1' R s_2'$;

- if $s_2 \xrightarrow{\sigma} s_2'$, then there is a transition $s_1 \xrightarrow{\sigma} s_1'$ such that $s_1' R s_2'$.

Two states $s$ and $s'$ are string bisimilar iff there is a string bisimulation that relates them.

Prove that string bisimilarity and strong bisimilarity coincide. That is, show that two states $s$ and $s'$ are string bisimilar iff they are strongly bisimilar.

Exercise 3.10 Assume that the defining equation for the constant $K$ is $K \overset{\text{def}}{=} P$. Show that $K \sim P$ holds.

Exercise 3.11 Prove that two strongly bisimilar processes afford the same traces, and thus that strong bisimulation equivalence satisfies the requirement for a behavioural equivalence we set out in equation (3.1). Hint: Use your solution to Exercise 3.9 to show that, for each trace $\alpha_1 \cdots \alpha_k$ ($k \geq 0$),

$$P \sim Q \text{ and } \alpha_1 \cdots \alpha_k \in \text{Traces}(P) \text{ imply } \alpha_1 \cdots \alpha_k \in \text{Traces}(Q).$$

Is it true that strongly bisimilar processes have the same completed traces? (See Exercise 3.2 for the definition of the notion of completed trace.)

Exercise 3.12 (Recommended) Show that the relations listed below are strong bisimulations:

$$\{ \langle P | Q, Q | P \rangle \mid \text{where } P, Q \text{ are CCS processes} \}$$

$$\{ \langle P | 0, P \rangle \mid \text{where } P \text{ is a CCS process} \}$$

$$\{ \langle (P | Q) | R, P | (Q \mid R) \rangle \mid \text{where } P, Q, R \text{ are CCS processes} \}.$$
3.3. STRONG BISIMILARITY

Conclude that, for all $P, Q, R$,

$$P \parallel Q \sim Q \parallel P,$$  \hspace{1cm} (3.4)

$$P \parallel 0 \sim P,$$ \hspace{1cm} (3.5)

$$(P \parallel Q) \parallel R \sim P \parallel (Q \parallel R).$$ \hspace{1cm} (3.6)

Find three CCS processes $P, Q, R$ such that $(P + Q) \parallel R \not\sim (P \parallel R) + (Q \parallel R)$.

**Exercise 3.13** Is it true that, for all CCS processes $P$ and $Q$,

$$(P \parallel Q) \setminus a \sim (P \setminus a) \parallel (Q \setminus a)?$$

Does the following equivalence hold for all CCS processes $P$ and $Q$, and relabelling function $f$?

$$(P \parallel Q)[f] \sim (P[f]) \parallel (Q[f]).$$

If your answer to the above questions is positive, then construct appropriate bisimulations. Otherwise, provide a counter-example to the claim.

As we saw in Exercise 3.12, parallel composition is associative and commutative modulo strong bisimilarity. Therefore, since the precise bracketing of terms in a parallel composition does not matter, we can use the notation $\Pi_{i=1}^{k} P_i$, where $k \geq 0$ and the $P_i$ are CCS processes, to stand for

$$P_1 \parallel P_2 \parallel \cdots \parallel P_k.$$

If $k = 0$, then, by convention, the above term is just $0$.

As mentioned before, one of the desirable properties for a notion of behavioural equivalence is that it should allow us to ‘replace equivalent processes for equivalent processes’ in any larger process expression without affecting its behaviour. The following theorem states that this is indeed possible for strong bisimilarity.

**Theorem 3.2** Let $P, Q, R$ be CCS processes. Assume that $P \sim Q$. Then

- $\alpha.P \sim \alpha.Q$, for each action $\alpha$;
- $P + R \sim Q + R$ and $R + P \sim R + Q$, for each process $R$;
- $P \parallel R \sim Q \parallel R$ and $R \parallel P \sim R \parallel Q$, for each process $R$;
- $P[f] \sim Q[f]$, for each relabelling $f$; and
- $P \setminus L \sim Q \setminus L$, for each set of labels $L$. 

Proof: We limit ourselves to showing that $\sim$ is preserved by parallel composition and restriction. We consider these two operations in turn. In both cases, we assume that $P \sim Q$.

- Let $R$ be a CCS process. We aim at showing that $P | R \sim Q | R$. To this end, we shall build a bisimulation $\mathcal{R}$ that contains the pair of processes $(P | R, Q | R)$.

Consider the relation

$$\mathcal{R} = \{(P' | R', Q' | R') \mid P' \sim Q' \text{ and } P', Q', R' \text{ are CCS processes}\}.$$

You should readily be able to convince yourselves that the pair of processes $(P | R, Q | R)$ is indeed contained in $\mathcal{R}$, and thus that all we are left to do to complete our argument is to show that $\mathcal{R}$ is a bisimulation. The proof of this fact will, hopefully, also highlight that the above relation $\mathcal{R}$ was not ‘built out of thin air’, and will epitomize the creative process that underlies the building of bisimulation relations.

First of all, observe that, by symmetry, to prove that $\mathcal{R}$ is a bisimulation, it is sufficient to argue that if $(P' | R', Q' | R')$ is contained in $\mathcal{R}$ and $P' | R' \overset{\alpha}{\rightarrow} S$ for some action $\alpha$ and CCS process $S$, then $Q' | R' \overset{\alpha}{\rightarrow} T$ for some CCS process $T$ such that $(S, T) \in \mathcal{R}$. This we now proceed to do.

Assume that $(P' | R', Q' | R')$ is contained in $\mathcal{R}$ and $P' | R' \overset{\alpha}{\rightarrow} S$ for some action $\alpha$ and CCS process $S$. We now proceed with the proof by a case analysis on the possible origins of the transition $P' | R' \overset{\alpha}{\rightarrow} S$. Recall that the transition we are considering must be provable using the SOS rules for parallel composition given in Table 2.2 on page 29. Therefore there are three possible forms that the transition $P' | R' \overset{\alpha}{\rightarrow} S$ may take, namely:

1. $P'$ is responsible for the transition and $R'$ ‘stands still’—that is,

$$P' | R' \overset{\alpha}{\rightarrow} S$$

because, by rule COM1 for parallel composition in Table 2.2, $P' \overset{\alpha}{\rightarrow} P''$ and $S = P'' | R'$, for some $P''$.

2. $R'$ is responsible for the transition and $P'$ ‘stands still’—that is,

$$P' | R' \overset{\alpha}{\rightarrow} S$$

because, by rule COM2 for parallel composition in Table 2.2, $R' \overset{\alpha}{\rightarrow} R''$ and $S = P' | R''$, for some $R''$, or
3. the transition under consideration is the result of a synchronization between a transition of $P'$ and one of $R'$—that is,

$$P' \mid R' \xrightarrow{\alpha} S$$

because, by rule COM3 for parallel composition in Table 2.2, $\alpha = \tau$, and there are a label $a$ and processes $P''$ and $R''$ such that $P' \xrightarrow{a} P''$, $R' \xrightarrow{\bar{a}} R''$ and $S = P'' \mid R''$.

We now proceed by examining each of these possibilities in turn.

1. Since $P' \xrightarrow{\alpha} P''$ and $P' \sim Q'$, we have that $Q' \xrightarrow{\alpha} Q''$ and $P'' \sim Q''$, for some $Q''$. Using the transition $Q' \xrightarrow{\alpha} Q''$ as premise in rule COM1 for parallel composition in Table 2.2 on page 29, we can infer that

$$Q' \mid R' \xrightarrow{\alpha} Q'' \mid R'.$$

By the definition of the relation $\mathcal{R}$, we have that

$$(P'' \mid R', Q'' \mid R') \in \mathcal{R}.$$ We can therefore take $T = Q'' \mid R'$, and we are done.

2. In this case, we have that $R' \xrightarrow{\alpha} R''$. Using this transition as premise in rule COM2 for parallel composition in Table 2.2 on page 29, we can infer that

$$Q' \mid R' \xrightarrow{\alpha} Q' \mid R''.$$

By the definition of the relation $\mathcal{R}$, we have that

$$(P' \mid R'', Q' \mid R'') \in \mathcal{R}.$$ We can therefore take $T = Q' \mid R''$, and we are done.

3. Since $P' \xrightarrow{\alpha} P''$ and $P' \sim Q'$, we have that $Q' \xrightarrow{\alpha} Q''$ and $P'' \sim Q''$, for some $Q''$. Using the transitions $Q' \xrightarrow{\alpha} Q''$ and $R' \xrightarrow{\bar{a}} R''$ as premises in rule COM3 for parallel composition in Table 2.2 on page 29, we can infer that

$$Q' \mid R' \xrightarrow{\tau} Q'' \mid R''.$$

By the definition of the relation $\mathcal{R}$, we have that

$$(P'' \mid R'', Q'' \mid R'') \in \mathcal{R}.$$ We can therefore take $T = Q'' \mid R''$, and we are done.
Therefore the relation $\mathcal{R}$ is a bisimulation, as claimed.

- Let $L$ be a set of labels. We aim at showing that $P \setminus L \sim Q \setminus L$. To this end, we shall build a bisimulation $\mathcal{R}$ that contains the pair of processes $(P \setminus L, Q \setminus L)$.

Consider the relation

$$\mathcal{R} = \{ (P', Q') \mid P' \sim Q' \text{ and } P', Q' \text{ are CCS processes} \}.$$  

You should readily be able to convince yourselves that the pair of processes $(P \setminus L, Q \setminus L)$ is indeed contained in $\mathcal{R}$. Moreover, following the lines of the proof we have just gone through for parallel composition, it is an instructive exercise to show that

- the relation $\mathcal{R}$ is symmetric, and
- if $(P', Q') \in \mathcal{R}$ and $P' \setminus L \xrightarrow{\alpha} S$ for some action $\alpha$ and CCS process $S$, then $Q' \setminus L \xrightarrow{\alpha} T$ for some CCS process $T$ such that $(S, T) \in \mathcal{R}$.

You are strongly encouraged to fill in the missing details in the proof. $\square$

**Exercise 3.14** Prove that $\sim$ is preserved by action prefixing, summation and relabelling.

**Exercise 3.15 (For the theoretically minded)**  For each set of labels $L$ and process $P$, we may wish to build the process $\tau_L(P)$ that is obtained by turning into a $\tau$ each action $\alpha$ performed by $P$ with $\alpha \in L$ or $\bar{\alpha} \in L$. Operationally, the behaviour of the construct $\tau_L(\ )$ can be described by the following two rules.

$$\frac{P \xrightarrow{\alpha} P'}{\tau_L(P) \xrightarrow{\tau} \tau_L(P')} \quad \text{if } \alpha \in L \text{ or } \bar{\alpha} \in L$$

$$\frac{P \xrightarrow{\alpha} P'}{\tau_L(P) \xrightarrow{\tau} \tau_L(P')} \quad \text{if } \alpha = \tau \text{ or } \alpha, \bar{\alpha} \notin L$$

Prove that $\tau_L(P) \sim \tau_L(Q)$, whenever $P \sim Q$.

Consider the question of whether the operation $\tau_L(\ )$ can be defined in CCS modulo $\sim$—that is, can you find a CCS expression $C_L[\ ]$ with a ‘hole’ (a placeholder when another process can be plugged) such that, for each process $P$,

$$\tau_L(P) \sim C_L[P] ?$$

Argue for your answer.
3.3. **STRONG BISIMILARITY**

Recall that we defined the specification of a counter thus:

\[
\begin{align*}
\text{Counter}_0 & \overset{\text{def}}{=} \text{up.}	ext{Counter}_1 \\
\text{Counter}_n & \overset{\text{def}}{=} \text{up.}	ext{Counter}_{n+1} + \text{down.}	ext{Counter}_{n-1} \quad (n > 0).
\end{align*}
\]

Moreover, we stated that we expect that process to be ‘behaviourally equivalent’ to the process \(C\) defined by

\[
C \overset{\text{def}}{=} \text{up.}(C \mid \text{down.}0).
\]

We can now show that, in fact, \(C\) and \(\text{Counter}_0\) are strongly bisimilar. To this end, note that this follows if we can show that the relation \(R\) below

\[
\{(C \mid \Pi_{i=1}^k P_i, \text{Counter}_n) \mid (1) \ k \geq 0,
\quad (2) P_i = 0 \text{ or } P_i = \text{down.}0, \text{ for each } i,
\quad (3) \text{the number of } i \text{ with } P_i = \text{down.}0 \text{ is } n\}
\]

is a strong bisimulation. (Can you see why?) The following result states that this does hold true.

**Proposition 3.1** The relation \(R\) defined above is a strong bisimulation.

**Proof:** Assume that

\[
(C \mid \Pi_{i=1}^k P_i) \ R \text{ Counter}_n.
\]

By the definition of the relation \(R\), each \(P_i\) is either \(0\) or \(\text{down.}0\), and the number of \(P_i = \text{down.}0\) is \(n\). We shall now show that

1. if \(C \mid \Pi_{i=1}^k P_i \overset{\alpha}{\rightarrow} P\) for some action \(\alpha\) and process \(P\), then there is some process \(Q\) such that \(\text{Counter}_n \overset{\alpha}{\rightarrow} Q\) and \(P \ R \ Q\), and
2. if \(\text{Counter}_n \overset{\alpha}{\rightarrow} Q\) for some some action \(\alpha\) and process \(Q\), then there is some process \(P\) such that \(C \mid \Pi_{i=1}^k P_i \overset{\alpha}{\rightarrow} P\) and \(P \ R \ Q\).

We establish these two claims separately.

1. Assume that \(C \mid \Pi_{i=1}^k P_i \overset{\alpha}{\rightarrow} P\) for some some action \(\alpha\) and process \(P\). Then
   
   - either \(\alpha = \text{up}\) and \(P = C \mid \text{down.}0 \mid \Pi_{i=1}^k P_i\)
   - or \(n > 0\), \(\alpha = \text{down}\) and \(P = C \mid \Pi_{i=1}^k P_i\), where the vectors of processes \((P_1, \ldots, P_k)\) and \((P'_1, \ldots, P'_k)\) differ in exactly one position \(\ell\), and at that position \(P_\ell = \text{down.}0\) and \(P'_\ell = 0\).
In the former case, argue that the matching transition is
\[ \text{Counter}_n \xrightarrow{\text{up}} \text{Counter}_{n+1} . \]

In the latter, argue that the matching transition is
\[ \text{Counter}_n \xrightarrow{\text{down}} \text{Counter}_{n-1} . \]

2. Assume that \( \text{Counter}_n \xrightarrow{\alpha} Q \) for some some action \( \alpha \) and process \( Q \). Then
   - either \( \alpha = \text{up} \) and \( Q = \text{Counter}_{n+1} \)
   - or \( n > 0 \), \( \alpha = \text{down} \) and \( Q = \text{Counter}_{n-1} \).

Finding matching transitions from \( C | \Pi_{i=1}^{k} P_i \) is left as an exercise for the reader.

We can therefore conclude that \( \mathcal{R} \) is a strong bisimulation, which was to be shown.

\( \square \)

**Exercise 3.16** Fill in the missing details in the above proof.

Using CCS, we may specify the desired behaviour of a buffer with capacity one thus:
\[
\begin{align*}
B_0^1 & \overset{\text{def}}{=} \text{in.} B_1^1 \\
B_1^1 & \overset{\text{def}}{=} \overline{\text{out.}} B_0^1 .
\end{align*}
\]
The constant \( B_0^1 \) stands for an empty buffer with capacity one—that is a buffer with capacity one holding zero items—, and \( B_1^1 \) stands for a full buffer with capacity one—that is a buffer with capacity one holding one item.

By analogy with the above definition, in general we may specify a buffer of capacity \( n \geq 1 \) as follows, where the superscript stands for the maximal capacity of the buffer and the subscript for the number of elements the buffer currently holds:
\[
\begin{align*}
B_0^n & \overset{\text{def}}{=} \text{in.} B_1^n \\
B_i^n & \overset{\text{def}}{=} \text{in.} B_i^{n+1} + \overline{\text{out.}} B_{i-1}^n \quad \text{for } 0 < i < n \\
B_n^n & \overset{\text{def}}{=} \overline{\text{out.}} B_{n-1}^n .
\end{align*}
\]

It seems natural to expect that we may implement a buffer of capacity \( n \geq 1 \) by means of the parallel composition of \( n \) buffers of capacity one. This expectation is
certainly met when \( n = 2 \) because, as you can readily check, the relation depicted in Figure 3.2 is a bisimulation showing that

\[
B_0^2 \sim B_0^1 \mid B_0^1 .
\]

That this holds regardless of the size of the buffer to be implemented is the import of the following result.

**Proposition 3.2** For each natural number \( n \geq 1 \),

\[
B_0^n \sim \underbrace{B_0^1 \mid B_0^1 \mid \cdots \mid B_0^1}_{n \text{ times}}.
\]

**Proof:** Construct the following binary relation, where \( i_1, i_2, \ldots, i_n \in \{0, 1\} \):

\[
\mathcal{R} = \{ (B_{i_1}^n, B_{i_2}^1 \mid B_{i_2}^1 \mid \cdots \mid B_{i_n}^1) \mid \sum_{j=1}^{n} i_j = i \} .
\]

Intuitively, the above relation relates a buffer of capacity \( n \) holding \( i \) items with a parallel composition of \( n \) buffers of capacity one, provided that exactly \( i \) of them are full.

It is not hard to see that

- \((B_0^n, B_0^1 \mid B_0^1 \mid \cdots \mid B_0^1) \in \mathcal{R}\), and
- \(\mathcal{R}\) is a strong bisimulation.

It follows that

\[
B_0^n \sim \underbrace{B_0^1 \mid B_0^1 \mid \cdots \mid B_0^1}_{n \text{ times}},
\]

which was to be shown. We encourage you to fill in the details in this proof. \(\square\)
Exercise 3.17 (Simulation) Let us say that a binary relation $\mathcal{R}$ over the set of states of an LTS is a simulation iff whenever $s_1 \mathcal{R} s_2$ and $\alpha$ is an action:

- if $s_1 \xrightarrow{\alpha} s'_1$, then there is a transition $s_2 \xrightarrow{\alpha} s'_2$ such that $s'_1 \mathcal{R} s'_2$.

We say that $s'$ simulates $s$, written $s \sqsubseteq s'$, iff there is a simulation $\mathcal{R}$ with $s \mathcal{R} s'$. Two states $s$ and $s'$ are simulation equivalent, written $s \simeq s'$, iff $s \sqsubseteq s'$ and $s' \sqsubseteq s$ both hold.

1. Prove that $\sqsubseteq$ is a preorder and $\simeq$ is an equivalence relation.

2. Build simulations showing that

$$a.0 \sqsubseteq a.a.0 \quad \text{and} \quad a.b.0 + a.c.0 \sqsubseteq a.(b.0 + c.0)$$

Do the converse relations hold?

3. Show that strong bisimilarity is included in simulation equivalence—that is, that for any two strongly bisimilar states $s$ and $s'$ it holds that $s'$ simulates $s$. Does the converse inclusion also hold?

Is there a CCS process that can simulate any other CCS process?

Exercise 3.18 (Ready Simulation) Let us say that a binary relation $\mathcal{R}$ over the set of states of an LTS is a ready simulation iff whenever $s_1 \mathcal{R} s_2$ and $\alpha$ is an action:

- if $s_1 \xrightarrow{\alpha} s'_1$, then there is a transition $s_2 \xrightarrow{\alpha} s'_2$ such that $s'_1 \mathcal{R} s'_2$; and
- if $s_2 \xrightarrow{\alpha}$, then $s_1 \xrightarrow{\alpha}$.

We say that $s'$ ready simulates $s$, written $s \sqsubseteq_{RS} s'$, iff there is a ready simulation $\mathcal{R}$ with $s \mathcal{R} s'$. Two states $s$ and $s'$ are ready simulation equivalent, written $s \simeq_{RS} s'$, iff $s \sqsubseteq_{RS} s'$ and $s' \sqsubseteq_{RS} s$ both hold.

1. Prove that $\sqsubseteq_{RS}$ is a preorder and $\simeq_{RS}$ is an equivalence relation.

2. Do the following relations hold?

$$a.0 \sqsubseteq_{RS} a.a.0 \quad \text{and} \quad a.b.0 + a.c.0 \sqsubseteq_{RS} a.(b.0 + c.0)$$

3. Show that strong bisimilarity is included in ready simulation equivalence—that is, that for any two strongly bisimilar states $s$ and $s'$ it holds that $s'$ ready simulates $s$. Does the converse inclusion also hold?
3.4. WEAK BISIMILARITY

Is there a CCS process that can ready simulate any other CCS process?

Exercise 3.19 (For the theoretically minded) Consider the processes

\[ P \overset{\text{def}}{=} a.b.c.0 + a.b.d.0 \quad \text{and} \quad Q \overset{\text{def}}{=} a.(b.c.0 + b.d.0). \]

Argue, first of all, that \( P \) and \( Q \) are not strongly bisimilar. Next show that:

1. \( P \) and \( Q \) have the same completed traces (see Exercise 3.2);
2. for each process \( R \) and set of labels \( L \), the processes \( (P \mid R) \setminus L \) and \( (Q \mid R) \setminus L \) have the same completed traces.

So \( P \) and \( Q \) have the same deadlock behaviour in all parallel contexts, even though strong bisimilarity distinguishes them.

The lesson to be learned from these observations is that more generous notions of behavioural equivalence than bisimilarity may be necessary to validate some desirable equivalences.

3.4 Weak bisimilarity

As we have seen in the previous section, strong bisimilarity affords many of the properties that we expect a notion of behavioural relation to be used in implementation verification to have. (See the introduction to Chapter 3.) In particular, strong bisimilarity is an equivalence relation that is preserved by all of the CCS operators, it is the largest strong bisimulation, supports a very elegant proof technique to demonstrate equivalences between process descriptions, and it suffices to establish several natural equivalences. For instance, you used strong bisimilarity in Exercise 3.12 to justify the expected equalities

\[ P \mid Q \sim Q \mid P, \quad P \mid 0 \sim P, \quad \text{and} \]
\[ (P \mid Q) \mid R \sim P \mid (Q \mid R). \]

Moreover, a wealth of other ‘structural equivalences’ like the ones above may be proven to hold modulo strong bisimilarity. (See (Milner, 1989, Propositions 7–8).)
CHAPTER 3. BEHAVIOURAL EQUIVALENCE

Should we look any further for a notion of behavioural equivalence to support implementation verification? Is there any item on our wish list that is not met by strong bisimilarity?

You might recall that we stated early on in this book that \( \tau \) actions in process behaviours are supposed to be internal, and thus unobservable. This is a natural consequence of Milner’s design decision to let \( \tau \) indicate the result of a successful communication between two processes. Since communication is binary in CCS, and observing the behaviour of a process means communicating with it in some fashion, the unobservable nature of \( \tau \) actions is the upshot of the assumption that they cannot be used for further communication. This discussion indicates that a notion of behavioural equivalence should allow us to abstract from such steps in process behaviours.

Consider, for instance, the processes \( a.\tau.0 \) and \( a.0 \). Since \( \tau \) actions should be unobservable, we intuitively expect these to be observationally equivalent. Unfortunately, however, the processes \( a.\tau.0 \) and \( a.0 \) are not strongly bisimilar. In fact, the definition of strong bisimulation requires that each transition in the behaviour of one process should be matched by one transition of the other, regardless of whether that transition is labelled by an observable action or \( \tau \), and \( a.\tau.0 \) affords the trace \( a\tau \), whereas \( a.0 \) does not.

In hindsight, this failure of strong bisimilarity to account for the unobservable nature of \( \tau \) actions is expected because the definition of strong bisimulation treats internal actions as if they were ordinary observable actions. What we should like to have is a notion of bisimulation equivalence that affords all of the good properties of strong bisimilarity, and abstracts from \( \tau \) actions in the behaviour of processes. However, in order to fulfill this aim, first we need to understand what ‘abstracting from \( \tau \) actions’ actually means. Does this simply mean that we can ‘erase’ all of the \( \tau \) actions in the behaviour of a process? This would be enough to show that \( a.\tau.0 \) and \( a.0 \) are equivalent, as the former process is identical to the latter if we ‘erase the \( \tau \) prefix’. But would this work in general?

To understand the issue better, let us make our old friend from the computer science department, namely the process CS defined in Table 2.1 on page 18 interact with a nasty variation on the coffee machine CM from equation 2.1 on page 11. This latest version of the coffee machine delivered to the computer scientist’s office is given by

\[
CM_b \triangleq \text{coin.co}\text{ffee.CM}_b + \text{coin.CM}_b.
\]  
(3.7)

(The subscript ‘b’ indicates that this version of the coffee machine is bad!)

Note that, upon receipt of a coin, the coffee machine \( CM_b \) can decide to go back to its initial state without delivering the coffee. You should be able to convince yourselves that the sequences of transitions in Table 3.1 describe the possible
3.4. **WEAK BISIMILARITY**

![Diagram showing weak bisimilarity](image)

where

\[
\begin{align*}
\text{Start} & \equiv (\text{CM}_b \mid \text{CS}) \setminus \{\text{coin, coffee}\} & \text{CS} & \equiv \overline{\text{pub}} \cdot \text{CS}_1 \\
\text{Good} & \equiv (\text{coffee} \cdot \text{CM}_b \mid \text{CS}_2) \setminus \{\text{coin, coffee}\} & \text{CS}_1 & \equiv \overline{\text{coin}} \cdot \text{CS}_2 \\
\text{Bad} & \equiv (\text{CM}_b \mid \text{CS}_2) \setminus \{\text{coin, coffee}\} & \text{CS}_2 & \equiv \text{coffee} \cdot \text{CS} .
\end{align*}
\]

Table 3.1: The behaviour of \((\text{CM}_b \mid \text{CS}) \setminus \{\text{coin, coffee}\}\)

behaviours of the system \((\text{CM}_b \mid \text{CS}) \setminus \{\text{coin, coffee}\}\). In that table, for the sake of notational convenience, we use `Start` as a short-hand for the CCS expression

\[(\text{CM}_b \mid \text{CS}) \setminus \{\text{coin, coffee}\} .\]

The short-hands `Bad` and `Good` are also introduced in the picture using the ‘declarations’

\[
\begin{align*}
\text{Good} & \equiv (\text{coffee} \cdot \text{CM}_b \mid \text{CS}_2) \setminus \{\text{coin, coffee}\} \quad \text{and} \\
\text{Bad} & \equiv (\text{CM}_b \mid \text{CS}_2) \setminus \{\text{coin, coffee}\} .
\end{align*}
\]

Note that, there are two possible \(\tau\)-transitions that stem from the process

\[(\text{CM}_b \mid \text{CS}_1) \setminus \{\text{coin, coffee}\} ,\]

and that one of them, namely

\[(\text{CM}_b \mid \text{CS}_1) \setminus \{\text{coin, coffee}\} \xrightarrow{\tau} (\text{CM}_b \mid \text{CS}_2) \setminus \{\text{coin, coffee}\} ,\]

leads to a deadlocked state. Albeit directly unobservable, this transition cannot be ignored in our analysis of the behaviour of this system because it pre-empts the other possible behaviour of the machine. So, unobservable actions cannot be just
erased from the behaviour of processes because, in light of their pre-emptive power in the presence of nondeterministic choices, they may affect what we may observe.

Note that the pre-emptive power of internal transitions is unimportant in the standard theory of automata as there we are only concerned with the possibility of processing our input strings correctly. Indeed, as you may recall from your courses in the theory of automata, the so-called \( \varepsilon \)-transitions do not increase the expressive power of nondeterministic finite automata—see, for instance, the textbook (Sipser, 2005, Chapter 1). In a reactive environment, on the other hand, this power of internal transitions must be taken into account in a reasonable definition of process behaviour because it may lead to undesirable consequences, e.g., the deadlock situation in the above example. We therefore expect that the behaviour of the process \( \text{SmUni} \) is not equivalent to that of the process \( (\text{CM}_b \mid \text{CS}) \setminus \{\text{coin, coffee}\} \) since the latter may deadlock after outputting a publication, whereas the former cannot.

In order to define a notion of bisimulation that allows us to abstract from internal transitions in process behaviours, and to differentiate the process \( \text{SmUni} \) from \( (\text{CM}_b \mid \text{CS}) \setminus \{\text{coin, coffee}\} \), we begin by introducing a new notion of transition relation between processes.

**Definition 3.3** Let \( P \) and \( Q \) be CCS processes, or, more generally, states in an LTS. For each action \( \alpha \), we shall write \( P \xrightarrow{\alpha} Q \) iff either

- \( \alpha \neq \tau \) and there are processes \( P' \) and \( Q' \) such that 
  \[ P(\tau)^* P' \xrightarrow{\alpha} Q'(\tau)^* Q \]

- or \( \alpha = \tau \) and \( P(\tau)^* Q \),

where we write \( (\tau)^* \) for the reflexive and transitive closure of the relation \( \tau \). ♦

Thus \( P \xrightarrow{\alpha} Q \) holds if \( P \) can reach \( Q \) by performing an \( \alpha \)-labelled transition, possibly preceded and followed by sequences of \( \tau \)-labelled transitions. For example, \( a.\tau.0 \xrightarrow{0} 0 \) and \( a.\tau.0 \xrightarrow{\tau.0} \) both hold, as well as \( a.\tau.0 \xrightarrow{\tau} a.\tau.0 \). In fact, we have \( P \xrightarrow{\tau} P \) for each process \( P \).

In the LTS depicted in Table 3.1, apart from the obvious one step \( \text{pub} \)-labelled transition, we have that

- \( \text{Start} \xrightarrow{\text{pub}} \text{Good} \),
- \( \text{Start} \xrightarrow{\text{pub}} \text{Bad} \), and
- \( \text{Start} \xrightarrow{\text{pub}} \text{Start} \).
Our order of business will now be to use the new transition relations presented above to define a notion of bisimulation that can be used to equate processes that offer the same observable behaviour despite possibly having very different amounts of internal computations. The idea underlying the definition of the new notion of bisimulation is that a transition of a process can now be matched by a sequence of transitions from the other that has the same ‘observational content’ and leads to a state that is bisimilar to that reached by the first process.

**Definition 3.4** [Weak bisimulation and observational equivalence] A binary relation $R$ over the set of states of an LTS is a weak bisimulation iff whenever $s_1 R s_2$ and $\alpha$ is an action (including $\tau$):

- if $s_1 \xrightarrow{\alpha} s_1'$ then there is a transition $s_2 \xrightarrow{\alpha} s_2'$ such that $s_1' R s_2'$;
- if $s_2 \xrightarrow{\alpha} s_2'$ then there is a transition $s_1 \xrightarrow{\alpha} s_1'$ such that $s_1' R s_2'$.

Two states $s$ and $s'$ are observationally equivalent (or weakly bisimilar), written $s \approx s'$, iff there is a weak bisimulation that relates them. Henceforth the relation $\approx$ will be referred to as observational equivalence or weak bisimilarity.

**Example 3.4** Let us consider the following labelled transition system.

$$
\xymatrix{ s \xrightarrow{\tau} s_1 \xrightarrow{a} s_2 & t \xrightarrow{a} t_1 }
$$

Obviously $s \not\approx t$. On the other hand $s \approx t$ because the relation

$$\mathcal{R} = \{(s,t), (s_1, t), (s_2, t_1)\}$$

is a weak bisimulation such that $(s, t) \in \mathcal{R}$. It remains to verify that $\mathcal{R}$ is indeed a weak bisimulation.

- Let us examine all possible transitions from the components of the pair $(s, t)$. If $s \xrightarrow{\tau} s_1$ then $t \xrightarrow{a} t$ and $(s_1, t) \in \mathcal{R}$. If $t \xrightarrow{a} t_1$ then $s \xrightarrow{a} s_2$ and $(s_2, t_1) \in \mathcal{R}$.
- Let us examine all possible transitions from $(s_1, t)$. If $s_1 \xrightarrow{a} s_2$ then $t \xrightarrow{a} t_1$ and $(s_2, t_1) \in \mathcal{R}$. Similarly if $t \xrightarrow{a} t_1$ then $s_1 \xrightarrow{a} s_2$ and again $(s_2, t_1) \in \mathcal{R}$.
- Consider now the pair $(s_2, t_1)$. Since neither $s_2$ nor $t_1$ can perform any transition, it is safe to have this pair in $\mathcal{R}$.

Hence we have shown that each pair from $\mathcal{R}$ satisfies the conditions given in Definition 3.4, which means that $\mathcal{R}$ is a weak bisimulation, as claimed.
We can readily argue that $a.0 \approx a.\tau.0$ by establishing a weak bisimulation that relates these two processes. (Do so by renaming the states in the labelled transition system and in the bisimulation above!) On the other hand, there is no weak bisimulation that relates the process SmUni and the process Start in Table 3.1. In fact, the process SmUni is observationally equivalent to the process

$$\text{Spec} \overset{\text{def}}{=} \text{pub.Spec}$$

but the process Start is not.

**Exercise 3.20** Prove the claims that we have just made.

**Exercise 3.21** Prove that the behavioural equivalences claimed in Exercise 2.11 hold with respect to observational equivalence (weak bisimilarity).

The definition of weak bisimilarity is so natural, at least to our mind, that it is easy to miss some of its crucial consequences. To highlight some of these, consider the process

$$A? \overset{\text{def}}{=} a.0 + \tau.B?$$

$$B? \overset{\text{def}}{=} b.0 + \tau.A?$$

Intuitively, this process describes a ‘polling loop’ that may be seen as an implementation of a process that is willing to receive on port $a$ and port $b$, and then terminate. Indeed, it is not hard to show that

$$A? \approx B? \approx a.0 + b.0$$

(Prove this!) This seems to be non-controversial until we note that $A?$ and $B?$ have a livelock (that is, a possibility of divergence) due to the $\tau$-loop

$$A? \xrightarrow{\tau} B? \xrightarrow{\tau} A?$$

but $a.0 + b.0$ does not. The above equivalences capture one of the main features of observational equivalence, namely the fact that it supports what is called ‘fair abstraction from divergence’. (See (Baeten, Bergstra and Klop, 1987), where Baeten, Bergstra and Klop show that a proof rule embodying this idea, namely Koomen’s fair abstraction rule, is valid with respect to observational equivalence.) This means that observational equivalence assumes that if a process can escape from a loop consisting of internal transitions, then it will eventually do so. This property of observational equivalence, that is by no means obvious from its definition, is crucial in using it as a correctness criterion in the verification of communication protocols,
3.4. **WEAK BISIMILARITY**

<table>
<thead>
<tr>
<th>Send</th>
<th>[\text{def} = \text{acc}.\text{Sending}]</th>
<th>Rec</th>
<th>[\text{def} = \text{trans}.\text{Del}]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sending</td>
<td>[\text{def} = \text{send}.\text{Wait}]</td>
<td>Del</td>
<td>[\text{def} = \text{del}.\text{Ack}]</td>
</tr>
<tr>
<td>Wait</td>
<td>[\text{def} = \text{ack}.\text{Send} + \text{error}.\text{Sending}]</td>
<td>Ack</td>
<td>[\text{def} = \text{ack}.\text{Rec}]</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{Med} & \overset{\text{def}}{=} \text{send}.\text{Med}' \\
\text{Med}' & \overset{\text{def}}{=} \tau.\text{Err} + \text{trans}.\text{Med} \\
\text{Err} & \overset{\text{def}}{=} \text{error}.\text{Med}
\end{align*}
\]

Table 3.2: The sender, receiver and medium in (3.8)

where the communication media may lose messages, and messages may have to be retransmitted some arbitrary number of times in order to ensure their delivery.

Note moreover that 0 is observationally equivalent to the process

\[
\text{Div} \overset{\text{def}}{=} \tau.\text{Div}.
\]

This means that a process that can only diverge is observationally equivalent to a deadlocked one. This may also seem odd at first sight. However, you will probably agree that, assuming that we can only observe a process by communicating with it, the systems 0 and Div are observationally equivalent since both refuse each attempt at communicating with them. (They do so for different reasons, but these reasons cannot be distinguished by an external observer.)

As an example of an application of observational equivalence to the verification of a simple protocol, consider the process Protocol defined by

\[
(Send | Med | Rec) \setminus L \quad (L = \{\text{send}, \text{error}, \text{trans}, \text{ack}\})
\]

consisting of a sender and a receiver that communicate via a potentially faulty medium. The sender, the receiver and the medium are given in Table 3.2. (In that table, we use the port names acc and del as short-hands for ‘accept’ and ‘deliver’, respectively.) Note that the potentially faulty behaviour of the medium Med is described abstractly in the defining equation for process Med’ by means of an internal transition to an ‘error state’. When it has entered that state, the medium informs the sender process that it has lost a message, and therefore that the message must be retransmitted. The sender will receive this message when in state Wait, and will proceed to retransmit the message.
We expect the protocol to behave like a one-place buffer described thus:

\[
\text{ProtocolSpec} \overset{\text{def}}{=} \text{acc}.\text{del}.\text{ProtocolSpec}
\]

Note, however, that the necessity of possibly having to retransmit a message some arbitrary number of times before a successful delivery means that the process describing the protocol has a livelock. (Find it!) However, you should be able to prove that

\[
\text{Protocol} \approx \text{ProtocolSpec}
\]

by building a suitable weak bisimulation.

**Exercise 3.22** Build the aforementioned weak bisimulation. ♦

The following theorem is the counterpart of Theorem 3.1 for weak bisimilarity. It states that \(\approx\) is an equivalence relation, and that it is the largest weak bisimulation.

**Theorem 3.3** For all LTSs, the relation \(\approx\) is

1. an equivalence relation,
2. the largest weak bisimulation, and
3. satisfies the following property:
   
   \[
   s_1 \approx s_2 \text{ iff for each action } \alpha,
   \]
   
   - if \(s_1 \xrightarrow{\alpha} s'_1\), then there is a transition \(s_2 \xrightarrow{\alpha} s'_2\) such that \(s'_1 \approx s'_2\);
   - if \(s_2 \xrightarrow{\alpha} s'_2\), then there is a transition \(s_1 \xrightarrow{\alpha} s'_1\) such that \(s'_1 \approx s'_2\).

**Proof:** The proof follows the lines of that of Theorem 3.1, and is therefore omitted. \(\square\)

**Exercise 3.23** Fill in the details of the proof of the above theorem. ♦

**Exercise 3.24** Show that strong bisimilarity is included in observational equivalence; that is, prove that any two strongly bisimilar states are also weakly bisimilar. ♦

**Exercise 3.25** Consider the following labelled transition system.

\[
\begin{align*}
&\text{Diagram} 1: \quad s \overset{\tau}{\rightarrow} s_1 \overset{\tau}{\rightarrow} s_2, \\
&\text{Diagram} 2: \quad t \overset{\tau}{\rightarrow} t_1 \overset{\tau}{\rightarrow} t_2 \overset{\tau}{\rightarrow} t_3
\end{align*}
\]
3.4. WEAK BISIMILARITY

Show that $s \approx t$ by finding a weak bisimulation containing the pair $(s, t)$.

**Exercise 3.26** Show that, for all $P, Q$, the following equivalences, which are usually referred to as Milner’s $\tau$-laws, hold:

\[
\alpha \cdot \tau . P \approx \alpha . P \\
P + \tau . P \approx \tau . P \\
\alpha . (P + \tau . Q) \approx \alpha . (P + \tau . Q) + \alpha . Q.
\]

(3.9) (3.10) (3.11)

Hint: Build appropriate weak bisimulations.

**Exercise 3.27** Show that, for all $P, Q$, if $P \xrightarrow{\tau} Q$ and $Q \xrightarrow{\tau} P$, then $P \approx Q$.

**Exercise 3.28** We say that a CCS process is $\tau$-free iff none of the states that it can reach by performing sequences of transitions affords a $\tau$-labelled transition. For example, $a.0$ is $\tau$-free, but $a.(b.0 \mid b.0)$ is not.

Prove that no $\tau$-free CCS process is observationally equivalent to $a.0 + \tau.0$.

**Exercise 3.29** Prove that, for each CCS process $P$, the process $P \backslash (\text{Act} - \{\tau\})$ is observationally equivalent to $0$. Does this remain true if we consider processes modulo strong bisimilarity?

**Exercise 3.30 (Mandatory)** Show that observational equivalence is the largest symmetric relation $\mathcal{R}$ satisfying that whenever $s_1 \mathcal{R} s_2$ then for each action $\alpha$ (including $\tau$), if $s_1 \xrightarrow{\alpha} s'_1$, then there is a transition $s_2 \xrightarrow{\alpha} s'_2$ such that $s'_1 \mathcal{R} s'_2$.

This means that observational equivalence may be defined like strong bisimilarity, but over a labelled transition system whose transitions are $\xrightarrow{\alpha}$, with $\alpha$ ranging over the set of actions including $\tau$.

**Exercise 3.31** For each sequence $\sigma$ of observable actions in $\mathcal{L}$, and states $s, t$ in an LTS, define the relation $\xrightarrow{\sigma}$ thus:

- $s \xrightarrow{\sigma} t$ iff $s \xrightarrow{\tau} t$, and
- $s \xrightarrow{\alpha \sigma'} t$ iff $s \xrightarrow{\alpha} s' \xrightarrow{\sigma'} t$, for some $s'$.

A binary relation $\mathcal{R}$ over the set of states of an LTS is a weak string bisimulation iff whenever $s_1 \mathcal{R} s_2$ and $\sigma$ is a (possibly empty) sequence of observable actions in $\mathcal{L}$:

- if $s_1 \xrightarrow{\sigma} s'_1$, then there is a transition $s_2 \xrightarrow{\sigma} s'_2$ such that $s'_1 \mathcal{R} s'_2$;
- if $s_2 \xrightarrow{\sigma} s'_2$, then there is a transition $s_1 \xrightarrow{\sigma} s'_1$ such that $s'_1 \mathcal{R} s'_2$. 

Two states \( s \) and \( s' \) are weakly string bisimilar iff there is a weak string bisimulation that relates them.

Prove that weak string bisimilarity and weak bisimilarity coincide. That is, show that two states \( s \) and \( s' \) are weakly string bisimilar iff they are weakly bisimilar.

The notion of observational equivalence that we have just defined seems to meet many of our desiderata. There is, however, one important property that observational equivalence does not enjoy. In fact, unlike strong bisimilarity, observational equivalence is not a congruence. This means that, in general, we cannot substitute observationally equivalent processes one for the other in a process context without affecting the overall behaviour of the system.

To see this, observe that \( 0 \) is observationally equivalent to \( \tau \). 0. However, it is not hard to see that

\[ a.0 + 0 \not\approx a.0 + \tau.0 . \]

In fact, the transition \( a.0 + \tau.0 \overset{\tau}{\rightarrow} 0 \) from the process \( a.0 + \tau.0 \) can only be matched by \( a.0 + 0 \overset{\tau}{\Rightarrow} a.0 + 0 \), and the processes \( 0 \) and \( a.0 + 0 \) are not observationally equivalent. However, we still have that weak bisimilarity is a congruence with respect to the remaining CCS operators.

**Theorem 3.4** Let \( P, Q, R \) be CCS processes. Assume that \( P \approx Q \). Then

- \( \alpha.P \approx \alpha.Q \), for each action \( \alpha \),
- \( P \mid R \approx Q \mid R \) and \( R \mid P \approx R \mid Q \), for each process \( R \),
- \( P[f] \approx Q[f] \), for each relabelling \( f \), and
- \( P \setminus L \approx Q \setminus L \), for each set of labels \( L \).

**Proof:** The proof follows the lines of that of Theorem 3.2, and is left as an exercise for the reader.

**Exercise 3.32** Prove Theorem 3.4. In the proof of the second claim in the proposition, you may find the following fact useful:

if \( Q \overset{a}{\Rightarrow} Q' \) and \( R \overset{a}{\Rightarrow} R' \), then \( Q|R \overset{\tau}{\Rightarrow} Q'|R' \).

Show this fact by induction on the number of \( \tau \)-steps in the transition \( Q \overset{a}{\Rightarrow} Q' \).

**Exercise 3.33** Give syntactic restrictions on the syntax of CCS terms so that weak bisimilarity becomes a congruence also with respect to the choice operator.
In light of Theorem 3.4, observational equivalence is very close to being a congruence over CCS. The characterization and the study of the largest congruence relation included in observational equivalence is a very interesting chapter in process theory. It is, however, one that we will not touch upon in this book. (See, however, Exercise 3.36 for a glimpse of this theory.) The interested reader is referred to (Milner, 1989, Chapter 7) and (Glabbeek, 2005) for an in depth treatment of this interesting topic.

**Exercise 3.34 (Dijkstra’s Dining Philosophers Problem)** In this exercise, we invite you to use the Edinburgh Concurrency Workbench—a software tool for the analysis of reactive systems specified as CCS processes—to model and analyze the dining philosophers problem proposed by the late Edsger Dijkstra in his classic paper (Dijkstra, 1971).

The problem is usually described as follows. Five philosophers spend their time eating and thinking. Each philosopher usually keeps thinking, but at any point in time he might become hungry and decide that it is time to eat. The research institute where the philosophers work has a round dining table with a large bowl of spaghetti at the centre of the table. There are five plates on the table and five forks set between the plates. Each philosopher needs two forks to eat, which he picks up one at a time. The funding agency sponsoring the institute is only interested in the thinking behaviour of the philosophers, and would like the institute to perform like an ideal think factory—that is, like a system that produces thinking forever.

1. Assume, to begin with, that there are only two philosophers and two forks. Model the philosophers and the forks as CCS processes, assuming that the philosophers and the forks are numbered from 1 to 2, and that the philosophers pick the forks up in increasing order. (When he becomes hungry, the second philosopher begins by picking up the second fork, and then picks up the first.) Argue that the system has a deadlock by finding a state in the resulting labelled transition system that is reachable from the start state, and has no outgoing transitions.

We encourage you to find a possible deadlock in the system by yourselves, and without using the Workbench.

2. Argue that a model of the system with five philosophers and five forks also exhibits a deadlock.

3. Finally, assume that there are five philosophers and five forks, and that the philosophers pick the forks up in increasing order, apart from the fifth, who picks up the first fork before the fifth. Use the the Edinburgh Concurrency
Workbench to argue that the resulting system is observationally equivalent to the process ThinkFactory specified by

\[
\text{ThinkFactory} \overset{\text{def}}{=} \text{think}.\text{ThinkFactory}
\]

Here we are assuming that each philosopher performs action ‘think’ when he is thinking, and that the funding agency is not interested in knowing which specific philosopher is thinking!

Exercise 3.35 (For the theoretically minded) A binary relation \( \mathcal{R} \) over the set of states of an LTS is a branching bisimulation (van Glabbeek and Weijland, 1996) iff it is symmetric, and whenever \( s \mathcal{R} t \) and \( \alpha \) is an action (including \( \tau \)):

1. Show that branching bisimilarity is contained in weak bisimilarity.
2. Can you find two processes that are weakly bisimilar, but not branching bisimilar?
3. Which of the \( \tau \)-laws from Exercise 3.26 holds with respect to branching bisimilarity?

Is branching bisimilarity a congruence over the language CCS?

Exercise 3.36 Define the binary relation \( \approx^c \) over the set of states of an LTS as follows:

\[
s_1 \approx^c s_2 \text{ iff for each action } \alpha \text{ (including } \tau):\n\begin{align*}
- & \text{ if } s_1 \xrightarrow{\alpha} s_1', \text{ then there is a sequence of transitions } s_2 \xrightarrow{\tau} s_2'' \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_2' \xrightarrow{\alpha} s_2''', \\
- & \text{ such that } s_1' \approx s_2''', s_2' \approx s_2', s_1 \approx s_2'\text{.}
\end{align*}
\]
3.5. GAME CHARACTERIZATION OF BISIMILARITY

- if \( s_2 \xrightarrow{\alpha} s_2' \), then there is a sequence of transitions \( s_1 \xrightarrow{\tau} s_1'' \xrightarrow{\alpha} s_1' \) such that \( s_1' \approx s_2' \).

Prove the following claims.

1. The relation \( \approx_c \) is an equivalence relation.

2. The relation \( \approx_c \) is preserved by the operators of CCS—that is, show that if \( P \approx_c Q \), then
   - \( \alpha.P \approx_c \alpha.Q \), for each action \( \alpha \);
   - \( P + R \approx_c Q + R \) and \( R + P \approx_c R + Q \), for each process \( R \);
   - \( P \mid R \approx_c Q \mid R \) and \( R \mid P \approx_c R \mid Q \), for each process \( R \);
   - \( P[f] \approx_c Q[f] \), for each relabelling \( f \); and
   - \( P \setminus L \approx_c Q \setminus L \), for each set of labels \( L \).

3. Argue that \( \approx_c \) is included in weak bisimilarity.

4. Find an example of two weakly bisimilar processes that are not related with respect to \( \approx_c \).

Which of the \( \tau \)-laws from Exercise 3.26 holds with respect to \( \approx_c \)?

3.5 Game characterization of bisimilarity

We can naturally ask ourselves the following question:

What techniques do we have to show that two states are not bisimilar?

In order to prove that for two given states \( s \) and \( t \) it is the case that \( s \not\sim t \), we should by Definition 3.2 enumerate all binary relations over the set of states and for each of them show that if it contains the pair \( (s, t) \) then it is not a strong bisimulation. For the transition system from Example 3.1 on page 44 this translates into investigating \( 2^{25} \) different candidates and in general for a transition system with \( n \) states one would have to go through \( 2^{n^2} \) different binary relations. (Can you see why?)

In what follows, we will introduce a game characterization of strong bisimilarity, which will enable us to determine in a much more perspicuous way whether two states are strongly bisimilar or not.

The idea is that there are two players in the bisimulation game, called ‘attacker’ and ‘defender’. The attacker is trying to show that two given states are not bisimilar while the defender aims to show the opposite. The formal definition follows.
CHAPTER 3.5. GAME CHARACTERIZATION OF BISIMILARITY

Definition 3.5 [Strong Bisimulation Game] Let \((\text{Proc}, \text{Act}, \{a \rightarrow \} \mid a \in \text{Act})\) be a labelled transition system. A **strong bisimulation game** starting from the pair of states \((s_1, t_1) \in \text{Proc} \times \text{Proc}\) is a two-player game of an ‘attacker’ and a ‘defender’.

The game is played in **rounds**, and **configurations** of the game are pairs of states from \(\text{Proc} \times \text{Proc}\). In every round exactly one configuration is called **current**; initially the configuration \((s_1, t_1)\) is the current one.

In each round the players change the current configuration \((s, t)\) according to the following rules.

1. The attacker chooses either the left- or the right-hand side of the current configuration \((s, t)\) and an action \(\alpha\) from \(\text{Act}\).
   
   - If the attacker chose left then he has to perform a transition \(s \xrightarrow{\alpha} s'\) for some state \(s' \in \text{Proc}\).
   
   - If the attacker chose right then he has to perform a transition \(t \xrightarrow{\alpha} t'\) for some state \(t' \in \text{Proc}\).

2. In this step the defender must provide an answer to the attack made in the previous step.
   
   - If the attacker chose left then the defender plays on the right-hand side, and has to respond by making a transitions \(t \xrightarrow{\alpha} t'\) for some \(t' \in \text{Proc}\).
   
   - If the attacker chose right then the defender plays on the left-hand side and has to respond by making a transitions \(s \xrightarrow{\alpha} s'\) for some \(s' \in \text{Proc}\).

3. The configuration \((s', t')\) becomes the current configuration and the game continues for another round according to the rules described above.

A **play** of the game is a maximal sequence of configurations formed by the players according to the rules described above, and starting from the initial configuration \((s_1, t_1)\). (A sequence of configurations is maximal if it cannot be extended while following the rules of the game.) Note that a bisimulation game can have many different plays according to the choices made by the attacker and the defender. The attacker can choose a side, an action and a transition. The defender’s only choice is in selecting one of the available transitions that are labelled with the same action picked by the attacker.

We shall now define when a play is winning for the attacker and when for the defender.
A finite play is lost by the player who is stuck and cannot make a move from the current configuration \((s, t)\) according to the rules of the game. Note that the attacker loses a finite play only if both \(s \not\rightarrow\) and \(t \not\rightarrow\), i.e., there is no transition from both the left- and the right-hand side of the configuration. The defender loses a finite play if he has (on his side of the configuration) no available transition under the action selected by the attacker.

It can also be the case that none of the players is stuck in any configuration and the play is infinite. In this situation the defender is the winner of the play. Intuitively, this is a natural choice of outcome because if the play is infinite then the attacker has been unable to find a ‘difference’ in the behaviour of the two systems—which will turn out to be bisimilar.

A given play is always winning either for the attacker or the defender and it cannot be winning for both at the same time.

The following proposition relates strong bisimilarity with the corresponding game characterization (see, e.g., (Stirling, 1995; Thomas, 1993)).

**Proposition 3.3** States \(s_1\) and \(t_1\) of a labelled transition system are strongly bisimilar if and only if the defender has a universal winning strategy in the strong bisimulation game starting from the configuration \((s_1, t_1)\). The states \(s_1\) and \(t_1\) are not strongly bisimilar if and only if the attacker has a universal winning strategy.

By universal winning strategy we mean that the player can always win the game, regardless of how the other player is selecting his moves. In case the opponent has more than one choice for how to continue from the current configuration, all these possibilities have to be considered.

The notion of a universal winning strategy is best explained by means of an example.

**Example 3.5** Let us recall the transition system from Example 3.1.

We will show that the defender has a universal winning strategy from the configuration \((s, t)\) and hence, in light of Proposition 3.3, that \(s \sim t\). In order to do so, we have to consider all possible attacker’s moves from this configuration and define
defender’s response to each of them. The attacker can make three different moves from \((s, t)\).

1. Attacker selects right-hand side, action \(a\) and makes the move \(t \xrightarrow{a} t_1\).
2. Attacker selects left-hand side, action \(a\) and makes the move \(s \xrightarrow{a} s_2\).
3. Attacker selects left-hand side, action \(a\) and makes the move \(s \xrightarrow{a} s_1\).

- Defender’s answer to attack 1. is by playing \(s \xrightarrow{a} s_2\).
  (Even though there are more possibilities it is sufficient to provide only one suitable answer.)
  The current configuration becomes \((s_2, t_1)\).
- Defender’s answer to attack 2. is by playing \(t \xrightarrow{a} t_1\).
  The current configuration becomes again \((s_2, t_1)\).
- Defender’s answer to attack 3. is by playing \(t \xrightarrow{a} t_1\).
  The current configuration becomes \((s_1, t_1)\).

Now it remains to show that the defender has a universal winning strategy from the configurations \((s_2, t_1)\) and \((s_1, t_1)\).

From \((s_2, t_1)\) it is easy to see that any continuation of the game will always go through the same current configuration \((s_2, t_1)\) and hence the game will be necessarily infinite. According to the definition of a winning play, the defender is the winner in this case.

From \((s_1, t_1)\) the attacker has two possible moves. Either \(s_1 \xrightarrow{b} s_2\) or \(t_1 \xrightarrow{b} t_1\).
In the former case the defender answers by \(t_1 \xrightarrow{b} t_1\), and in the latter case by \(s_1 \xrightarrow{b} s_2\). The next configuration is in both cases \((s_2, t_1)\), and we already know that the defender has a winning strategy from this configuration.

Hence we have shown that the defender has a universal winning strategy from the configuration \((s, t)\) and, according to Proposition 3.3, this means that \(s \sim t\). ♦

The game characterization of bisimilarity introduced above is simple, yet powerful. It provides an intuitive understanding of this notion. It can be used both to show that two states are strongly bisimilar as well as that they are not. The technique is particularly useful for showing non-bisimilarity of two states. This is demonstrated by the following examples.

**Example 3.6** Let us consider the following transition system (we provide only its graphical representation).
We will show that $s \not\sim t$ by describing a universal winning strategy for the attacker in the bisimulation game starting from $(s, t)$. We will in fact show two different strategies (but of course finding one is sufficient for proving non-bisimilarity).

- In the first strategy, the attacker selects the left-hand side, action $a$ and the transition $s \xrightarrow{a} s_1$. Defender can answer by $t \xrightarrow{a} t_1$ or $t \xrightarrow{a} t_2$. This means that we will have to consider two different configurations in the next round, namely $(s_1, t_1)$ and $(s_1, t_2)$. From $(s_1, t_1)$ the attacker wins by playing the transition $s_1 \xrightarrow{c} s_3$ on the left-hand side, and the defender cannot answer as there is no $c$-transition from $t_1$. From $(s_1, t_2)$ the attacker wins by playing $s_1 \xrightarrow{b} s_2$ and the defender has again no answer from $t_2$. As we analyzed all different possibilities for the defender and in every one the attacker wins, we have found a universal winning strategy for the attacker. Hence $s$ and $t$ are not bisimilar.

- Now we provide another strategy, which is easier to describe and involves switching of sides. Starting from $(s, t)$ the attacker plays on the right-hand side according to the transition $t \xrightarrow{a} t_1$ and the defender can only answer by $s \xrightarrow{a} s_1$ on the left-hand side (no more configurations need to be examined as this is the only possibility for the defender). The current configuration hence becomes $(s_1, t_1)$. In the next round the attacker plays $s_1 \xrightarrow{c} s_3$ and wins the game as $t_1 \nrightarrow c$.

Example 3.7 Let us consider a slightly more complex transition system.
We will define attacker’s universal winning strategy from \((s, t)\) and hence show that \(s \not\sim t\).

In the first round the attacker plays on the left-hand side the move \(s \xrightarrow{a} s_1\) and the defender can only answer by \(t \xrightarrow{a} t_1\). The current configuration becomes \((s_1, t_1)\). In the second round the attacker plays on the right-hand side according to the transition \(t_1 \xrightarrow{b} t_1\) and the defender can only answer by \(s_1 \xrightarrow{b} s_3\). The current configuration becomes \((s_3, t_1)\). Now the attacker wins by playing again the transition \(t_1 \xrightarrow{b} t_1\) (or \(t_1 \xrightarrow{b} t_2\)) and the defender loses because \(s_3 \not\sim s_3\).

Exercise 3.37 Consider the following labelled transition system.

\[
\begin{array}{c}
\text{s} \\
a \\
\downarrow \\
s_1 \\
b \\
\downarrow \\
s_2 \\
\text{b} \\
\downarrow \\
\end{array}
\quad
\begin{array}{c}
\text{t} \\
a \\
\downarrow \\
t_1 \\
b \\
\downarrow \\
t_2 \\
\text{b} \\
\downarrow \\
\end{array}
\quad
\begin{array}{c}
\text{u} \\
a \\
\downarrow \\
u_1 \\
b \\
\downarrow \\
u_2 \\
\text{b} \\
\downarrow \\
\end{array}
\quad
\begin{array}{c}
\text{v} \\
a \\
\downarrow \\
v_1 \\
b \\
\downarrow \\
v_2 \\
\text{b} \\
\downarrow \\
\end{array}
\]

Decide whether \(s \sim t\), \(s \sim u\), and \(s \sim v\). Support your claims by giving a universal winning strategy either for the attacker (in the negative case) or the defender (in the positive case). In the positive case, you should also define a strong bisimulation relating the pair of processes in question.

Exercise 3.38 (For the theoretically minded) Prove Proposition 3.3 on page 75. Hint: Argue that, using the universal winning strategy for the defender, you can find a strong bisimulation, and conversely that, given a strong bisimulation, you can define a universal winning strategy for the defender.

Exercise 3.39 (For the theoretically minded) Recall from Exercise 3.17 that a binary relation \(R\) over the set of states of an LTS is a simulation iff whenever \(s_1 \xrightarrow{a} s_2\) and \(a\) is an action then

- if \(s_1 \xrightarrow{a} s'_1\), then there is a transition \(s_2 \xrightarrow{a} s'_2\) such that \(s'_1 \xrightarrow{R} s'_2\).

A binary relation \(R\) over the set of states of an LTS is a 2-nested simulation iff \(R\) is a simulation and moreover \(R^{-1} \subseteq R\).
Two states $s$ and $s'$ are in simulation preorder (respectively in 2-nested simulation preorder) iff there is a simulation (respectively a 2-nested simulation) that relates them.

Modify the rules of the strong bisimulation game in such a way that it characterizes the simulation preorder and the 2-nested simulation preorder.

**Exercise 3.40 (For the theoretically minded)** Can you change the rules of the strong bisimulation game in such a way that it characterizes the ready simulation preorder introduced in Exercise 3.18?

### 3.5.1 Weak bisimulation games

We shall now introduce a notion of weak bisimulation game that can be used to characterize weak bisimilarity, as introduced in Definition 3.4. Recall that the main idea is that weak bisimilarity abstracts away from the internal behaviour of systems, which is modelled by the silent action $\tau$, and that to prove that two states in an LTS are weakly bisimilar it suffices only to exhibit a weak bisimulation that relates them.

As was the case for strong bisimilarity, showing that two states are not weakly bisimilar is more difficult and, using directly the definition of weak bisimilarity, means that we have to enumerate all binary relations on states, and verify that none of them is a weak bisimulation and at the same time contains the pair of states that we test for equivalence.

Fortunately, the rules of the strong bisimulation game as defined in the previous section need only be slightly modified in order to achieve a characterization of weak bisimilarity in terms of weak bisimulation games.

**Definition 3.6** [Weak Bisimulation Game] A weak bisimulation game is defined in the same way as the strong bisimulation game in Definition 3.5, with the only exception that the defender can answer using the weak transition relation $\alpha \Rightarrow$ instead of only $\alpha \rightarrow$ as in the strong bisimulation game. The attacker is still allowed to use only the $\alpha \rightarrow$ moves.

The definitions of a play and winning strategy are exactly as before and we have a similar proposition as for the strong bisimulation game.

**Proposition 3.4** Two states $s_1$ and $t_1$ of a labelled transition system are weakly bisimilar if and only if the defender has a universal winning strategy in the weak bisimulation game starting from the configuration $(s_1, t_1)$. The states $s_1$ and $t_1$ are not weakly bisimilar if and only if the attacker has a universal winning strategy.
We remind the reader of the fact that, in the weak bisimulation game from the current configuration \((s, t)\), if the attacker chooses a move under the silent action \(\tau\) (let us say \(s \xrightarrow{\tau} s'\)) then the defender can (as one possibility) simply answer by doing ‘nothing’, i.e., by idling in the state \(t\) (as we always have \(t \xRightarrow{\tau} t\)). In that case, the current configuration becomes \((s', t)\).

Again, the notions of play and universal winning strategy in the weak bisimulation game are best explained by means of an example.

**Example 3.8** Consider the following transition system.

We will show that \(s \not\approx t\) by defining a universal winning strategy for the attacker in the weak bisimulation game from \((s, t)\).

In the first round, the attacker selects the left-hand side and action \(a\), and plays the move \(s \xrightarrow{a} s_1\). The defender has three possible moves to answer: (i) \(t \xrightarrow{a} t_2\) via \(t_1\), (ii) \(t \xrightarrow{a} t_2\) via \(t_1\) and \(t_3\), and (iii) \(t \xrightarrow{a} t_3\) via \(t_1\). In case (i) and (ii) the current configuration becomes \((s_1, t_2)\) and in case (iii) it becomes \((s_1, t_3)\).

From the configuration \((s_1, t_2)\) the attacker wins by playing \(s_1 \xrightarrow{b} s_3\), and the defender loses because \(t_2 \xrightarrow{b} \not= t_2\).

From the configuration \((s_1, t_3)\) the attacker plays the \(\tau\)-move from the right-hand side: \(t_3 \xrightarrow{\tau} t_2\). Defender’s only answer from \(s_1\) is \(s_1 \xrightarrow{\tau} s_1\) because no \(\tau\) actions are enabled from \(s_1\). The current configuration becomes \((s_1, t_2)\) and, as argued above, the attacker has a winning strategy from this pair.

This concludes the proof and shows that \(s \not\approx t\) because we found a universal winning strategy for the attacker.

**Exercise 3.41** In the weak bisimulation game the attacker is allowed to use \(\xrightarrow{a}\) moves for the attacks, and the defender can use \(\xrightarrow{a}\) moves in response. Argue that if we modify the rules of the game so that the attacker can also use moves of the form \(\xrightarrow{a}\) then this does not provide any additional power for the attacker. Conclude that both versions of the game provide the same answer about bisimilarity/nonbisimilarity of two processes.
3.6 Further results on equivalence checking

In the following few paragraphs we shall provide an overview of a number of interesting results achieved within concurrency theory in the area of equivalence checking. We shall also provide pointers to selected references in the literature that the interested readers may wish to consult for further independent study.

The first class of systems we consider is the one generated by CCS processes which have finitely many reachable states and finitely many transitions only. Such systems, usually called regular, can simply be viewed as labelled transition systems with a finite set of states and finitely many transitions. For a labelled transition system with \( n \) states and \( m \) transitions, strong bisimilarity between any two given states is decidable in deterministic polynomial time—more precisely in \( O(nm) \) time (Kanellakis and Smolka, 1990). This result by Kanellakis and Smolka was subsequently improved upon by Paige and Tarjan who devised an algorithm that runs in \( O(m \log n) \) time (Paige and Tarjan, 1987). This is in strong contrast with the complexity of deciding language equivalence, where the problem is known to be PSPACE-complete (Hunt, Rosenkrantz and Szymanski, 1976). By way of further comparison, we recall that deciding strong bisimilarity between finite labelled transition systems is P-complete (Balcázar, Gabarró and Santha, 1992)—this means that it is one of the ‘hardest problems’ in the class P of problems solvable in polynomial time. (P-complete problems are of interest because they appear to lack highly parallel solutions. See, for instance, the book (Greenlaw, Hoover and Ruzzo, 1995).)

We remind the reader that the aforementioned complexity results for finite labelled transition systems are valid if the size of the input problem is measured as the number of states plus the number of transitions in the input labelled transition system. If we assume that the size of the input is the length of the CCS equations that describe a finite transition system, then we face the so called state explosion problem because relatively short CCS definitions can generate exponentially large labelled transition systems. (For example, you should be able to convince yourselves that the labelled transition system associated with the CCS expression

\[
a_1.0 \mid a_2.0 \mid \cdots \mid a_n.0
\]

has \( 2^n \) states.) In this case the strong bisimilarity checking problem becomes EXPTIME-complete (Laroussinie and Schnoebelen, 2000)—this means that it is one of the ‘hardest problems’ in the class EXPTIME of problems solvable in exponential time using deterministic algorithms.

The problem of checking observational equivalence (weak bisimilarity) over finite labelled transition systems can be reduced to that of checking strong bisimilarity using a technique called saturation. Intuitively, saturation amounts to
1. first pre-computing the weak transition relation, and then

2. constructing a new pair of finite processes whose original transitions are replaced with the weak transitions.

The question whether two states are weakly bisimilar now amounts to checking strong bisimilarity over the saturated systems. Since the computation of the weak transition relation can be carried out in polynomial time, the problem of checking for weak bisimilarity can also be decided in polynomial time.

This means that both strong and weak bisimilarity can be decided on finite-state transition systems faster than many other equivalences. This story repeats itself also when we consider more general classes of transition systems.

Let us consider a class called BPP for *Basic Parallel Processes*, first studied by Christensen in his PhD thesis (Christensen, 1993). This is a class of infinite-state transition systems generated by a subclass of CCS expressions containing action prefixing, bounded nondeterminism and a pure parallel composition with neither restriction nor communication. In the case of BPP the difference between equivalence checking with respect to strong bisimilarity and other notions of equivalence is even more striking. It is known that language equivalence (Hirshfeld, 1994) as well as essentially any other notion of equivalence except for bisimilarity is undecidable (Hüttel, 1994). On the other hand, a surprising result by Christensen, Hirshfeld and Moller (Christensen, Hirshfeld and Moller, 1993) shows that strong bisimilarity is decidable in general, and Hirshfeld, Jerrum and Moller (Hirshfeld, Jerrum and Moller, 1996b) showed that it is decidable in polynomial time for its subclass containing normed processes only. (A BPP process is *normed* iff from any of its reachable states it is possible to reach a situation where all actions are disabled.) Recently, the general bisimilarity problem for BPP was shown to be PSPACE-complete (Jančar, 2003; Srba, 2002a).

Should we try to go even further up (with respect to expressive power), we can consider the class of *Petri nets* (Reisig, 1985), a very well studied model of concurrent computation that strictly includes that of BPP processes. In fact, BPP is a subclass of Petri nets where every transition has exactly one input place. (This is also called the communication-free fragment of Petri nets.) The problem of whether two marked Petri nets are bisimilar, as well as a number of other problems, is undecidable, as shown by Jančar in (Jančar, 1995).

Researchers have also considered a sequential analogue to the BPP class, called BPA for *Basic Process Algebra*, introduced by Bergstra and Klop (see (Bergstra and Klop, 1982)). Here, instead of the parallel operator we have a full sequential composition operator. (Action prefixing in CCS enables only a limited way to express sequential composition, whereas in BPA one is allowed to write processes like $E.F$ where both $E$ and $F$ can have a rather complicated behaviour.) This
class also corresponds to context-free grammars in Greibach normal form where only left-most derivations are allowed. Bar-Hillel, Perles, and Shamir (Bar-Hillel, Perles and Shamir, 1961) showed that language equivalence for languages generated by BPA is undecidable. In fact, most of the studied equivalences (apart from bisimilarity, again!) are undecidable for this class of processes (Huynh and Tian, 1995; Groote and Hüttel, 1994). On the other hand, Baeten, Bergstra, and Klop showed that strong bisimilarity is decidable for normed BPA processes (Baeten, Bergstra and Klop, 1993), and there is even a polynomial time algorithm for checking strong bisimilarity over this subclass of BPA processes by Hirshfeld, Jerrum and Moller (Hirshfeld, Jerrum and Moller, 1996a).

Christensen, Hüttel, and Stirling proved in (Christensen, Hüttel and Stirling, 1995) that strong bisimilarity remains decidable for arbitrary (unnormed) BPA processes, but the precise complexity of the problem has not been determined yet. The problem is known to be PSPACE-hard (Srba, 2002b), yet no worse than doubly-exponential (Burkart, Caucal and Steffen, 1995).

The positive decidability trend is preserved even for a superclass of BPA called PDA for pushdown automata. Even though BPA and PDA coincide with respect to language equivalence (they both generate the class of context-free languages), PDA is strictly more expressive when bisimilarity is considered as the notion of equivalence. Celebrated results by Sénizergues (Senizergues, 1998) and Stirling (Stirling, 2000) both show the decidability of bisimulation equivalence over the class of pushdown automata. On the other hand, the problem of checking for weak bisimilarity over PDA is already undecidable (Srba, 2002c).

There are still some open problems left in the theory, mainly concerning the decidability of weak bisimilarity. We refer the reader to an up-to-date overview of the state-of-the-art (Srba, 2004) and to a more thorough introduction to the area available, for instance, in (Burkart, Caucal, Moller and Steffen, 2001; Mayr, 2000).
Chapter 4

Theory of fixed points and bisimulation equivalence

The aim of this chapter is to collect under one roof all the mathematical notions from the theory of partially ordered sets and lattices that is needed to introduce Tarski’s classic fixed point theorem. You might think that this detour into some exotic looking mathematics is unwarranted in this textbook. However, we shall then put these possible doubts of yours to rest by using this fixed point theorem to give an alternative definition of strong bisimulation equivalence. This reformulation of the notion of strong bisimulation equivalence is not just mathematically pleasing, but it also yields an algorithm for computing the largest strong bisimulation over finite labelled transition systems—i.e., labelled transition systems with only finitely many states, actions and transitions. This is an illustrative example of how apparently very abstract mathematical notions turn out to have algorithmic content and, possibly unexpected, applications in Computer Science. As you will see in what follows, we shall also put Tarski’s fixed point theorem to good use in Chapter 6, where the theory developed in this chapter will allow us to understand the meaning of recursively defined properties of reactive systems.

4.1 Posets and complete lattices

We start our technical developments in this chapter by introducing the notion of partially ordered set (also known as poset) and some useful classes of such structures that will find application in what follows. As you will see, you are already familiar with many of the examples of posets that we shall mention in this chapter.

Definition 4.1 [Partially Ordered Sets] A partially ordered set (poset) is a pair
CHAPTER 4. THEORY OF FIXED POINTS

\((D, \sqsubseteq)\), where \(D\) is a set, and \(\sqsubseteq\) is a binary relation over \(D\) (i.e., a subset of \(D \times D\)) such that:

- \(\sqsubseteq\) is reflexive, i.e., \(d \sqsubseteq d\) for all \(d \in D\);
- \(\sqsubseteq\) is antisymmetric, i.e., \(d \sqsubseteq e\) and \(e \sqsubseteq d\) imply \(d = e\) for all \(d, e \in D\);
- \(\sqsubseteq\) is transitive, i.e., \(d \sqsubseteq e \sqsubseteq d'\) implies \(d \sqsubseteq d'\) for all \(d, d', e \in D\).

We moreover say that \((D, \sqsubseteq)\) is a totally ordered set if, for all \(d, e \in D\), either \(d \sqsubseteq e\) or \(e \sqsubseteq d\) holds.

\begin{itemize}
  \item \((\mathbb{N}, \leq)\), where \(\mathbb{N}\) denotes the set of natural numbers, and \(\leq\) stands for the standard ordering over \(\mathbb{N}\).
  \item \((\mathbb{R}, \leq)\), where \(\mathbb{R}\) denotes the set of real numbers, and \(\leq\) stands for the standard ordering over \(\mathbb{R}\).
  \item \((A^*, \leq)\), where \(A^*\) is the set of strings over alphabet \(A\), and \(\leq\) denotes the prefix ordering between strings, i.e., for all \(s, t \in A^*\), \(s \leq t\) iff there exists \(w \in A^*\) such that \(sw = t\). (Check that this is indeed a poset!)
  \item Let \((A, \leq)\) be a finite totally ordered set. Then \((A^*, \prec)\), the set of strings in \(A^*\) ordered lexicographically, is a poset. Recall that, for all \(s, t \in A^*\), the relation \(s \prec t\) holds with respect to the lexicographic order if one of the following conditions apply:
    \begin{enumerate}
      \item the length of \(s\) is smaller than that of \(t\);
      \item \(s\) and \(t\) have equal length, and either \(s = \varepsilon\) or there are strings \(u, v, z \in A^*\) and letters \(a, b \in A\) such that \(s = uav\), \(t = ubz\) and \(a \leq b\).
    \end{enumerate}
  \item Let \((D, \sqsubseteq)\) be a poset and \(S\) be a set. Then the collection of functions from \(S\) to \(D\) is a poset when equipped with the ordering relation defined thus:
    \[ f \sqsubseteq g \iff f(s) \sqsubseteq g(s), \text{ for each } s \in S. \]
\end{itemize}

We encourage you to think of other examples of posets you are familiar with.

\begin{itemize}
  \item Exercise 4.1 Convince yourselves that the structures mentioned in the above example are indeed posets. Which of the above posets is a totally ordered set?
\end{itemize}
4.1. POSETS AND COMPLETE LATTICES

As witnessed by the list of structures in Example 4.1 and by the many other examples that you have met in your discrete mathematics courses, posets are abundant in mathematics. Another example of a poset that will play an important role in the developments to follow is the structure $(2^S, \subseteq)$, where $S$ is a set, $2^S$ stands for the set of all subsets of $S$, and $\subseteq$ denotes set inclusion. For instance, the structure $(2^{\text{Proc}}, \subseteq)$ is a poset for each set of states $\text{Proc}$ in a labelled transition system.

Exercise 4.2 Is the poset $(2^S, \subseteq)$ totally ordered?

Definition 4.2 [Least Upper Bounds and Greatest Lower Bounds] Let $(D, \sqsubseteq)$ be a poset, and take $X \subseteq D$.

- We say that $d \in D$ is an upper bound for $X$ iff $x \sqsubseteq d$ for all $x \in X$. We say that $d$ is the least upper bound (lub) of $X$, notation $\bigcup X$, iff
  - $d$ is an upper bound for $X$ and, moreover,
  - $d \sqsubseteq d'$ for every $d' \in D$ which is an upper bound for $X$.

- We say that $d \in D$ is a lower bound for $X$ iff $d \sqsubseteq x$ for all $x \in X$. We say that $d$ is the greatest lower bound (glb) of $X$, notation $\bigcap X$, iff
  - $d$ is a lower bound for $X$ and, moreover,
  - $d' \sqsubseteq d$ for every $d' \in D$ which is a lower bound for $X$.

In the poset $(\mathbb{N}, \leq)$, all finite subsets of $\mathbb{N}$ have least upper bounds. Indeed, the least upper bound of such a set is its largest element. On the other hand, no infinite subset of $\mathbb{N}$ has an upper bound. All subsets of $\mathbb{N}$ have a least element, which is their greatest lower bound.

In $(2^S, \subseteq)$, every subset $X$ of $2^S$ has a lub and a glb given by $\bigcup X$ and $\bigcap X$, respectively. For example, consider the poset $(2^\mathbb{N}, \subseteq)$, consisting of the family of subsets of the set of natural numbers $\mathbb{N}$ ordered by inclusion. Take $X$ to be the collection of finite sets of even numbers. Then $\bigcup X$ is the set of even numbers and $\bigcap X$ is the empty set. (Can you see why?)

Exercise 4.3 (Strongly recommended) Let $(D, \sqsubseteq)$ be a poset, and take $X \subseteq D$. Prove that the lub and the glb of $X$ are unique, if they exist.

Exercise 4.4

1. Prove that the lub and the glb of a subset $X$ of $2^S$ are indeed $\bigcup X$ and $\bigcap X$, respectively.
2. Give examples of subsets of \( \{a, b\}^* \) that have upper bounds in the poset \((\{a, b\}^*, \leq)\). Find examples of subsets that do not have upper bounds in that poset. ♦

As you have seen already, a poset like \((2^S, \subseteq)\) has the pleasing property that each of its subsets has both a least upper bound and a greatest lower bound. Posets with this property will play a crucial role in what follows, and we now introduce them formally.

**Definition 4.3** [Complete Lattices] A poset \((D, \sqsubseteq)\) is a complete lattice iff \(\bigcup X\) and \(\bigcap X\) exist for every subset \(X\) of \(D\). ♦

Note that a complete lattice \((D, \sqsubseteq)\) has a least element \(\bot = \bigcap D\), often called *bottom*, and a top element \(\top = \bigcup D\). For example, the bottom element of the poset \((2^S, \subseteq)\) is the empty set, and the top element is \(S\). (Why?) By Exercise 4.3, the least and top elements of a complete lattice are unique.

**Exercise 4.5** Let \((D, \sqsubseteq)\) be a complete lattice. What are \(\bigcup \emptyset\) and \(\bigcap \emptyset\)? Hint: Each element of \(D\) is both a lower bound and an upper bound for \(\emptyset\). Why? ♦

**Example 4.2**

- The poset \((\mathbb{N}, \leq)\) is not a complete lattice because, as remarked previously, it does not have lub’s for its infinite subsets.

- The poset \((\mathbb{N} \cup \{\infty\}, \sqsubseteq)\), obtained by adding a largest element \(\infty\) to \((\mathbb{N}, \leq)\), is instead a complete lattice. This complete lattice can be pictured as follows:

\[
\begin{align*}
\infty & \\
\vdots & \\
\uparrow & \\
2 & \\
\uparrow & \\
1 & \\
\uparrow & \\
0 & \\
\end{align*}
\]

where \(\leq\) is the reflexive and transitive closure of the \(\uparrow\) relation.

- \((2^S, \subseteq)\) is a complete lattice.

Of course, you should convince yourselves of these claims! ♦
4.2 Tarski’s fixed point theorem

Now that we have some familiarity with posets and complete lattices, we are in a position to state and prove Tarski’s fixed point theorem—Theorem 4.1. As you will see in due course, this apparently very abstract result plays a key role in computer science because it is a general tool that allows us to make sense of recursively defined objects. If you are interested in the uses of the theorem rather than in the reason why it holds, you can safely skip the proof of Theorem 4.1 on first reading. None of the future applications of that result in this textbook depend on its proof, and you should feel free to use it as a ‘black-box’.

In the statement of Tarski’s fixed point theorem, and in the applications to follow, the collection of monotonic functions will play an important role. We now proceed to define this type of function for the sake of completeness.

**Definition 4.4** [Monotonic Functions and Fixed Points] Let \((D, \sqsubseteq)\) be a poset. A function \(f : D \to D\) is **monotonic** iff \(d \sqsubseteq d'\) implies that \(f(d) \sqsubseteq f(d')\), for all \(d, d' \in D\).

An element \(d \in D\) is called a **fixed point** of \(f\) iff \(d = f(d)\).

For example, the function \(f : 2^\mathbb{N} \to 2^\mathbb{N}\) defined, for each \(X \subseteq \mathbb{N}\), by

\[
f(X) = X \cup \{1, 2\}
\]

is monotonic. The set \(\{1, 2\}\) is a fixed point of \(f\) because

\[
f(\{1, 2\}) = \{1, 2\} \cup \{1, 2\} = \{1, 2\}.
\]

**Exercise 4.6** Can you give another example of a fixed point of \(f\)? Can you characterize all of the fixed points of that function? Argue for your answers.

**Exercise 4.7** Consider the function that is like \(f\) above, but maps the set \(\{2\}\) to \(\{1, 2, 3\}\). Is such a function monotonic?

As another example, consider the poset
The identify function is monotonic, but the function mapping \( \bot \) to 0 and acting like the identity function on all of the other elements is not. (Why?) Note that both of the posets mentioned above are in fact complete lattices.

Intuitively, if we view the partial order relation in a poset \((D, \sqsubseteq)\) as an ‘information order’—that is, if we view \(d \sqsubseteq d'\) as meaning that ‘\(d'\) has at least as much information as \(d\)’—, then monotonic functions have the property that providing more information in the input will offer at least as much information as we had before in the output. (Our somewhat imprecise, but hopefully suggestive, slogan during lectures on this topic is that a monotonic function is one with the property that ‘the more you get in, the more you get out!’)

The following important theorem is due to Tarski (Tarski, 1955), and was also independently proven for the special case of lattices of sets by Knaster (Knaster, 1928).

**Theorem 4.1** [Tarski’s Fixed Point Theorem] Let \((D, \sqsubseteq)\) be a complete lattice, and let \(f : D \rightarrow D\) be monotonic. Then \(f\) has a largest fixed point \(z_{\text{max}}\) and a least fixed point \(z_{\text{min}}\) given by

\[
\begin{align*}
    z_{\text{max}} &= \bigsqcup \{x \in D \mid x \sqsubseteq f(x)\} \quad \text{and} \\
    z_{\text{min}} &= \bigsqcap \{x \in D \mid f(x) \sqsubseteq x\}.
\end{align*}
\]

**Proof:** First we shall prove that \(z_{\text{max}}\) is the largest fixed point of \(f\). This involves proving the following two statements:

1. \(z_{\text{max}}\) is a fixed point of \(f\), i.e., \(z_{\text{max}} = f(z_{\text{max}})\), and
2. for every \(d \in D\) that is a fixed point of \(f\), it holds that \(d \sqsubseteq z_{\text{max}}\).

In what follows we prove each of these statements separately. In the rest of the proof we let

\[
A = \{x \in D \mid x \sqsubseteq f(x)\}.
\]

1. Since \(\sqsubseteq\) is antisymmetric, to prove that \(z_{\text{max}}\) is a fixed point of \(f\), it is sufficient to show that

\[
\begin{align*}
    z_{\text{max}} &\sqsubseteq f(z_{\text{max}}) \quad \text{and} \\
    f(z_{\text{max}}) &\sqsubseteq z_{\text{max}}. 
\end{align*}
\]

(4.1) (4.2)

First of all, we shall show that (4.1) holds. By definition, we have that

\[
z_{\text{max}} = \bigsqcup A.
\]
4.2. TARSKI’S FIXED POINT THEOREM

Thus, for every \( x \in A \), it holds that \( x \subseteq z_{\text{max}} \). As \( f \) is monotonic, \( x \subseteq z_{\text{max}} \) implies that \( f(x) \subseteq f(z_{\text{max}}) \). It follows that, for every \( x \in A \),

\[
  x \subseteq f(x) \subseteq f(z_{\text{max}})
\]

Thus \( f(z_{\text{max}}) \) is an upper bound for the set \( A \). By definition, \( z_{\text{max}} \) is the least upper bound of \( A \). Thus \( z_{\text{max}} \subseteq f(z_{\text{max}}) \), and we have shown (4.1).

To prove that (4.2) holds, note that, from (4.1) and the monotonicity of \( f \), we have that \( f(z_{\text{max}}) \subseteq f(f(z_{\text{max}})) \). This implies that \( f(z_{\text{max}}) \in A \). Therefore \( f(z_{\text{max}}) \subseteq z_{\text{max}} \), as \( z_{\text{max}} \) is an upper bound for \( A \).

From (4.1) and (4.2), we have that \( z_{\text{max}} \subseteq f(z_{\text{max}}) \subseteq z_{\text{max}} \). By antisymmetry, it follows that \( z_{\text{max}} = f(z_{\text{max}}) \), i.e., \( z_{\text{max}} \) is a fixed point of \( f \).

2. We now show that \( z_{\text{max}} \) is the largest fixed point of \( f \). Let \( d \) be any fixed point of \( f \). Then, in particular, we have that \( d \subseteq f(d) \). This implies that \( d \in A \) and therefore that \( d \subseteq \bigcup A = z_{\text{max}} \).

We have thus shown that \( z_{\text{max}} \) is the largest fixed point of \( f \).

To show that \( z_{\text{min}} \) is the least fixed point of \( f \), we proceed in a similar fashion by proving the following two statements:

1. \( z_{\text{min}} \) is a fixed point of \( f \), i.e., \( z_{\text{min}} = f(z_{\text{min}}) \), and
2. \( z_{\text{min}} \subseteq d \), for every \( d \in D \) that is a fixed point of \( f \).

To prove that \( z_{\text{min}} \) is a fixed point of \( f \), it is sufficient to show that:

\[
  f(z_{\text{min}}) \subseteq z_{\text{min}} \quad \text{and} \quad z_{\text{min}} \subseteq f(z_{\text{min}}) .
\]

Claim (4.3) can be shown following the proof for (4.1), and claim (4.4) can be shown following the proof for (4.2). The details are left as an exercise for the reader. Having shown that \( z_{\text{min}} \) is a fixed point of \( f \), it is a simple matter to prove that it is indeed the least fixed point of \( f \). (Do this as an exercise).

Consider, for example, a complete lattice of the form \((2^{S}, \subseteq)\), where \( S \) is a set, and a monotonic function \( f : S \rightarrow S \). If we instantiate the statement of the above theorem to this setting, the largest and least fixed points for \( f \) can be characterized thus:

\[
  z_{\text{max}} = \bigcup \{X \subseteq S \mid X \subseteq f(X)\} \quad \text{and} \quad z_{\text{min}} = \bigcap \{X \subseteq S \mid f(X) \subseteq X\} .
\]
For instance, the largest fixed point of the function \( f : 2^\mathbb{N} \to 2^\mathbb{N} \) defined by \( f(X) = X \cup \{1, 2\} \) is

\[
\bigcup \{X \subseteq \mathbb{N} \mid X \subseteq X \cup \{1, 2\}\} = \mathbb{N} .
\]

On the other hand, the least fixed point of \( f \) is

\[
\bigcap \{X \subseteq \mathbb{N} \mid X \cup \{1, 2\} \subseteq X\} = \{1, 2\} .
\]

This follows because \( X \cup \{1, 2\} \subseteq X \) means that \( X \) already contains \( 1 \) and \( 2 \), and the smallest set with this property is \( \{1, 2\} \).

The following important theorem gives a characterization of the largest and least fixed points for monotonic functions over finite complete lattices. We shall see in due course how this result gives an algorithm for computing the fixed points which will find application in equivalence checking and in the developments in Chapter 6.

**Definition 4.5** Let \( D \) be a set, \( d \in D \), and \( f : D \to D \). For each natural number \( n \), we define \( f^n(d) \) as follows:

\[
\begin{align*}
    f^0(d) &= d \quad \text{and} \\
    f^{n+1}(d) &= f(f^n(d)) .
\end{align*}
\]

**Theorem 4.2** Let \( (D, \sqsubseteq) \) be a finite complete lattice and let \( f : D \to D \) be monotonic. Then the least fixed point for \( f \) is obtained as

\[
    z_{\min} = f^m(\bot) ,
\]

for some natural number \( m \). Furthermore the largest fixed point for \( f \) is obtained as

\[
    z_{\max} = f^M(\top) ,
\]

for some natural number \( M \).

**Proof:** We only prove the first statement as the proof for the second one is similar. As \( f \) is monotonic we have the following non-decreasing sequence

\[
    \bot \subseteq f(\bot) \subseteq f^2(\bot) \subseteq \ldots \subseteq f^i(\bot) \subseteq f^{i+1}(\bot) \subseteq \ldots
\]
of elements of $D$. As $D$ is finite, the sequence must be eventually constant, i.e., there is an $m$ such that $f^k(\bot) = f^m(\bot)$ for all $k \geq m$. In particular,

$$f(f^m(\bot)) = f^{m+1}(\bot) = f^m(\bot),$$

which is the same as saying that $f^m(\bot)$ is a fixed point for $f$.

To prove that $f^m(\bot)$ is the least fixed point for $f$, assume that $d$ is another fixed point for $f$. Then we have that $\bot \sqsubseteq d$ and therefore, as $f$ is monotonic, that $\bot \sqsubseteq f(\bot) \sqsubseteq f(d) = d$. By repeating this reasoning $m-1$ more times we get that $f^m(\bot) \sqsubseteq d$. We can therefore conclude that $f^m(\bot)$ is the least fixed point for $f$.

The proof of the statement that characterizes largest fixed points is similar, and left as an exercise for the reader. □

**Exercise 4.8 (For the Theoretically Minded)** Fill in the details in the proof of the above theorem. ✷

**Example 4.3** Consider the function $f : 2^{\{0,1\}} \to 2^{\{0,1\}}$ defined by

$$f(X) = X \cup \{0\}.$$ 

This function is monotonic, and $2^{\{0,1\}}$ is a complete lattice, when ordered using set inclusion, with the empty set as least element and $\{0, 1\}$ as largest element. The above theorem gives an algorithm for computing the least and largest fixed point of $f$. To compute the least fixed point, we begin by applying $f$ to the empty set. The result is $\{0\}$. Since, we have added $0$ to the input of $f$, we have not found our least fixed point yet. Therefore we proceed by applying $f$ to $\{0\}$. We have that

$$f(\{0\}) = \{0\} \cup \{0\} = \{0\}.$$ 

It follows that, not surprisingly, $\{0\}$ is the least fixed point of the function $f$.

To compute the largest fixed point of $f$, we begin by applying $f$ to the top element in our lattice, namely the set $\{0, 1\}$. Observe that

$$f(\{0, 1\}) = \{0, 1\} \cup \{0\} = \{0, 1\}.$$ 

Therefore $\{0, 1\}$ is the largest fixed point of the function $f$. ✷

**Exercise 4.9** Consider the function $g : 2^{\{0,1,2\}} \to 2^{\{0,1,2\}}$ defined by

$$g(X) = (X \cap \{1\}) \cup \{2\}.$$ 

Use Theorem 4.2 to compute the least and largest fixed point of $g$. ✷
Exercise 4.10 (For the Theoretically Minded) This exercise is for those amongst you that enjoy the mathematics of partially ordered sets. It has no direct bearing on the theory of reactive systems covered in the rest of the textbook.

1. Let \((D, \sqsubseteq)\) be a poset. An \(\omega\)-chain in \((D, \sqsubseteq)\) is a sequence \(d_i (i \geq 0)\) of elements of \(D\) such that \(d_i \sqsubseteq d_{i+1}\) for each \(i \geq 0\).

We say that \((D, \sqsubseteq)\) is a complete partial order (cpo) if each \(\omega\)-chain \(d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \cdots\) in \((D, \sqsubseteq)\) has a least upper bound (written \(\bigsqcup_{i \geq 0} d_i\)). A function \(f : D \to D\) is continuous (see, for instance, (Nielson and Nielson, 1992, Page 103)) if

\[
f\left(\bigsqcup_{i \geq 0} d_i\right) = \bigsqcup_{i \geq 0} f(d_i),
\]

for each \(\omega\)-chain \(d_i (i \geq 0)\).

Prove that if \((D, \sqsubseteq)\) is a cpo and \(f : D \to D\) is continuous, then the poset

\[
\left\{x \in D \mid f(x) = x\right\}, \sqsubseteq
\]

which consists of the set of fixed points of \(f\), is itself a cpo.

2. Give an example of a complete lattice \((D, \sqsubseteq)\) and of a monotonic function \(f : D \to D\) such that there are \(x, y \in D\) that are fixed points of \(f\), but \(\bigsqcup\{x, y\}\) is not a fixed point. Hint: Consider the complete lattice \(D\) pictured below

\[
\bullet \\
\uparrow \\
\bullet
\]

and construct such an \(f : D \to D\).

3. Let \((D, \sqsubseteq)\) be a complete lattice, and let \(f : D \to D\) be monotonic. Consider a subset \(X\) of \(\{x \in D \mid x \sqsubseteq f(x)\}\).

(a) Prove that \(\bigsqcup X \in \{x \in D \mid x \sqsubseteq f(x)\}\).
4.2. TARSKI’S FIXED POINT THEOREM

(b) Give an example showing that, in general,
\[ \bigcap X \notin \{ x \in D \mid x \sqsubseteq f(x) \} . \]

Hint: Consider the lattice pictured above, but turned upside down.

4. Let \((D, \sqsubseteq)\) be a complete lattice, and let \(f : D \to D\) be monotonic. Consider a subset \(X\) of \(\{ x \in D \mid f(x) \subseteq x \}\).

(a) Prove that \(\bigcap X \in \{ x \in D \mid f(x) \subseteq x \}\).

(b) Give an example showing that, in general, \(\bigcup X \notin \{ x \in D \mid f(x) \subseteq x \}\). Hint: Use your solution to part 2 above.

5. Let \((D, \sqsubseteq)\) be a complete lattice.

(a) Let \(D \to_{mon} D\) be the set of monotonic functions from \(D\) to \(D\) and \(\preceq\) be the relation defined on \(D \to_{mon} D\) by
\[ f \preceq g \iff f(d) \subseteq g(d), \text{ for each } d \in D . \]

Show that \(\preceq\) is a partial order on \(D \to_{mon} D\).

(b) Let \(\bigvee\) and \(\bigwedge\) be defined on \(D \to_{mon} D\) as follows.
- If \(\mathcal{F} \subseteq D \to_{mon} D\) then, for each \(d \in D\),
\[ (\bigvee \mathcal{F})(d) = \bigcup \{ f(d) \mid f \in \mathcal{F} \} . \]
- If \(\mathcal{F} \subseteq D \to_{mon} D\) then, for each \(d \in D\),
\[ (\bigwedge \mathcal{F})(d) = \bigcap \{ f(d) \mid f \in \mathcal{F} \} . \]

Show that \((D \to_{mon} D, \preceq)\) is a complete lattice with \(\bigvee\) and \(\bigwedge\) as lub and glb.

We invite those amongst you who would like to learn more about the mathematics of partially ordered sets and lattices to consult the book (Davey and Priestley, 2002) and the collection of notes (Harju, 2006).
4.3 Bisimulation as a fixed point

Now that we have the theory underlying Tarski’s fixed point theorem in place, it is high time to put it into practice. We shall first use the theory we have just developed to provide the promised reformulation of bisimulation equivalence, and next we shall show by means of examples how this reformulation leads directly to an algorithm for computing bisimilarity over finite labelled transition systems. The algorithm for computing bisimilarity that stems from the theory of fixed points is not the most efficient one that has been devised; however, it is really pleasing to see how apparently very abstract notions from mathematics turn out to have unexpected applications in computer science.

Throughout this section, we let \((\text{Proc}, \text{Act}, \{ \alpha \mid a \in \text{Act} \})\) be a labelled transition system. We recall that a relation \(R \subseteq \text{Proc} \times \text{Proc}\) is a strong bisimulation—see Definition 3.2 on page 43—if the following holds:

1. \(p \xrightarrow{\alpha} p'\) implies \(q \xrightarrow{\alpha} q'\) for some \(q'\) such that \((p', q') \in R\);
2. \(q \xrightarrow{\alpha} q'\) implies \(p \xrightarrow{\alpha} p'\) for some \(p'\) such that \((p', q') \in R\).

Then strong bisimulation equivalence (or strong bisimilarity) is defined as

\[
\sim = \bigcup \{ R \in 2^{(\text{Proc} \times \text{Proc})} \mid R \text{ is a strong bisimulation} \}
\]

In what follows we shall describe the relation \(\sim\) as a fixed point to a suitable monotonic function. First we note that \((2^{(\text{Proc} \times \text{Proc})}, \subseteq)\) (i.e., the set of binary relations over \(\text{Proc}\) ordered by set inclusion) is a complete lattice with \(\bigcup\) and \(\bigcap\) as least upper bound and greatest lower bound. (Why? In fact, you should be able to realize readily that we have seen this kind of complete lattice in our previous developments!)

Consider now a binary relation \(R\) over \(\text{Proc}\)—that is, an element of the set \(2^{(\text{Proc} \times \text{Proc})}\). We define the set \(\mathcal{F}(R)\) as follows:

\[(p, q) \in \mathcal{F}(R), \text{ for all } p, q \in \text{Proc}, \text{ if and only if}
\]

1. \(p \xrightarrow{\alpha} p'\) implies \(q \xrightarrow{\alpha} q'\) for some \(q'\) such that \((p', q') \in R\);
2. \(q \xrightarrow{\alpha} q'\) implies \(p \xrightarrow{\alpha} p'\) for some \(p'\) such that \((p', q') \in R\).

In other words, \(\mathcal{F}(R)\) contains all the pairs of processes from which, in one round of the bisimulation game, the defender can make sure that the players reach a current pair of processes that is already contained in \(R\).
You should now convince yourselves that a relation $R$ is a bisimulation if and only if $R \subseteq F(R)$, and consequently that
\[
\sim = \bigcup \{ R \in 2^{(\mathsf{Proc} \times \mathsf{Proc})} \mid R \subseteq F(R) \}.
\]
Take a minute to look at the above equality, and compare it with the characterization of the largest fixed point of a monotonic function given by Tarski’s fixed point theorem (Theorem 4.1). That theorem tells us that the largest fixed point of a monotonic function $f$ is the least upper bound of the set of elements $x$ such that $x \sqsubseteq f(x)$—these are called the post-fixed points of the function. In our specific setting, the least upper bound of a subset of $2^{(\mathsf{Proc} \times \mathsf{Proc})}$ is given by $\bigcup$, and the post-fixed points of $F$ are precisely the binary relations $R$ over Proc such that $R \subseteq F(R)$. This means that the definition of $\sim$ matches the one for the largest fixed point for $F$ perfectly!

We note that if $R, S \in 2^{(\mathsf{Proc} \times \mathsf{Proc})}$ and $R \subseteq S$ then $F(R) \subseteq F(S)$—that is, the function $F$ is monotonic over $(2^{(\mathsf{Proc} \times \mathsf{Proc})}, \subseteq)$. (Check this!) Therefore, as all the conditions for Tarski’s theorem are satisfied, we can conclude that $\sim$ is indeed the largest fixed point of $F$. In particular, by Theorem 4.2, if Proc is finite then $\sim$ is equal to $F^M(\mathsf{Proc} \times \mathsf{Proc})$ for some integer $M \geq 0$. Note how this gives us an algorithm to calculate $\sim$ for a given finite labelled transition system.

To compute $\sim$, simply evaluate the non-increasing sequence
\[
F^0(\mathsf{Proc} \times \mathsf{Proc}) \supseteq F^1(\mathsf{Proc} \times \mathsf{Proc}) \supseteq F^2(\mathsf{Proc} \times \mathsf{Proc}) \supseteq \cdots
\]
until the sequence stabilizes. (Recall, that $F^0(\mathsf{Proc} \times \mathsf{Proc})$ is just the top element in the complete lattice, namely $\mathsf{Proc} \times \mathsf{Proc}$.)

**Example 4.4** Consider the labelled transition system described by the following defining equations in CCS:
\[
\begin{align*}
Q_1 &= b.Q_2 + a.Q_3 \\
Q_2 &= c.Q_4 \\
Q_3 &= c.Q_4 \\
Q_4 &= b.Q_2 + a.Q_3 + a.Q_1.
\end{align*}
\]

In this labelled transition system, we have that
\[
\mathsf{Proc} = \{ Q_i \mid 1 \leq i \leq 4 \}.
\]

Below, we use $I$ to denote the identity relation over Proc—that is,
\[
I = \{(Q_i, Q_i) \mid 1 \leq i \leq 4\}.
\]
We calculate the sequence $F^i(\text{Proc} \times \text{Proc})$ for $i \geq 1$ thus:

$F^1(\text{Proc} \times \text{Proc}) = \{(Q_1, Q_4), (Q_4, Q_1), (Q_2, Q_3), (Q_3, Q_2)\} \cup I$

$F^2(\text{Proc} \times \text{Proc}) = \{(Q_2, Q_3), (Q_3, Q_2)\} \cup I$ and finally

$F^3(\text{Proc} \times \text{Proc}) = F^2(\text{Proc} \times \text{Proc})$.

Therefore, the only distinct processes that are related by the largest strong bisimulation over this labelled transition system are $Q_2$ and $Q_3$, and indeed $Q_2 \sim Q_3$. ♦

**Exercise 4.11** Using the iterative algorithm described above, compute the largest strong bisimulation over the labelled transition system described by the following defining equations in CCS:

\[
\begin{align*}
P_1 &= a.P_2 \\
P_2 &= a.P_1 \\
P_3 &= a.P_2 + a.P_4 \\
P_4 &= a.P_3 + a.P_5 \\
P_5 &= 0.
\end{align*}
\]

You may find it useful to draw the labelled transition system associated with the above CCS definition first. ♦

**Exercise 4.12** Use the iterative algorithm described above to compute the largest bisimulation over the labelled transition system in Example 3.7. ♦

**Exercise 4.13** What is the worst case complexity of the algorithm outlined above when run on a labelled transition system consisting of $n$ states and $m$ transitions? Express your answer using $O$-notation, and compare it with the complexity of the algorithm due to Page and Tarjan mentioned in Section 3.6. ♦

**Exercise 4.14** Let $(\text{Proc}, \text{Act}, \{ \alpha \rightarrow | a \in \text{Act}\})$ be a labelled transition system. For each $i \geq 0$, define the relation $\sim_i$ as follows.

- $s_1 \sim_0 s_2$ holds always.
- $s_1 \sim_{i+1} s_2$ holds iff for each action $\alpha$:
  - if $s_1 \xrightarrow{\alpha} s'_1$, then there is a transition $s_2 \xrightarrow{\alpha} s'_2$ such that $s'_1 \sim_i s'_2$;
  - if $s_2 \xrightarrow{\alpha} s'_2$, then there is a transition $s_1 \xrightarrow{\alpha} s'_1$ such that $s'_1 \sim_i s'_2$. 
4.3. **BISIMULATION AS A FIXED POINT**

Prove that, for each $i \geq 0$:

1. the relation $\sim_i$ is an equivalence relation,
2. $\sim_{i+1}$ is included in $\sim_i$, and
3. $\sim_i = \mathcal{F}^i(\text{Proc} \times \text{Proc})$.

♦

**Exercise 4.15**

1. Give a characterization for observational equivalence as a fixed point for a monotonic function similar to the one we presented above for strong bisimilarity.

2. Use your characterization to compute observational equivalence over the labelled transition system in Example 3.8.

What is the worst case complexity of your algorithm? ♦
Chapter 5

Hennessy-Milner logic

In the previous chapters we have seen that implementation verification is a natural approach to establishing the correctness of (models of) reactive systems described, for instance, in the language CCS. This is because CCS, like all other process algebras, can be used to describe both actual systems and their specifications. However, when establishing the correctness of our system with respect to a specification using a notion of equivalence like observational equivalence, we are somehow forced to specify the overall behaviour of the system under consideration.

Suppose, for instance, that all we want to know about our system is whether it can perform an $a$-labelled transition ‘now’. Phrasing this correctness requirement in terms of observational equivalence seems at best unnatural, and maybe cannot be done at all! (See the paper (Boudol and Larsen, 1992) for an investigation of this issue.)

We can imagine a whole array of similar properties of the behaviour of a process we may be interested in specifying and checking. For instance, we may wish to know whether our computer scientist

- is not willing to drink tea now,
- is willing to drink both coffee and tea now,
- is willing to drink coffee, but not tea, now,
- never drinks alcoholic beverages, or
- always produces a publication after drinking coffee.

No doubt, you will be able to come up with many others examples of similar properties of the computer scientist that we may wish to verify.
All of the aforementioned properties, and many others, seem best checked by exploring the state space of the process under consideration, rather than by transforming them into equivalence checking questions. However, before even thinking of checking whether these properties hold of a process, either manually or automatically, we need to have a language for expressing them. This language must have a formal syntax and semantics, so that it can be understood by a computer, and algorithms to check whether a process affords a property may be devised. Moreover, the use of a language with a well defined and intuitively understandable semantics will also allow us to overcome the imprecision that often accompanies natural language descriptions. For instance, what do we really mean when we say that

our computer scientist is willing to drink both coffee and tea now?

Do we mean that, in its current state, the computer scientist can perform both a coffee-labelled transition and a tea-labelled one? Or do we mean that these transitions should be possible one after the other? And, may these transitions be preceded and/or followed by sequences of internal steps? Whether our computer scientist affords the specified property clearly depends on the answer to the questions above, and the use of a language with a formal semantics will help us understand precisely what is meant. Moreover, giving a formal syntax to our specification language will tell us what properties we can hope to express using it.

The approach to specification and verification of reactive systems that we shall begin exploring in this section is often referred to as ‘model checking’. In this approach we usually use different languages for describing actual systems and their specifications. For instance, we may use CCS expressions or the LTSs that they denote to describe actual systems, and some kind of logic to describe specifications. In this section, we shall present a property language that has been introduced in process theory by Hennessy and Milner in (Hennessy and Milner, 1985). This logic is often referred to as Hennessy-Milner logic (or HML for short), and, as we shall see in due course, has a very pleasing connection with the notion of bisimilarity.

**Definition 5.1** The set $\mathcal{M}$ of Hennessy-Milner formulae over a set of actions $\text{Act}$ is given by the following abstract syntax:

$$F, G ::= \top | \bot | F \land G | F \lor G | \langle a \rangle F | [a]F,$$

where $a \in \text{Act}$, and we use $\top$ and $\bot$ to denote ‘true’ and ‘false’, respectively. If $A = \{a_1, \ldots, a_n\} \subseteq \text{Act}$ ($n \geq 0$), we use the abbreviation $\langle A \rangle F$ for the formula $\langle a_1 \rangle F \lor \ldots \lor \langle a_n \rangle F$ and $[A]F$ for the formula $[a_1]F \land \ldots \land [a_n]F$. (If $A = \emptyset$, then $\langle A \rangle F = \bot$ and $[A]F = \top$.)

\end{document}
We are interested in using the above logic to describe properties of CCS processes, or, more generally, of states in an LTS over the set of actions $\text{Act}$. The meaning of a formula in the language $\mathcal{M}$ is given by characterizing the collection of processes that satisfy it. Intuitively, this can be described as follows.

- All processes satisfy $\mathcal{t}$.
- No process satisfies $\mathcal{f}$.
- A process satisfies $F \land G$ (respectively, $F \lor G$) iff it satisfies both $F$ and $G$ (respectively, either $F$ or $G$).
- A process satisfies $\langle a \rangle F$ for some $a \in \text{Act}$ iff it affords an $a$-labelled transition leading to a state satisfying $F$.
- A process satisfies $[a]F$ for some $a \in \text{Act}$ iff all of its $a$-labelled transitions lead to a state satisfying $F$.

So, intuitively, a formula of the form $\langle a \rangle F$ states that it is possible to perform action $a$ and thereby satisfy property $F$. Whereas a formula of the form $[a]F$ states that no matter how a process performs action $a$, the state it reaches in doing so will necessarily satisfy the property $F$.

Logics that involve the use of expressions like possibly and necessarily are usually called modal logics, and, in some form or another, have been studied by philosophers throughout history, notably by Aristotle and philosophers in the middle ages. So Hennessy-Milner logic is a modal logic—in fact, a so-called multi-modal logic, since the logic involves modal operators that are parameterized by actions. The semantics of formulae is defined with respect to a given labelled transition system

$$(\text{Proc}, \text{Act}, \{a \rightarrow | a \in \text{Act}\}) .$$

We shall use $[F]$ to denote the set of processes in $\text{Proc}$ that satisfy $F$. This we now proceed to define formally.

**Definition 5.2** [Denotational Semantics] We define $[F] \subseteq \text{Proc}$ for $F \in \mathcal{M}$ by

1. $[\mathcal{t}] = \text{Proc}$
2. $[\mathcal{f}] = \emptyset$
3. $[F \land G] = [F] \cap [G]$  
4. $[F \lor G] = [F] \cup [G]$
5. $[\langle a \rangle F] = \langle a \cdot \rangle [F]$  
6. $[[a]F] = [\cdot a \cdot] [F]$ ,

where we use the set operators $\langle a \cdot \rangle, [\cdot a \cdot] : 2^{\text{Proc}} \to 2^{\text{Proc}}$ defined by $\langle a \cdot \rangle S = \{ p \in \text{Proc} | p \xrightarrow{a} p' \text{ and } p' \in S, \text{ for some } p' \}$ and $[\cdot a \cdot] S = \{ p \in \text{Proc} | p \xrightarrow{a} p' \text{ implies } p' \in S, \text{ for each } p' \}$.
We write $p \models F$ iff $p \in \{F\}$.

Two formulae are equivalent if, and only if, they are satisfied by the same processes in every transition system.

**Example 5.1** In order to understand the definition of the set operators $\langle \cdot a \cdot \rangle$, $\{ \cdot a \cdot \}$ introduced above, it is instructive to look at an example. Consider the following labelled transition system.

![Labelled Transition System](image)

Then

$$\langle \cdot a \cdot \rangle \{ s_1, t_1 \} = \{ s, t \} .$$

This means that $\langle \cdot a \cdot \rangle \{ s_1, t_1 \}$ is the collection of states from which it is possible to perform an $a$-labelled transition ending up in either $s_1$ or $t_1$. On the other hand,

$$\{ \cdot a \cdot \} \{ s_1, t_1 \} = \{ s_1, s_2, t, t_1 \} .$$

The idea here is that $\{ \cdot a \cdot \} \{ s_1, t_1 \}$ consists of the set of all processes that become either $s_1$ or $t_1$ no matter how they perform an $a$-labelled transition. Clearly, $s$ does not have this property because it can perform the transition $s \xrightarrow{a} s_2$, whereas $t$ does because its only $a$-labelled transition ends up in $t_1$. But why are $s_1$, $s_2$ and $t_1$ in $\{ \cdot a \cdot \} \{ s_1, t_1 \}$? To see this, look at the formal definition of the set

$$\{ \cdot a \cdot \} \{ s_1, t_1 \} = \{ p \in \text{Proc} \mid p \xrightarrow{a} p' \text{ implies } p' \in \{ s_1, t_1 \}, \text{ for each } p' \} .$$

Since $s_1$, $s_2$ and $t_1$ do not afford $a$-labelled transitions it is vacuously true that all of their $a$-labelled transitions end up in either $s_1$ or $t_1$! This is the reason why those states are in the set $\{ \cdot a \cdot \} \{ s_1, t_1 \}$.

We shall come back to this important point repeatedly in what follows.

**Exercise 5.1** Consider the labelled transition system in the example above. What are $\langle \cdot b \cdot \rangle \{ s_1, t_1 \}$ and $\{ \cdot b \cdot \} \{ s_1, t_1 \}$?

Let us now re-examine the properties of our computer scientist that we mentioned earlier, and let us see whether we can express them using HML. First of all, note that, for the time being, we have defined the semantics of formulae in $\mathcal{M}$ in terms
of the one step transitions $\alpha$. This means, in particular, that we are not considering $\tau$ actions as unobservable. So, if we say that ‘a process $P$ can do action $a$ now’, then we really mean that the process can perform a transition of the form $P \xrightarrow{a} Q$ for some $Q$.

How can we express, for instance, that our computer scientist is willing to drink coffee now? Well, one way to say so using our logic is to say that the computer scientist has the possibility of doing a coffee-labelled transition. This suggests that we use a formula of the form $\langle \text{coffee} \rangle F$ for some formula $F$ that should be satisfied by the state reached by the computer scientist after having drunk her coffee. What should this $F$ be? Since we are not requiring anything of the subsequent behaviour of the computer scientist, it makes sense to set $F = \mathit{tt}$. So, it looks as if we can express our natural language requirement in terms of the formula $\langle \text{coffee} \rangle \mathit{tt}$. In fact, since our property language has a formal semantics, we can actually prove that our proposed formula is satisfied exactly by all the processes that have an outgoing coffee-labelled transition. This can be done as follows:

$$\begin{align*}
[\langle \text{coffee} \rangle \mathit{tt}] &= \langle \cdot \rangle \langle \text{coffee} \rangle \mathit{tt} \\
&= \langle \cdot \rangle \langle \text{coffee} \rangle \mathit{tt} \\
&= \langle \cdot \rangle \mathit{tt} \mathit{Proc} \\
&= \{ P \mid P \xrightarrow{\text{coffee}} P' \text{ for some } P' \in \mathit{Proc} \} \\
&= \{ P \mid P \xrightarrow{\text{coffee}} \}.
\end{align*}$$

So the formula we came up with does in fact say what we wanted.

Can we express using HML that the computer scientist cannot drink tea now? Consider the formula $[\text{tea}] \mathit{ff}$. Intuitively this formula says that all the states that a process can reach by doing a tea-labelled transition must satisfy the formula $\mathit{ff}$, i.e., false. Since no state has the property ‘false’, the only way that a process can satisfy the property $[\text{tea}] \mathit{ff}$ is that it has no tea-labelled transition. To prove formally that our proposed formula is satisfied exactly by all the processes that have no outgoing tea-labelled transition, we proceed as follows:

$$\begin{align*}
[\langle \text{tea} \rangle \mathit{ff}] &= [\cdot \langle \text{tea} \rangle \mathit{ff}] \\
&= [\cdot \langle \text{tea} \rangle \emptyset] \\
&= \{ P \mid P \xrightarrow{\text{tea}} P' \text{ implies } P' \in \emptyset, \text{ for each } P' \} \\
&= \{ P \mid P \xrightarrow{\text{tea}} \}.
\end{align*}$$

The last equality above follows from the fact that, for each process $P$,

$$P \xrightarrow{\text{tea}} \iff (P \xrightarrow{\text{tea}} P' \text{ implies } P' \in \emptyset, \text{ for each } P').$$
To see that this holds, observe first of all that if \( P \rightarrow \text{tea} \rightarrow Q \) for some \( Q \), then it is not true that \( P' \in \emptyset \) for all \( P' \) such that \( P \rightarrow \text{tea} \rightarrow P' \). In fact, \( Q \) is a counter-example to the latter statement. So the implication from right to left is true. To establish the implication from left to right, assume that \( P \rightarrow \text{tea} \rightarrow P' \). Then it is vacuously true that \( P' \in \emptyset \) for all \( P' \) such that \( P \rightarrow \text{tea} \rightarrow P' \)—indeed, since there is no such \( P' \), there is no counter-example to that statement!

To sum up, we can express that a process cannot perform action \( a \in \text{Act} \) with the formula \( [a]f \).

Suppose now that we want to say that the computer scientist must have a biscuit after drinking coffee. This means that it is possible for the computer scientist to have a biscuit in all the states that she can reach by drinking coffee. This can be expressed by means of the formula

\[ \text{coffee} \langle \text{biscuit} \rangle \# . \]

**Exercise 5.2 (Recommended)**

1. Use the semantics of the logic to check that the above formula expresses the desired property of the computer scientist.

2. Give formulae that express the following natural language requirements:
   - the process is willing to drink both coffee and tea now;
   - the process is willing to drink coffee, but not tea now;
   - the process can always drink tea immediately after having drunk two coffees in a row.

3. What do the formulae \( \langle a \rangle f \) and \( [a]t \) express?

**Exercise 5.3** Consider the following labelled transition system.
1. Decide whether the following statements hold:

- \( s \models (a)\# \),
- \( s \models (b)\# \),
- \( s \models [a]ff \),
- \( s \models [b]ff \),
- \( s \models [a](b)\# \),
- \( s \models (a)(b)\# \),
- \( s \models [a]([a]b)ff \),
- \( s \models (a)(a)\# \land (b)\# \),
- \( s \models [a]([a]\# \land [b]ff \land (b)\# \). and
- \( s \models (a)([a]\# \land [b]ff \land (b)\# \). and

2. Compute the following sets using the denotational semantics for Hennessy-Milner logic.

- \([[[a]b]ff] = ?\)
- \([[[a]([[a]\# \land (b)\# ]]] = ?\)
- \([[[a]b]ff] = ?\)
- \([[[a]([a]\# \land (b)\# ]]] = ?\)

**Exercise 5.4** Consider an everlasting clock whose behaviour is defined thus:

\[
\text{Clock} \overset{\text{def}}{=} \text{tick.Clock}.
\]

Prove that the process Clock satisfies the formula

\[
[tick](\langle \text{tick}\rangle\# \land \langle \text{tock}\rangle ff) .
\]
CHAPTER 5. HENNESSY-MILNER LOGIC

Show also that, for each \( n \geq 0 \), the process \( \text{Clock} \) satisfies the formula

\[
\langle \text{tick} \rangle \cdots \langle \text{tick} \rangle \ \#.
\]

\[\text{n-times}\]

**Exercise 5.5 (Mandatory)** Find a formula in \( \mathcal{M} \) that is satisfied by \( a.b.0 + a.c.0 \), but not by \( a.(b.0 + c.0) \).

Find a formula in \( \mathcal{M} \) that is satisfied by \( a.(b.c.0 + b.d.0) \), but not by \( a.b.c.0 + a.b.d.0 \).

It is sometimes useful to have an alternative characterization of the satisfaction relation \( \models \) presented in Definition 5.2. This can be obtained by defining the binary relation \( \models \) relating processes to formulae by structural induction on formulae thus:

- \( P \models \# \), for each \( P \),
- \( P \models \mathit{ff} \), for no \( P \),
- \( P \models F \land G \) iff \( P \models F \) and \( P \models G \),
- \( P \models F \lor G \) iff \( P \models F \) or \( P \models G \),
- \( P \models \langle a \rangle F \) iff \( P \xrightarrow{a} P' \) for some \( P' \) such that \( P' \models F \), and
- \( P \models [a] F \) iff \( P' \models F \), for each \( P' \) such that \( P \xrightarrow{a} P' \).

**Exercise 5.6** Show that the above definition of the satisfaction relation is equivalent to that given in Definition 5.2. Hint: Use induction on the structure of formulae.

**Exercise 5.7** Find one labelled transition system with initial state \( s \) that satisfies all of the following properties:

- \( \langle a \rangle \langle \langle b \rangle \langle c \rangle \# \land \langle c \rangle \# \rangle \),
- \( \langle a \rangle \langle b \rangle ([a] \mathit{ff} \land [b] \mathit{ff} \land [c] \mathit{ff}) \), and
- \( [a] \langle b \rangle ([c] \mathit{ff} \land \langle a \rangle \#) \).
Note that logical negation is not one of the constructs in the abstract syntax for \( \mathcal{M} \). However, the language \( \mathcal{M} \) is closed under negation, in the sense that, for each formula \( F \in \mathcal{M} \), there is a formula \( F^c \in \mathcal{M} \) that is equivalent to the negation of \( F \). This formula \( F^c \) is defined inductively on the structure of \( F \) as follows:

1. \( \#^c = \af \)
2. \( \af^c = \# \)
3. \( (F \land G)^c = F^c \lor G^c \)
4. \( (F \lor G)^c = F^c \land G^c \)
5. \( \langle a \rangle F^c = [a]F^c \)
6. \( ([a]F)^c = \langle a \rangle F^c \).

Note, for instance, that

\[
(\langle a \rangle \#)^c = [a] \af \quad \text{and} \quad ([a] \af)^c = \langle a \rangle \#.
\]

**Proposition 5.1** Let \((\text{Proc}, \text{Act}, \{ \overset{a}{\rightarrow} \mid a \in \text{Act} \})\) be a labelled transition system. Then, for every formula \( F \in \mathcal{M} \), it holds that \([F^c] = \text{Proc} \setminus [F] \).

**Proof:** The proposition can be proven by structural induction on \( F \). The details are left as an exercise to the reader. \(\square\)

**Exercise 5.8**

1. Prove Proposition 5.1.

2. Prove, furthermore, that \( (F^c)^c = F \) for every formula \( F \in \mathcal{M} \). Hint: Use structural induction on \( F \).

As a consequence of Proposition 5.1, we have that, for each process \( P \) and formula \( F \), exactly one of \( P \models F \) and \( P \models F^c \) holds. In fact, each process is either contained in \([F]\) or in \([F^c]\).

In Exercise 5.5 you were asked to come up with formulae that distinguished processes that we know are not strongly bisimilar. As a further example, consider the processes

\[
A \overset{\text{def}}{=} a.A + a.0 \quad \text{and} \quad B \overset{\text{def}}{=} a.a.B + a.0.
\]

These two processes are not strongly bisimilar. In fact, \( A \) affords the transition

\[
A \overset{a}{\rightarrow} A.
\]
This transition can only be matched by either
\[ B \xrightarrow{a} 0 \]
or
\[ B \xrightarrow{a} a.B. \]
However, neither 0 nor a.B is strongly bisimilar to A, because this process can perform an a-labelled transition and become 0 in doing so. On the other hand,
\[ a.B \xrightarrow{a} B \]
is the only transition that is possible from a.B, and B is not strongly bisimilar to 0.

Based on this analysis, it seems that a property distinguishing the processes A and B is \( \langle a \rangle \langle a \rangle \mathbb{F} \mathbb{F} \)—that is, the process can perform a sequence of two a-labelled transitions, and in so doing reach a state from which no a-labelled transition is possible. In fact, you should be able to establish that A satisfies this property, but B does not. (Do so!)

Again, faced with two non-bisimilar processes, we have been able to find a formula in the logic \( \mathcal{M} \) that distinguishes them, in the sense that one process satisfies it, but the other does not. Is this true in general? And what can we say about two processes that satisfy precisely the same formulae in \( \mathcal{M} \)? Are they guaranteed to be strongly bisimilar?

We shall now present a seminal theorem, due to Hennessy and Milner, that answers both of these questions in one fell swoop by establishing an elegant, and very fruitful, connection between the apparently unrelated notions of strong bisimilarity and the logic \( \mathcal{M} \). The theorem applies to a class of processes that we now proceed to define.

**Definition 5.3** [Image Finite Process] A process \( P \) is **image finite** iff the collection \( \{ P' \mid P \xrightarrow{a} P' \} \) is finite for each action \( a \).

An LTS is image finite if so is each of its states.

For example, the process \( A_{\text{rep}} \) (for ‘A replicated’) defined thus:
\[ A_{\text{rep}} \overset{\text{def}}{=} a.0 \mid A_{\text{rep}} \]
is not image finite. In fact, you should be able to prove by induction on \( n \) that, for each \( n \geq 0 \),
\[ A_{\text{rep}} \xrightarrow{a} a.0 \mid \cdots \mid a.0 | 0 \mid A_{\text{rep}}. \]
Another example of a process that is not image finite is
\[ A^{<\omega} \overset{\text{def}}{=} \sum_{i \geq 0} a^i, \]  
(5.1)

where \( a^0 = 0 \) and \( a^{i+1} = a.a^i \).

On the other hand all of the other processes that we have met so far in this text are image finite.

**Theorem 5.1** [Hennessy and Milner (Hennessy and Milner, 1985)] Let
\[ (\text{Proc}, \text{Act}, \{ \overset{a}{\to} | a \in \text{Act} \}) \]
be an image finite LTS. Assume that \( P, Q \) are states in \( \text{Proc} \). Then \( P \sim Q \) iff \( P \) and \( Q \) satisfy exactly the same formulae in Hennessy-Milner logic.

**Proof:** We prove the two implications separately.

- Assume that \( P \sim Q \) and \( P \models F \) for some formula \( F \in M \). Using structural induction on \( F \), we prove that \( Q \models F \). By symmetry, this is enough to establish that \( P \) and \( Q \) satisfy the same formulae in \( M \).

The proof proceeds by a case analysis on the form of \( F \). We only present the details for the case \( F = [a]G \) for some action \( a \) and formula \( G \). Our inductive hypothesis is that, for all processes \( R \) and \( S \), if \( R \sim S \) and \( R \models G \), then \( S \models G \). Using this hypothesis, we shall prove that \( Q \models [a]G \). To this end, assume that \( Q \overset{\alpha}{\to} Q' \) for some \( Q' \). We wish to show that \( Q' \models G \). Now, since \( P \sim Q \) and \( Q \overset{\alpha}{\to} Q' \), there is a process \( P' \) such that \( P \overset{\alpha}{\to} P' \) and \( P' \sim Q' \). (Why?) By our assumption that \( P \models [a]G \), we have that \( P' \models G \). The inductive hypothesis yields that \( Q' \models G \). Therefore each \( Q' \) such that \( Q \overset{\alpha}{\to} Q' \) satisfies \( G \), and we may conclude that \( Q \models [a]G \), which was to be shown.

- Assume that \( P \) and \( Q \) satisfy the same formulae in \( M \). We shall prove that \( P \) and \( Q \) are strongly bisimilar. To this end, note that it is sufficient to show that the relation
\[ \mathcal{R} = \{(R, S) \mid R, S \in \text{Proc} \text{ satisfy the same formulae in } M\} \]
is a strong bisimulation.

Assume that \( R \mathcal{R} S \) and \( R \overset{\alpha}{\to} R' \). We shall now argue that there is a process \( S' \) such that \( S \overset{\alpha}{\to} S' \) and \( R' \mathcal{R} S' \). Since \( \mathcal{R} \) is symmetric, this suffices to establish that \( \mathcal{R} \) is a strong bisimulation.
Assume, towards a contradiction, that there is no $S'$ such that $S \xrightarrow{a} S'$ and $S'$ satisfies the same properties as $R'$. Since $S$ is image finite, the set of processes $S$ can reach by performing an $a$-labelled transition is finite, say $\{S_1, \ldots, S_n\}$ with $n \geq 0$. By our assumption, none of the processes in the above set satisfies the same formulae as $R'$. So, for each $i \in \{1, \ldots, n\}$, there is a formula $F_i$ such that $R' \models F_i$ and $S_i \not\models F_i$.

(Why? Could it not be that $R' \not\models F_i$ and $S_i \models F_i$, for some $i \in \{1, \ldots, n\}$?)

We are now in a position to construct a formula that is satisfied by $R$, but not by $S$—contradicting our assumption that $R$ and $S$ satisfy the same formulae. In fact, the formula

$$\langle a \rangle (F_1 \land F_2 \land \cdots \land F_n)$$

is satisfied by $R$, but not by $S$. The easy verification is left to the reader.

The proof of the theorem is now complete. $\square$

**Exercise 5.9 (Mandatory)** Fill in the details that we have omitted in the above proof. What is the formula that we have constructed to distinguish $R$ and $S$ in the proof of the implication from right to left if $n = 0$?

**Remark 5.1** In fact, the implication from left to right in the above theorem holds for arbitrary processes, not just image finite ones.

The above theorem has many applications in the theory of processes, and in verification technology. For example, a consequence of its statement is that if two image finite processes are not strongly bisimilar, then there is a formula in $\mathcal{M}$ that tells us one reason why they are not. Moreover, as the proof of the above theorem suggests, we can always construct this distinguishing formula.

Note, moreover, that the above characterization theorem for strong bisimilarity is very general. For instance, in light of your answer to Exercise 3.30, it also applies to observational equivalence, provided that we interpret HML over the labelled transition system whose set of actions consists of all of the observable actions and of the label $\tau$, and whose transitions are precisely the ‘weak transitions’ whose labels are either observable actions or $\tau$.

**Exercise 5.10** Consider the following labelled transition system.
Argue that \( s \not\sim t \), \( s \not\sim v \) and \( t \not\sim v \). Next, find a distinguishing formula of Hennessy-Milner logic for the pairs

- \( s \) and \( t \),
- \( s \) and \( v \), and
- \( t \) and \( v \).

Verify your claims in the Edinburgh Concurrency Workbench (use the \texttt{strongeq} and \texttt{checkprop} commands) and check whether you found the shortest distinguishing formula (use the \texttt{dfstrong} command).

\textbf{Exercise 5.11} For each of the following CCS expressions decide whether they are strongly bisimilar and, if they are not, find a distinguishing formula in Hennessy-Milner logic.

- \( b.a.0 + b.0 \) and \( b.(a.0 + b.0) \),
- \( a.(b.c.0 + b.d.0) \) and \( a.b.c.0 + a.b.d.0 \),
- \( a.0 \mid b.0 \) and \( a.b.0 + b.a.0 \), and
- \( (a.0 \mid b.0) + c.a.0 \) and \( a.0 \mid (b.0 + c.0) \).

Verify your claims in the Edinburgh Concurrency Workbench (use the \texttt{strongeq} and \texttt{checkprop} commands) and check whether you found the shortest distinguishing formula (use the \texttt{dfstrong} command).

\textbf{Exercise 5.12 (For the Theoretically Minded)} Let \( (\text{Proc}, \text{Act}, \{ \sim_a \mid a \in \text{Act} \}) \) be image-finite. Show that

\[
\sim = \bigcap_{i \geq 0} \sim_i ,
\]
where $\sim_i (i \geq 0)$ is the sequence of equivalence relations defined in Exercise 4.14.

Exercise 5.13 (For the Theoretically Minded) Consider the process $A^\omega$ defined as follows:

$$A^\omega \overset{\text{def}}{=} a.A^\omega.$$ 

Show that the processes $A^{<\omega}$ and $A^\omega + A^{<\omega}$, where $A^{<\omega}$ was defined in equation (5.1) on page 111,

1. are not strongly bisimilar, but

2. satisfy the same properties in $\mathcal{M}$.

Conclude that Theorem 5.1 does not hold for processes that are not image finite.

Hint: To prove that the two processes satisfy the same formulae in $\mathcal{M}$, use structural induction on formulae. You will find it useful to first establish the following statement:

$$A^\omega \text{ satisfies a formula } F \in \mathcal{M} \text{ iff so does } a^i, \text{ where } i \text{ is the modal depth of } F.$$ 

The modal depth of a formula is the maximum nesting of the modal operators in it.
Chapter 6

Hennessy-Milner logic with recursive definitions

An HML formula can only describe a finite part of the overall behaviour of a process. In fact, as each modal operator allows us to explore the effect of taking one step in the behaviour of a process, using a single HML formula we can only describe properties of a fixed finite fragment of the computations of a process. As those of you who solved Exercise 5.13 already discovered, how much of the behaviour of a process we can explore using a single formula is entirely determined by its so-called modal depth—i.e., by the maximum nesting of modal operators in it. For example, the formula $\left[ a \right] \langle a \rangle ff \vee \langle b \rangle tt$ has modal depth 2, and checking whether a process satisfies it or not involves only an analysis of its sequences of transitions whose length is at most 2. (We will return to this issue in Section 6.6, where a formal definition of the modal depth of a formula will be given.)

However, we often wish to describe properties that describe states of affairs that may or must occur in arbitrarily long computations of a process. If we want to express properties as, for example, that a process is always able to perform a given action, we have to extend the logic. As the following example indicates, one way of doing so is to allow for infinite conjunctions and disjunctions in our property language.

**Example 6.1** Consider the processes $p$ and $q$ in Figure 6.1. It is not hard to come up with an HML formula that $p$ satisfies and $q$ does not. In fact, after performing an $a$-action, $p$ will always be able to perform another one, whereas $q$ may fail to do so. This can be captured formally in HML as follows:

$$p \models [a] \langle a \rangle tt \quad \text{but}$$

$$q \not\models [a] \langle a \rangle tt.$$
Since a difference in the behaviour of the two processes can already be found by examining their behaviour after two transitions, a formula that distinguishes them is ‘small’.

Assume, however, that we modify the labelled transition system for $q$ by adding a sequence of transitions to $r$ thus:

$$ r = r_0 \xrightarrow{a} r_1 \xrightarrow{a} r_2 \xrightarrow{a} \ldots \xrightarrow{a} r_{n-1} \xrightarrow{a} r_n \ (n \geq 0). $$

No matter how we choose a non-negative integer $n$, there is an HML formula that distinguishes the processes $p$ and $q$. In fact, we have that

$$ p \models [\langle a \rangle^+ \langle a \rangle^\#]^{n+1} \quad \text{but} \quad q \not\models [\langle a \rangle^+ \langle a \rangle^\#]^{n+1}, $$

where $[\langle a \rangle]^n$ stands for a sequence of modal operators $[\langle a \rangle]$ of length $n+1$. However, no formula in HML would work for all values of $n$. (Prove this claim!) This is unsatisfactory as there appears to be a general reason why the behaviours of $p$ and $q$ are different. Indeed, the process $p$ in Figure 6.1 can always (i.e., at any point in each of its computations) perform an $a$-action—that is, $\langle a \rangle^\#$ is always true. Let us call this *invariance* property $Inv(\langle a \rangle^\#)$. We could describe it in an extension of HML as an infinite conjunction thus:

$$ Inv(\langle a \rangle^\#) = \langle a \rangle^\# \wedge \langle a \rangle^\# \wedge \langle a \rangle^\# \wedge \ldots = \bigwedge_{i=0}^{\infty} [\langle a \rangle^i \langle a \rangle^\#]. $$

This formula can be read as follows:

In order for a process to be always able to perform an $a$-action, this action should be possible now (as expressed by the conjunct $\langle a \rangle^\#$), and, for each positive integer $i$, it should be possible in each state...
that the process can reach by performing a sequence of \(i\) actions (as expressed by the conjunct \([a]^i(a)\#\) because \(a\) is the only action in our example labelled transition system).

On the other hand, the process \(q\) has the option of terminating at any time by performing the \(a\)-labelled transition leading to process \(r\), or equivalently it is possible from \(q\) to satisfy \([a]ff\). Let us call this property \(Pos([a]ff)\). We can express it in an extension of HML as the following infinite disjunction:

\[
Pos([a]ff) = [a]ff \lor \langle a \rangle [a]ff \lor \langle a \rangle [a]ff \lor \cdots = \bigvee_{i=0}^{\infty} \langle a \rangle^i [a]ff,
\]

where \(\langle a \rangle^i\) stands for a sequence of modal operators \(\langle a \rangle\) of length \(i\). This formula can be read as follows:

In order for a process to have the possibility of refusing an \(a\)-action at some point, this action should either be refused now (as expressed by the disjunct \([a]ff\)), or, for some positive integer \(i\), it should be possible to reach a state in which an \(a\) can be refused by performing a sequence of \(i\) actions (as expressed by the disjunct \(\langle a \rangle^i [a]ff\) because \(a\) is the only action in our example labelled transition system).

Even if it is theoretically possible to extend HML with infinite conjunctions and disjunctions, infinite formulae are not particularly easy to handle (for instance they are infinitely long, and we would have a hard time using them as inputs for an algorithm). What do we do instead? The answer is in fact both simple and natural for a computer scientist; let us introduce recursion into our logic. Assuming for the moment that \(a\) is the only action, we can then express \(Inv(\langle a \rangle\#)\) by means of the following recursive equation:

\[
X \equiv \langle a \rangle\# \land [a]X,
\]

where we write \(F \equiv G\) if and only if the formulae \(F\) and \(G\) are satisfied by exactly the same processes—i.e., if \([F] = [G]\). The above recursive equation captures the intuition that a process that can invariantly perform an \(a\)-labelled transition—that is, one that can perform an \(a\)-labelled transition in all of its reachable states—can certainly perform one now, and, moreover, each state that it reaches via one such transition can invariantly perform an \(a\)-labelled transition. This looks deceptively easy and natural. However, the mere fact of writing down an equation like (6.1)
does not mean that this equation makes sense! Indeed, equations may be seen as implicitly defining the set of their solutions, and we are all familiar with equations that have no solutions at all. For instance, the equation

\[ x = x + 1 \] (6.2)

has no solution over the set of natural numbers, and there is no \( X \subseteq \mathbb{N} \) such that

\[ X = \mathbb{N} \setminus X \] (6.3)

On the other hand, there are uncountably many \( X \subseteq \mathbb{N} \) such that

\[ X = \{2\} \cup X \] (6.4)

namely all of the sets that contain the number 2. There are also equations that have a finite number of solutions, but not a unique one. As an example, consider the equation

\[ X = \{10\} \cup \{n - 1 \mid n \in X, \ n \neq 0\} \] (6.5)

The only finite set that is the solution for this equation is the set \( \{0, 1, \ldots, 10\} \), and the only infinite solution is \( \mathbb{N} \) itself.

**Exercise 6.1** Check the claims that we have just made.

**Exercise 6.2** Reconsider equations (6.2)–(6.5).

1. Why doesn’t Tarski’s fixed point theorem apply to yield a solution to the first two of these equations?

2. Consider the structure introduced in the second bullet of Example 4.2. For each \( d \in \mathbb{N} \cup \{\infty\} \), define

\[ \infty + d = d + \infty = \infty \]

Does equation (6.2) have a solution in the resulting structure? How many solutions does that equation have?

3. Use Tarski’s fixed point theorem to find the largest and least solutions of (6.5).

Since an equation like (6.1) is meant to describe a formula, it is therefore natural to ask ourselves the following questions:
Does (6.1) have a solution? And what precisely do we mean by that?

If (6.1) has more than one solution, which one do we choose?

How can we compute whether a process satisfies the formula described by (6.1)?

Precise answers to these questions will be given in the remainder of this chapter. However, to motivate our subsequent technical developments, it is appropriate here to discuss briefly the first two questions above.

Recall that the meaning of a formula (with respect to a labelled transition system) is the set of processes that satisfy it. Therefore, it is natural to expect that a set $S$ of processes that satisfy the formula described by equation (6.1) should be such that:

$$S = \langle a \rangle \text{Proc} \cap [a] S.$$

It is clear that $S = \emptyset$ is a solution to the equation (as no process can satisfy both $\langle a \rangle \text{tt}$ and $[a] \text{ff}$). But the process $p$ on Figure 6.1 can perform an $a$-transition invariantly and $p \not\in \emptyset$, so this cannot be the solution we are looking for. Actually it turns out that it is the largest solution we need here, namely where $S = \{p\}$. The set $S = \emptyset$ is the least solution.

In other cases it is the least solution we are interested in. For instance, we can express $\text{Pos}([a] \text{ff})$ by the following equation:

$$Y \equiv [a] \text{ff} \lor \langle a \rangle Y.$$

Here the largest solution is $Y = \{p, q, r\}$ but, as the process $p$ on Figure 6.1 cannot terminate at all, this is clearly not the solution we are interested in. The least solution over the labelled transition system on Figure 6.1 is $Y = \{q, r\}$ and is exactly the set of processes in that labelled transition system that intuitively satisfy $\text{Pos}([a] \text{ff})$.

When we write down a recursively defined property, we can indicate whether we desire the least or the largest solution by adding this information to the equality sign. For $\text{Inv}(\langle a \rangle \text{tt})$ we want the largest solution, and in this case we write

$$X_{\text{max}} = \langle a \rangle \text{tt} \land [a] X.$$

For $\text{Pos}([a] \text{ff})$ we will write

$$Y_{\text{min}} = [a] \text{ff} \lor \langle a \rangle Y.$$

More generally we can express that the formula $F$ holds for each reachable state in a labelled transition system having set of actions $\text{Act}$ (written $\text{Inv}(F)$), and read
‘invariantly $F’$) by means of the equation

$$X^{\text{max}} = F \land [\text{Act}]X$$

and that $F$ possibly holds at some point (written $Pos(F)$) by

$$Y^{\text{min}} = F \lor \langle \text{Act} \rangle Y.$$ 

Intuitively, we use largest solutions for those properties that hold of a process unless it has a finite computation that disproves the property. For instance, process $q$ does not have property $Inv(\langle a \rangle \#)$ because it can reach a state in which no $a$-labelled transition is possible. Conversely, we use least solutions for those properties that hold of a process if it has a finite computation sequence which ‘witnesses’ the property. For instance, a process has property $Pos(\langle a \rangle \#)$ if it has a computation leading to a state that can perform an $a$-labelled transition. This computation is a witness for the fact that the process can perform an $a$-labelled transition at some point in its behaviour.

We shall appeal to the intuition given above in the following section, where we present examples of recursively defined properties.

**Exercise 6.3** Give a formula, built using HML and the temporal operators $Pos$ and/or $Inv$, that expresses a property satisfied by exactly one of the processes in Exercise 5.13.

### 6.1 Examples of recursive properties

Adding recursive definitions to Hennessy-Milner logic gives us a very powerful language for specifying properties of processes. In particular this extension allows us to express different kinds of ‘safety’ and ‘liveness’ properties. Before developing the theory of HML with recursion, we give some more examples of its uses.

Consider the formula $Safe(F)$ that is satisfied by a process $p$ whenever it has a complete transition sequence

$$p = p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} p_2 \cdots,$$

where each of the processes $p_i$ satisfies $F$. (A transition sequence is complete if it is infinite or its last state affords no transition.) This *invariance of $F$ under some computation* can be expressed in the following way:

$$X^{\text{max}} = F \land ([\text{Act}]ff \lor \langle \text{Act} \rangle X).$$
It turns out to be the largest solution that is of interest here as we will argue for formally later.

Intuitively, the recursively defined formula above states that a process $p$ has a complete transition sequence all of whose states satisfy the formula $F$ if, and only if,

- $p$ itself satisfies $F$, and
- either $p$ has no outgoing transition (in which case $p$ will satisfy the formula $\text{[Act]}ff$) or $p$ has a transition leading to a state that has a complete transition sequence all of whose states satisfy the formula $F$.

A process $p$ satisfies the property $\text{Even}(F)$, read ‘eventually $F$’, if each of its complete transition sequences will contain at least one state that has the property $F$. This means that either $p$ satisfies $F$, or $p$ can perform some transition and every state that it can reach by performing a transition can itself eventually reach a state that has property $F$. This can be expressed by means of the following equation:

$$Y_{\text{min}} = F \lor (\langle \text{Act} \rangle tt \land \text{[Act]}Y).$$

In this case we are interested in the least solution because $\text{Even}(F)$ should only be satisfied by those processes that can be reached from $p$ by a finite number of transitions.

Note that the definitions of $\text{Safe}(F)$ and $\text{Even}(F)$, respectively $\text{Inv}(F)$ and $\text{Pos}(F)$, are mutually dual, i.e., they can be obtained from one another by replacing $\lor$ by $\land$, $[A]$ by $\langle A \rangle$ and $\min$ by $\max$. One can show that $\neg \text{Inv}(F) \equiv \text{Pos}(\neg F)$ and $\neg \text{Safe}(F) \equiv \text{Even}(\neg F)$, where we write $\neg$ for logical negation.

It is also possible to express that $F$ should be satisfied in each transition sequence until $G$ becomes true. There are two well known variants of this construction:

- $F U^* G$, the so-called strong until, that says that sooner or later $p$ reaches a state where $G$ is true and in all the states it reaches before this happens $F$ must hold;
- $F U^w G$, the so-called weak until, that says that $F$ must hold in all states $p$ reaches until it gets into a state where $G$ holds (but maybe this will never happen!).

We express these operators as follows:
\[ F \text{ } \mathcal{U} \text{ } G \overset{\min}{=} G \lor (F \land (\text{Act}) \# \land \text{[Act]}(FU^sG)), \text{ and } F \text{ } \mathcal{U}^w \text{ } G \overset{\max}{=} G \lor (F \land \text{[Act]}(FU^wG)). \]

It should be clear that, as the names indicate, strong until is a stronger condition than weak until. We can use the ‘until’ operators to express \( \text{Even}(F) \) and \( \text{Inv}(F) \). Thus \( \text{Even}(G) \equiv \# U^s G \) and \( \text{Inv}(F) \equiv F \mathcal{U}^w ff \).

Properties like ‘some time in the future’ and ‘until’ are examples of what we call temporal properties. Tempora is Latin—it is plural for tempus, which means ‘time’—, and a logic that expresses properties that depend on time is called temporal logic. The study of temporal logics is very old and can be traced back to Aristotle. Within the last 30 years, researchers in computer science have started showing interest in temporal logic as within this framework it is possible to express properties of the behaviour of programs that change over time (Clarke, E.A. Emerson and A.P. Sistla, 1986; Pnueli, 1977a).

The modal \( \mu \)-calculus (Kozen, 1983a) is a generalization of Hennessy-Milner logic with recursion that allows for largest and least fixed point definitions to be mixed freely. It has been shown that the modal \( \mu \)-calculus is expressive enough to describe any of the standard operators that occur in the framework of temporal logic. In this sense by extending Hennessy-Milner logic with recursion we obtain a temporal logic.

From the examples in this section we can see that least fixed points are used to express that something will happen sooner or later, whereas the largest fixed points are used to express invariance of some state of affairs during computations, or that something does not happen as a system evolves.

### 6.2 Syntax and semantics of HML with recursion

The first step towards introducing recursion in HML is to add variables to the syntax. To start with we only consider one recursively defined property. We will return to the more general case of properties defined by mutual recursion later.

The syntax for Hennessy-Milner-logic with one variable \( X \), denoted by \( \mathcal{M}_{\{X\}} \), is given by the following grammar:

\[ F ::= X \mid \# f f \mid F_1 \land F_2 \mid F_1 \lor F_2 \mid (a) F \mid [a] F. \]

Semantically a formula \( F \) (that may contain a variable \( X \)) is interpreted as a function \( O_F : 2^{\mathcal{P}roc} \rightarrow 2^{\mathcal{P}roc} \), that, given a set of processes that are assumed to satisfy \( X \), gives us the set of processes that satisfy \( F \).
Example 6.2 Consider the formula $F = \langle a \rangle X$ and let $\text{Proc}$ be the set of states in the transition graph in Figure 6.2. If $X$ is satisfied by $p_1$, then $\langle a \rangle X$ will be satisfied by $p_3$, i.e., we expect that

$$O_{\langle a \rangle X}(\{p_1\}) = \{p_3\}.$$ 

If the set of states satisfying $X$ is $\{p_1, p_2\}$ then $\langle a \rangle X$ will be satisfied by $\{p_1, p_3\}$. Therefore we expect to have that

$$O_{\langle a \rangle X}(\{p_1, p_2\}) = \{p_1, p_3\}.$$ 

What is the set $O_{\langle b \rangle X}(\{p_2\})$? ♦

The above intuition is captured formally in the following definition.

Definition 6.1 Let $(\text{Proc}, \text{Act}, \{ a \rightarrow \mid a \in \text{Act}\})$ be a labelled transition system. For each $S \subseteq \text{Proc}$ and formula $F$, we define $O_F(S)$ inductively as follows:

$$O_X(S) = S$$
$$O_{\emptyset}(S) = \text{Proc}$$
$$O_{\emptyset}(S) = \emptyset$$

$$O_{F_1 \land F_2}(S) = O_{F_1}(S) \cap O_{F_2}(S)$$
$$O_{F_1 \lor F_2}(S) = O_{F_1}(S) \cup O_{F_2}(S)$$

$$O_{\langle a \rangle F}(S) = \langle \cdot a \cdot \rangle O_F(S)$$
$$O_{[a] F}(S) = [\cdot a \cdot] O_F(S).$$ ♦
A few words of explanation for the above definition are in order here. Intuitively, the first equality in Definition 6.1 expresses the trivial observation that if we assume that $S$ is the set of states that satisfy $X$, then the set of states satisfying $X$ is $S$! The second equation states the, equally obvious, fact that every state satisfies $\top$ irrespective of the set of states that are assumed to satisfy $X$. The last equation instead says that to calculate the set of states satisfying the formula $[a]F$ under the assumption that the states in $S$ satisfy $X$, it is sufficient to

1. compute the set of states satisfying the formula $F$ under the assumption that the states in $S$ satisfy $X$, and then

2. find the collection of states that end up in that set no matter how they perform an $a$-labelled transition.

Exercise 6.4 Given the transition graph from Example 6.2, use the above definition to calculate $O_{[b]g \land [a]X}(\{p_2\})$.

One can show that for every formula $F$, the function $O_F$ is monotonic (see Definition 4.4) over the complete lattice $(2^{\text{Proc}}, \subseteq)$. In other words, for all subsets $S_1, S_2$ of $\text{Proc}$, if $S_1 \subseteq S_2$ then $O_F(S_1) \subseteq O_F(S_2)$.

Exercise 6.5 Show that $O_F$ is monotonic for all $F$. Consider what will happen if we introduce negation into our logic.

As mentioned before, the idea underlying the definition of the function $O_F$ is that if $[X] \subseteq \text{Proc}$ gives the set of processes that satisfy $X$, then $O_F([X])$ will be the set of processes that satisfy $F$. What is this set $[X]$ then? Syntactically we shall assume that $[X]$ is implicitly given by a recursive equation for $X$ of the form

$$X \overset{\text{min}}{=} F_X \text{ or } X \overset{\text{max}}{=} F_X .$$

As shown in the previous section, such an equation can be interpreted as the set equation

$$[X] = O_{F_X}([X]) .$$

(6.6)

As $O_{F_X}$ is a monotonic function over a complete lattice we know that (6.6) has solutions, i.e., that $O_{F_X}$ has fixed points. In particular Tarski’s Fixed Point Theorem (see Theorem 4.1) gives us that there is a unique largest fixed point, denoted by $\text{FIX} O_{F_X}$, and also a unique least one, denoted by $\text{fix} O_{F_X}$, given respectively by

$$\text{FIX} O_{F_X} = \bigcup \{S \subseteq \text{Proc} \mid S \subseteq O_{F_X}(S)\} \text{ and } \text{fix} O_{F_X} = \bigcap \{S \subseteq \text{Proc} \mid O_{F_X}(S) \subseteq S\} .$$
A set $S$ with the property that $S \subseteq O_F X (S)$ is called a post fixed point for $O_F X$. Correspondingly $S$ is pre fixed point for $O_F X$ if $O_F X (S) \subseteq S$.

In what follows, for a function $f : 2^{\mathsf{Proc}} \rightarrow 2^{\mathsf{Proc}}$ we define

\[ f^0 = \text{id}_{2^{\mathsf{Proc}}} \text{ (the identity function on } 2^{\mathsf{Proc}}), \text{ and} \]

\[ f^{m+1} = f \circ f^m . \]

When $\mathsf{Proc}$ is finite we have the following characterization of the largest and least fixed points.

**Theorem 6.1** If $\mathsf{Proc}$ is finite then $\text{FIX } O_F X = (O_F X)^M (\mathsf{Proc})$ for some $M$ and $\text{fix } O_F X = (O_F X)^m (\emptyset)$ for some $m$.

**Proof:** This follows directly from the fixed point theorem for finite complete lattices. See Theorem 4.2 for the details.

The above theorem gives us an algorithm for computing the least and largest set of processes solving an equation of the form (6.6). Consider, by way of example, the formula

\[ X_{\text{max}} = F_X , \]

where $F_X = \langle b \rangle \mathbb{F} \land [b] X$. The set of processes in the labelled transition system

that satisfy this property is the largest solution to the equation

\[ [X] = (\langle \cdot \cdot b \rangle \{s, s_1, s_2, t, t_1\}) \cap [b][X] . \]

This solution is nothing but the largest fixed point of the set function defined by the right-hand side of the above equation—that is, the function mapping each set of states $S$ to the set

\[ O_F X (S) = (\langle \cdot b \rangle \{s, s_1, s_2, t, t_1\}) \cap [b]S . \]

Since we are looking for the largest fixed point of this function, we begin the iterative algorithm by taking $S = \{s, s_1, s_2, t, t_1\}$, the set of all states in our labelled
transition system. We therefore have that our first approximation to the largest fixed point is the set
\[
O_F \left( \{s, s_1, s_2, t, t_1\} \right) = \left( \langle \cdot b \cdot \rangle \{s, s_1, s_2, t, t_1\} \right) \cap \{s_1, s_2, t_1\} \cap \{s, s_1, s_2, t, t_1\} = \{s_1, s_2, t_1\}.
\]
Note that our candidate solution to the equation has shrunk in size, since an application of \(O_F\) to the set of all processes has removed the states \(s\) and \(t\) from our candidate solution. Intuitively, this is because, by applying \(O_F\) to the set of all states, we have found a reason why \(s\) and \(t\) do not afford the property specified by
\[
X_{\text{max}} = \langle b \rangle \# \land [b]X;
\]
namely that \(s\) and \(t\) do not have a \(b\)-labelled outgoing transition, and therefore that neither of them is in the set \(\langle \cdot b \cdot \rangle \{s, s_1, s_2, t, t_1\}\).

Following our iterative algorithm for the computation of a largest fixed point, we now apply the function \(O_F\) to the new candidate largest solution, namely \(\{s_1, s_2, t_1\}\). We now have that
\[
O_F \left( \{s_1, s_2, t_1\} \right) = \left( \langle \cdot b \cdot \rangle \{s, s_1, s_2, t, t_1\} \right) \cap [b] \{s_1, s_2, t_1\} = \{s_1, s_2, t_1\} \cap \{s, s_1, s_2, t, t_1\} = \{s_1, s_2, t_1\}.
\]
(You should convince yourselves that the above calculations are correct!) We have now found that \(\{s_1, s_2, t_1\}\) is a fixed point of the function \(O_F\). By Theorem 6.1, this is the largest fixed point and therefore states \(s_1, s_2\) and \(t_1\) are the only states in our labelled transition system that satisfy the property
\[
X_{\text{max}} = \langle b \rangle \# \land [b]X.
\]
This is in complete agreement with our intuition because those are the only states that can perform a \(b\)-action in all states that they can reach by performing sequences of \(b\)-labelled transitions.

Exercise 6.6 Consider the property
\[
Y_{\text{min}} = \langle b \rangle \# \lor \langle \{a, b\} \rangle Y.
\]
Use Theorem 6.1 to compute the set of processes in the labelled transition system above that satisfy this property. ♦
6.3 Largest fixed points and invariant properties

In this section we shall have a closer look at the meaning of formulae defined by means of largest fixed points. More precisely we consider an equation of the form

\[ X^{\text{max}} = F_X \]

and define \([X] \subseteq \text{Proc}\) by

\[
[X] = \text{FIX } O_{F_X}.
\]

We have previously given an informal argument for why invariant properties are obtained as largest fixed points. In what follows we will formalize this argument, and prove its correctness.

As we saw in the previous section, the property \(\text{Inv}(F)\) is obtained as the largest fixed point to the recursive equation

\[ X = F \land [\text{Act}]X. \]

We will now show that \(\text{Inv}(F)\) defined in this way indeed expresses that \(F\) holds at all states in all transitions sequences.

For this purpose we let \(\mathcal{I} : 2^{\text{Proc}} \rightarrow 2^{\text{Proc}}\) be the corresponding semantic function, i.e.,

\[
\mathcal{I}(S) = [F] \cap [\cdot \text{Act} \cdot]S.
\]

By Tarski’s Fixed Point Theorem this equation has exactly one largest solution given by

\[
\text{FIX } \mathcal{I} = \bigcup\{S \mid S \subseteq \mathcal{I}(S)\}.
\]

To show that \(\text{FIX } \mathcal{I}\) indeed characterizes precisely the set of processes for which all states in all computations satisfy the property \(F\), we need a direct (and obviously correct) formulation of this set. This is given by the set \(\text{Inv}\) defined as follows:

\[
\text{Inv} = \{p \mid p \xrightarrow{\sigma} p' \text{ implies } p' \in [F], \text{ for each } \sigma \in \text{Act}^*, p' \in \text{Proc}\}.
\]

The correctness of \(\text{Inv}(F)\) with respect to this description can now be formulated as follows.

**Theorem 6.2** For every labelled transition system \((\text{Proc}, \text{Act}, \{ \xrightarrow{a} \mid a \in \text{Act}\})\), it holds that \(\text{Inv} = \text{FIX } \mathcal{I}\).

**Proof:** We show the statement by proving each of the inclusions \(\text{Inv} \subseteq \text{FIX } \mathcal{I}\) and \(\text{FIX } \mathcal{I} \subseteq \text{Inv}\) separately.
INV \subseteq \text{FIX } I: \text{ To prove this inclusion it is sufficient to show that } INV \subseteq I(INV) \text{ (Why?). To this end, let } p \in INV. \text{ Then, for all } \sigma \in \text{Act}^* \text{ and } p' \in \text{Proc},
\begin{equation}
p^{\sigma} \rightarrow p' \text{ implies that } p' \in [F]. \tag{6.7}
\end{equation}
We must establish that \( p \in I(INV) \), or equivalently that \( p \in [F] \) and that \( p \in [\cdot\text{Act}\cdot]INV \). We obtain the first one of these two statements by taking \( \sigma = \varepsilon \) in (6.7) because \( p^{\varepsilon} \rightarrow p \) always holds.

To prove that \( p \in [\cdot\text{Act}\cdot]INV \), we have to show that, for each process \( p' \) and action \( a \),
\[ p^{a} \rightarrow p' \text{ implies } p' \in INV. \]
This is equivalent to proving that, for each sequence of actions \( \sigma' \) and process \( p'' \),
\[ p^{a} \rightarrow p' \text{ and } p'^{\sigma'} \rightarrow p'' \text{ imply } p'' \in [F]. \]
However, this follows immediately by letting \( \sigma = a\sigma' \) in (6.7).

\text{FIX } I \subseteq INV: \text{ First we note that, since } \text{FIX } I \text{ is a fixed point of } I, \text{ it holds that}
\text{FIX } I = [F] \cap [\cdot\text{Act}\cdot]\text{FIX } I. \tag{6.8}

To prove that \( \text{FIX } I \subseteq INV \), assume that \( p \in \text{FIX } I \) and that \( p^{\sigma} \rightarrow p' \). We shall show that \( p' \in [F] \) by induction on \( |\sigma| \), the length of \( \sigma \).

Base case \( \sigma = \varepsilon \): Then \( p = p' \) and therefore, by (6.8) and our assumption that \( p \in \text{FIX } I \), it holds that \( p' \in [F] \), which was to be shown.

Inductive step \( \sigma = a\sigma' \): Then \( p^{a} \rightarrow p'' ^{\sigma'} \rightarrow p' \) for some \( p'' \). By (6.8) and our assumption that \( p \in \text{FIX } I \), it follows that \( p'' \in \text{FIX } I \). As \( |\sigma'| < |\sigma| \) and \( p'' \in \text{FIX } I \), by the induction hypothesis we may conclude that \( p' \in [F] \), which was to be shown.

This completes the proof of the second inclusion.

The proof of the theorem is now complete. \hfill \Box

### 6.4 A game characterization for HML with recursion

Let us recall the definition of Hennessy-Milner logic with one recursively defined variable \( X \). The formulae are defined using the following abstract syntax
\[ F ::= X \mid t \mid f \mid F_1 \land F_2 \mid F_1 \lor F_2 \mid \langle a \rangle F \mid [a] F , \]
where \( a \in \text{Act} \) and there is exactly one defining equation for the variable \( X \), which is of the form

\[
X^\text{min} = F_X
\]

or

\[
X^\text{max} = F_X,
\]

where \( F_X \) is a formula of the logic which may contain occurrences of the variable \( X \).

Let \((\text{Proc}, \text{Act}, \{ \stackrel{a}{\rightarrow} | a \in \text{Act} \})\) be a labelled transition system and \( F \) a formula of Hennessy-Milner logic with one (recursively defined) variable \( X \). Let \( s \in \text{Proc} \). We shall describe a game between an ‘attacker’ and a ‘defender’ which has the following goal:

- the attacker is aiming to prove that \( s \not\models F \), while
- the defender is aiming to prove that \( s \models F \).

The configurations of the game are pairs of the form \((s, F)\) where \( s \in \text{Proc} \) and \( F \) is a formula of Hennessy-Milner logic with one variable \( X \). For every configuration we define the following successor configurations according to the structure of the formula \( F \) (here \( s \) is ranging over \text{Proc}):

- \((s, \#)\) and \((s, \#\#)\) have no successor configurations,
- \((s, F_1 \land F_2)\) and \((s, F_1 \lor F_2)\) both have two successor configurations, namely \((s, F_1)\) and \((s, F_2)\),
- \((s, (a) F)\) and \((s, [a] F)\) both have the successor configurations \((s', F)\) for every \( s' \) such that \( s \stackrel{a}{\rightarrow} s' \), and
- \((s, X)\) has only one successor configuration \((s, F_X)\), where \( X \) is defined via the equation \( X^\text{max} = F_X \) or \( X^\text{min} = F_X \).

A play of the game starting from \((s, F)\) is a maximal sequence of configurations formed by the players according to the following rules.

- The attacker picks up a successor configuration for every current configuration of the form \((s, F_1 \land F_2)\) and \((s, [a] F)\).
- The defender picks up a successor configuration for every current configuration of the form \((s, F_1 \lor F_2)\) and \((s, (a) F)\).
Note that the successor configuration of \((s, X)\) is always uniquely determined and we will denote this move by \((s, X) \rightarrow (s, F_X)\). (It is suggestive to think of these moves that unwind fixed points as moves made by a referee for the game.) Similarly successor configurations selected by the attacker will be denoted as \(\rightarrow^A\) moves and by the defender as \(\rightarrow^D\) moves.

We also notice that every play either

- terminates in \((s, \#)\) or \((s, \#\#)\), or
- it can be the case that the attacker (or the defender) gets stuck in the current configuration \((s, [a]F)\) (or \((s, \langle a \rangle F)\)) whenever \(s \not\rightarrow\), or
- the play is infinite.

The following rules decide who is the winner of a play.

- The attacker is a winner in every play ending in a configuration of the form \((s, \#\#)\) or in a play in which the defender gets stuck.
- The defender is a winner in every play ending in a configuration of the form \((s, \#)\) or in a play in which the attacker gets stuck.
- The attacker is a winner in every infinite play provided that \(X\) is defined via \(X_{\min} = F_X\); the defender is a winner in every infinite play provided that \(X\) is defined via \(X_{\max} = F_X\).

**Remark 6.1** The intuition for the least and largest fixed point is as follows. If \(X\) is defined as a least fixed point then the defender has to prove infinitely many rounds that the property is satisfied. If a play of the game is infinite, then the defender has failed to do so, and the attacker wins. If instead \(X\) is defined as a largest fixed point, then it is the attacker who has to disprove in finitely many rounds that the formula is satisfied. If a play of the game is infinite, then the attacker has failed to do so, and the defender wins. ♦

**Theorem 6.3** [Game Characterization] Let \((\text{Proc}, \text{Act}, \{\alpha \mid a \in \text{Act}\})\) be a labelled transition system and \(F\) a formula of Hennessy-Milner logic with one (recursively defined) variable \(X\). Let \(s \in \text{Proc}\). Then the following statements hold.

- \(s \models F\) if and only if the defender has a universal winning strategy starting from \((s, F)\).
- \(s \not\models F\) if and only if the attacker has a universal winning strategy starting from \((s, F)\).
The proof of this result is beyond the scope of this introductory textbook. We refer the reader to (Stirling, 2001) for a proof of the above result and more information on model checking games.

6.4.1 Examples of use

In this section let us consider the following labelled transition system.

Example 6.3 We start with an example which is not using any recursively defined variable. We shall demonstrate that $s \models [b](\langle b \rangle [b]ff \land \langle b \rangle [a]ff)$ by defining a universal winning strategy for the defender. As remarked before, we will use $\rightarrow^A$ to denote that the successor configuration was selected by the attacker and $\rightarrow^D$ to denote that it was selected by the defender. The game starts from

$$(s, [b](\langle b \rangle [b]ff \land \langle b \rangle [a]ff))$$.

Because $[b]$ is the topmost operation, the attacker selects the successor configuration and he has only one possibility, namely

$$(s, [b](\langle b \rangle [b]ff \land \langle b \rangle [a]ff)) \rightarrow^A (s_1, \langle b \rangle [b]ff \land \langle b \rangle [a]ff)$$.

Now the topmost operation is $\land$ so the attacker has two possibilities:

$$(s_1, \langle b \rangle [b]ff \land \langle b \rangle [a]ff) \rightarrow^A (s_1, \langle b \rangle [b]ff)$$

or

$$(s_1, \langle b \rangle [b]ff \land \langle b \rangle [a]ff) \rightarrow^A (s_1, \langle b \rangle [a]ff)$$.

We have to show that the defender wins from any of these two configurations (we have to find a universal winning strategy).

- From $(s_1, \langle b \rangle [b]ff)$ it is the defender who makes the next move; let him so play $(s_1, \langle b \rangle [b]ff) \rightarrow^D (s_2, [b]ff)$. Now the attacker should continue but $s_2 \not\rightarrow^b$ so he is stuck and the defender wins this play.

- From $(s_1, \langle b \rangle [a]ff)$ it is also the defender who makes the next move; let him play $(s_1, \langle b \rangle [a]ff) \rightarrow^D (s, [a]ff)$. Now the attacker should continue but $s \not\rightarrow^a$ so he is stuck again and the defender wins this play.
Hence the defender has a universal winning strategy.

Example 6.4 Let $X \equiv \langle a \rangle \# \lor \langle b \rangle X$. This property informally says that it is possible to perform a sequence of $b$ actions leading to a state where the action $a$ is enabled. We will show that $s \models X$ by defining a universal winning strategy for the defender starting from $(s, X)$. The strategy looks as follows (note that it consists solely of the defender’s moves $D \rightarrow$ or the referee’s $\rightarrow$ moves for expanding the variable $X$, so it is truly a universal winning strategy):

$$(s, X) \rightarrow (s, \langle a \rangle \# \lor \langle b \rangle X) \xrightarrow{D} (s, \langle b \rangle X) \xrightarrow{D} (s_1, X) \rightarrow$$

$$\rightarrow (s_1, \langle a \rangle \# \lor \langle b \rangle X) \xrightarrow{D} (s_1, \langle b \rangle X) \xrightarrow{D} (s_2, X) \rightarrow$$

$$\rightarrow (s_2, \langle a \rangle \# \lor \langle b \rangle X) \xrightarrow{D} (s_2, \langle a \rangle \#) \xrightarrow{D} (s_3, \#) .$$

According to the definition $(s_3, \#)$ is a winning configuration for the defender.

Example 6.5 Let $X \equiv \langle b \rangle \# \land [b]X$. This property informally says that along every path where the edges are labelled by the action $b$, the action $b$ never becomes disabled. It is easy to see that $s \not\models X$ and we will prove it by finding a universal winning strategy for the attacker starting from $(s, X)$. As before, the attacker’s strategy will not give any selection possibility to the defender and hence it is a universal one.

$$(s, X) \rightarrow (s, \langle b \rangle \# \land [b]X) \xrightarrow{A} (s, [b]X) \xrightarrow{A} (s_1, X) \rightarrow$$

$$\rightarrow (s_1, \langle b \rangle \# \land [b]X) \xrightarrow{A} (s_1, [b]X) \xrightarrow{A} (s_2, X) \rightarrow$$

$$\rightarrow (s_2, \langle b \rangle \# \land [b]X) \xrightarrow{A} (s_2, [b]\#) .$$

From the last configuration $(s_2, [b]\#)$ the defender is supposed to continue but he is stuck as $s_2 \not\rightarrow b$ and hence the attacker wins.

Example 6.6 Let $X \equiv \langle a \rangle \# \land [a]X$. This is the same property as in the previous example (with $a$ exchanged for $b$). We will show that $s_2 \models X$ by finding a universal winning strategy for the defender from $(s_2, X)$. In the first round we expand the variable $X$ by the move $(s_2, X) \rightarrow (s_2, \langle a \rangle \# \land [a]X)$ and in the second round the attacker can play either

$$(s_2, \langle a \rangle \# \land [a]X) \xrightarrow{A} (s_2, \langle a \# \rangle)$$

or

$$(s_2, \langle a \rangle \# \land [a]X) \xrightarrow{A} (s_2, [a]X) .$$
It is easy to see that the defender wins from the configuration \((s_2, (a)\#)\) by the move \((s_2, (a)\#) \xrightarrow{D} (s_3, \#)\), so we shall investigate only the continuation of the game from \((s_2, [a]X)\). The attacker has only the move \((s_2, [a]X) \xrightarrow{A} (s_3, X)\). After expanding the variable \(X\) the game continues from \((s_3, (a)\# \land [a]X)\). Again the attacker can play either

\[(s_3, (a)\# \land [a]X) \xrightarrow{A} (s_3, (a)\#)\]

or

\[(s_3, (a)\# \land [a]X) \xrightarrow{A} (s_3, [a]X)\, .\]

In the first case the attacker loses as before. In the second case, the only continuation of the game is \((s_3, [a]X) \xrightarrow{A} (s_3, X)\). However, we have already seen this configuration earlier in the game. To sum up, either the attacker loses in finitely many steps or the game can be infinite. As we consider the largest fixed point, in both cases the defender is the winner of the game.

\[\diamondsuit\]

**Example 6.7** Let \(X = \langle a \rangle\# \lor ([b]X \land \langle b \rangle\#)\). This property informally says that along each \(b\) labelled sequence there is eventually a state where the action \(a\) is enabled. We shall argue that \(s_1 \not\models X\) by finding a winning strategy for the attacker starting from \((s_1, X)\). The first move of the game is

\[(s_1, X) \rightarrow (s_1, (a)\# \lor ([b]X \land \langle b \rangle\#))\, ,\]

and then the defender has two options, namely

\[(s_1, (a)\# \lor ([b]X \land \langle b \rangle\#)) \xrightarrow{D} (s_1, (a)\#)\]

or

\[(s_1, (a)\# \lor ([b]X \land \langle b \rangle\#)) \xrightarrow{D} (s_1, [b]X \land \langle b \rangle\#)\, .\]

In the first case the defender loses as he is supposed to pick up an \(a\)-successor of the state \(s_1\) but \(s_1 \not\models A\). In the second case the attacker proceeds as follows.

\[(s_1, [b]X \land \langle b \rangle\#) \xrightarrow{A} (s_1, [b]X) \xrightarrow{A} (s, X)\, .\]

The game now continues from \((s, X)\) by the move

\[(s, X) \rightarrow (s, (a)\# \lor ([b]X \land \langle b \rangle\#))\, .\]

Again, if the defender plays \((s, (a)\# \lor ([b]X \land \langle b \rangle\#)) \xrightarrow{D} (s, (a)\#)\) then he loses in the next round, so the defender has to play

\[(s, (a)\# \lor ([b]X \land \langle b \rangle\#)) \xrightarrow{D} (s, [b]X \land \langle b \rangle\#)\, .\]
The attacker continues by \((s, [b]X \land \langle b \rangle t) \xrightarrow{A} (s, [b]X) \xrightarrow{A} (s_1, X)\) and the situation \((s_1, X)\) has already been seen before. This means that the game is infinite (unless the defender loses in finitely many rounds) and hence the attacker is the winner of the game (since we are considering a least fixed point).

**Exercise 6.7** Consider the labelled transition system

\[
\begin{array}{c}
 s_1 \xrightarrow{b} s_2 \\
 s \xrightarrow{a} t \xrightarrow{a} t_1 \xrightarrow{a} t_2
\end{array}
\]

Use the game characterization for HML with recursion to show that

1. \(s_1\) satisfies the formula
   
   \[X = \max (\langle b \rangle t \land [b]X)\;,
   \]

2. \(s\) satisfies the formula
   
   \[Y = \min (\langle b \rangle t \lor \langle a, b \rangle Y)\;,
   \]
   but \(t\) does not.

Find a recursively defined property that \(t\) satisfies and argue that it does so using the game characterization of satisfaction presented above.

### 6.5 Mutually recursive equational systems

As you may have noticed, so far we have only allowed one equation with one variable in our recursive definitions. A **mutually recursive equational system** has the form

\[
\begin{align*}
X_1 & = F_{X_1} \\
& \vdots \\
X_n & = F_{X_n}
\end{align*}
\]
where $\mathcal{X} = \{X_1, \ldots, X_n\}$ is a set of variables and, for $i \leq n$, the formula $F_{X_i}$ is in $\mathcal{M}_\mathcal{X}$, and can therefore contain any variable from $\mathcal{X}$. An example of such an equational system is

$$X = [a]Y$$

$$Y = \langle a \rangle X.$$ 

An equational system is sometimes given by specifying a (finite) set of variables $\mathcal{X}$ together with a declaration. A declaration is a function $D : \mathcal{X} \rightarrow \mathcal{M}_\mathcal{X}$ that associates a formula with each variable—$D(X) = F_X$ in the notation used above.

To define the semantics of such an equational system it is not enough to consider simply the complete lattice consisting of subsets of processes. Instead such a system is interpreted over $n$-dimensional vectors of sets of processes, where $n$ is the number of variables in $\mathcal{X}$. Thus the new domain is $\mathcal{D} = (2^{\mathcal{P}roc})^n$ ($n$-times cross product of $2^{\mathcal{P}roc}$ with itself) with a partial order defined ‘component wise’:

$$(S_1, \ldots, S_n) \leq (S'_1, \ldots, S'_n)$$ if $S_1 \subseteq S'_1$ and $S_2 \subseteq S'_2$ and $\ldots$ and $S_n \subseteq S'_n$.

$\mathcal{D}$ defined in this way yields a complete lattice with the least upper bound and the greatest lower bound also defined component wise:

$$\bigcup \{A_i^1, \ldots, A_i^n \mid i \in I\} = (\bigcup \{A_i^1 \mid i \in I\}, \ldots, \bigcup \{A_i^n \mid i \in I\})$$ and

$$\bigcap \{A_i^1, \ldots, A_i^n \mid i \in I\} = (\bigcap \{A_i^1 \mid i \in I\}, \ldots, \bigcap \{A_i^n \mid i \in I\}),$$

where $I$ is an index set. The semantic function $[D] : \mathcal{D} \rightarrow \mathcal{D}$ that is used to obtain the largest and least solutions of the system of recursive equations described by the declaration $D$ is obtained from the syntax in the following way:

$$[D](\llbracket X_1 \rrbracket, \ldots, \llbracket X_n \rrbracket) =$$

$$(\mathcal{O}_{F_{X_1}}(\llbracket X_1 \rrbracket, \ldots, \llbracket X_n \rrbracket), \ldots, \mathcal{O}_{F_{X_n}}(\llbracket X_1 \rrbracket, \ldots, \llbracket X_n \rrbracket)),$$ (6.9)

where each argument $\llbracket X_i \rrbracket$ ($1 \leq i \leq n$) can be replaced by an arbitrary $S \subseteq \mathcal{P}roc$. By analogy with our previous developments, for each formula $F$ in $\mathcal{M}_\mathcal{X}$, the set

$$\mathcal{O}_F(\llbracket X_1 \rrbracket, \ldots, \llbracket X_n \rrbracket)$$

stands for the set of processes that satisfy $F$ under the assumption that $\llbracket X_i \rrbracket$ is the collection of processes satisfying $X_i$, for each $1 \leq i \leq n$.

For each formula $F$ that may contain occurrences of the variables $X_1, \ldots, X_n$, the set $\mathcal{O}_F(\llbracket X_1 \rrbracket, \ldots, \llbracket X_n \rrbracket)$ is defined exactly as in Definition 6.1, but with

$$\mathcal{O}_{X_i}(\llbracket X_1 \rrbracket, \ldots, \llbracket X_n \rrbracket) = \llbracket X_i \rrbracket$$ (1 $\leq i \leq n$).
The function \([D]\) turns out to be monotonic over the complete lattice \((D, \leq)\), and we can obtain both the largest and least fixed point for the equational system in the same way as for the case of one variable.

Consider, for example, the mutually recursive formulae described by the system of equations below:

\[
X \overset{\text{max}}{=} (a)Y \land [a]Y \land [b]ff \\
Y \overset{\text{max}}{=} (b)X \land [b]X \land [a]ff .
\]

We wish to find out the set of states in the following labelled transition that satisfies the formula \(X\).

\[
\begin{array}{cccc}
s & \overset{a}{\searrow} & s_1 & \overset{a}{\rightarrow} & s_2 & \overset{a}{\rightarrow} & s_3 & \overset{b}{\nearrow} & s_2 \\
\overset{b}{\nearrow} & \overset{a}{\leftarrow} & s_1 & \overset{b}{\leftarrow} & s_2 & \overset{b}{\leftarrow} & s_3 & \overset{a}{\rightarrow} & s_2
\end{array}
\]

To this end, we can again apply the iterative algorithm for computing the largest fixed point of the function determined by the above system of equations. Note that, as formally explained before, such a function maps a pair of sets of states \((S_1, S_2)\) to the pair of states

\[
\left(\langle a \cdot \rangle S_2 \cap \langle a \cdot \rangle S_2 \cap \{s, s_2\}, \langle b \cdot \rangle S_1 \cap \langle b \cdot \rangle S_1 \cap \{s_1, s_3\}\right) . \tag{6.10}
\]

There

- \(S_1\) stands for the set of states that are assumed to satisfy \(X\),
- \(S_2\) stands for the set of states that are assumed to satisfy \(Y\),
- \(\langle a \cdot \rangle S_2 \cap \langle a \cdot \rangle S_2 \cap \{s, s_2\}\) is the set of states that satisfies the right-hand side of the defining equation for \(X\) under these assumptions, and
- \(\langle b \cdot \rangle S_1 \cap \langle b \cdot \rangle S_1 \cap \{s_1, s_3\}\) is the set of states that satisfies the right-hand side of the defining equation for \(Y\) under these assumptions.

To compute the largest solution to the system of equations above, we use the iterative algorithm provided by Theorem 6.1 starting from the top element in our complete lattice, namely the pair

\[
\left(\{s, s_1, s_2, s_3\}, \{s, s_1, s_2, s_3\}\right) .
\]

This corresponds to assuming that all states satisfy both \(X\) and \(Y\). To obtain the next approximation to the largest solution to our system of equations, we compute the pair (6.10) taking \(S_1 = S_2 = \{s, s_1, s_2, s_3\}\). The result is the pair

\[
\left(\{s, s_2\}, \{s_1, s_3\}\right) .
\]
Note that we have shrunk both components in our original estimate to the largest solution. This means that we have not yet found the largest solution we are looking for. We therefore compute the pair (6.10) again taking the above pair as our new input \((S_1, S_2)\). You should convince yourselves that the result of this computation is the pair 

\[ (\{s, s_2\}, \{s_1\}) . \]

Note that the first component in the pair has not changed since our previous approximation, but that \(s_3\) has been removed from the second component. This is because at this point we have discovered, for instance, that \(s_3\) does not afford a \(b\)-labelled transition ending up in either \(s\) or \(s_2\).

Since we have not yet found a fixed point, we compute the pair (6.10) again, taking \((\{s, s_2\}, \{s_1\})\) as our new input \((S_1, S_2)\). The result of this computation is the pair 

\[ (\{s\}, \{s_1\}) . \]

Intuitively, at this iteration we have discovered a reason why \(s_2\) does not afford property \(X\)—namely that \(s_2\) has an \(a\)-labelled transition leading to state \(s_3\), which, as we saw before, does not have property \(Y\).

If we now compute the pair (6.10) again, taking \((\{s\}, \{s_1\})\) as our new input \((S_1, S_2)\), we obtain \((\{s\}, \{s_1\})\). We have therefore found the largest solution to our system of equations. It follows that process \(s\) satisfies \(X\) and process \(s_1\) satisfies \(Y\).

**Exercise 6.8**

1. Show that \(((2^{Proc})^n, \leq, \bigsqcup, \bigsqcap)\), with \(\leq, \bigsqcup\) and \(\bigsqcap\) defined as described in the text above, is a complete lattice.

2. Show that (6.9) defines a monotonic function 

\[ [D] : (2^{Proc})^n \rightarrow (2^{Proc})^n . \]

3. Compute the least and largest solutions of the system of equations 

\[
X = [a]Y \\
Y = \langle a \rangle X
\]

over the transition system associated with the CCS term

\[
A_0 = a.A_1 + a.a.0 \\
A_1 = a.A_2 + a.0 \\
A_2 = a.A_1 .
\]
6.6 Characteristic properties

The characterization theorem for bisimulation equivalence in terms of Hennessy-Milner logic (Theorem 5.1 on page 111) tells us that if our transition system is image finite, the equivalence classes of bisimulation equivalence are completely characterized by the logic—see (Hennessy and Milner, 1985) for the original reference. More precisely, for image finite processes, the equivalence class that contains \( p \) consists exactly of the set of processes that satisfy the same formulae in HML as \( p \)—that is, letting \([p] \sim = \{ q \mid q \sim p \}\), we have that
\[
[p] \sim = \{ q \mid \text{\( F \)} \implies \text{\( q \)} \subseteq \text{\( F \)}, \text{for each } F \in \mathcal{M} \} .
\]

**Exercise 6.9** Note that in the above rephrasing of the characterization theorem for HML, we only require that each formula satisfied by \( p \) is also satisfied by \( q \), but not that the converse also holds. Show, however, that if \( q \) satisfies all the formulae in HML satisfied by \( p \), then \( p \) and \( q \) satisfy the same formulae in HML.

In this section we will show that if our transition system is finite, by extending the logic with recursion, we can characterize the equivalence classes for strong bisimulation with a single formula. The formula that characterizes the bisimulation equivalence class for \( p \) is called the characteristic formula for \( p \), and will use the facility for mutually recursive definitions we introduced in Section 6.5. (Since the material in this section depends on that in Section 6.5, you might wish to review your knowledge of the syntax and semantics for mutually recursive formulae while reading the technical material to follow.) That such a characteristic formula is unique from a semantic point of view is obvious as the semantics for such a formula is exactly the equivalence class \([p] \sim \).

Our aim in this section is therefore, given a process \( p \) in a finite transition system, to find a formula \( F_p \in \mathcal{M}_X \) for a suitable set of variables \( X \), such that for all processes \( q \)
\[
q \models F_p \iff q \sim p .
\]

Let us start by giving an example that shows that in general bisimulation equivalence cannot be characterized by a recursion free formula.

**Example 6.8** Assume that \( \text{Act} = \{ a \} \) and that the process \( p \) is given by the equation
\[
X \overset{\text{def}}{=} a.X .
\]

We will show that \( p \) cannot be characterized up to bisimulation equivalence by a single recursion free formula. To see this we assume that such a formula exists and
show that this leads to a contradiction. Towards a contradiction, we assume that for some $F_p \in \mathcal{M}$,

$$\llbracket F_p \rrbracket = \llbracket p \rrbracket_{\sim}.$$  \hfill (6.11)

In particular we have that

$$p \models F_p \quad \text{and} \quad (q \models F_p \text{ implies } q \sim p, \text{ for each } q).$$  \hfill (6.12)

We will obtain contradiction by proving that (6.12) cannot hold for any formula $F_p$. Before we prove our statement we have to introduce some notation.

Recall that, by the modal depth of a formula $F$, notation $md(F)$, we mean the maximum number of nested occurrences of the model operators in $F$. Formally this is defined by the following recursive definition:

1. $md(\#) = md(ff) = 0$,
2. $md(\langle a \rangle F) = md(\langle a \rangle F) = 1 + md(F)$,
3. $md(F_1 \lor F_2) = md(F_1 \land F_2) = \max\{md(F_1), md(F_2)\}$.

Next we define a sequence $p_0, p_1, p_2, \ldots$ of processes inductively as follows:
1. \( p_0 = 0 \),
2. \( p_{i+1} = a.p_i \).

(The processes \( p \) and \( p_i \), for \( i \geq 1 \), are depicted in Figure 6.3.) Observe that each process \( p_i \) can perform a sequence of \( i \) \( a \)-labelled transitions in a row and terminate in doing so. Moreover, this is the only behaviour that \( p_i \) affords.

Now we can prove the following:

\[
p \models F \implies p_{\text{md}(F)} \models F, \text{ for each } F. \tag{6.13}
\]

The statement in (6.13) can be proven by structural induction on \( F \) and is left as an exercise for the reader. As obviously \( p \) and \( p_n \) are not bisimulation equivalent for any \( n \) (why?), the statement in (6.13) contradicts (6.12). Indeed, (6.12) and (6.13) imply that \( p \) is bisimilar to \( p_k \), where \( k \) is the modal depth of the formula \( F_p \).

As (6.12) is a consequence of (6.11), we can therefore conclude that no recursion free formula \( F_p \) can characterize the process \( p \) up to bisimulation equivalence.

\[\Box\]

**Exercise 6.10** Prove statement (6.13).

\[\Box\]

**Exercise 6.11 (Recommended)** Before reading on, you might want to try and define a characteristic formula for some processes for which HML suffices. If you fancy this challenge, we encourage you to read Example 6.9 to follow for inspiration.

Assume that \( a \) is the only action. For each \( i \geq 1 \), construct an HML formula that is the characteristic formula for process \( p_i \) in Figure 6.3. Hint: First give a characteristic formula for \( p_1 \). Next show how to construct a characteristic formula for \( p_{i+1} \) from that for \( p_i \).

\[\Box\]

Example 6.8 shows us that in order to obtain a characteristic formula even for finite labelled transition systems we need to make use of the recursive extension of Hennessy-Milner logic.

The construction of the characteristic formula involves two steps. First of all, we need to construct an equational system that describes the formula; next we should decide whether to adopt the least or the largest solution to this system. We start our search for the characteristic formula by giving the equational system, and choose the suitable interpretation for the fixed points afterwards.

We start by assuming that we have a finite transition system

\[
(\{p_1, \ldots, p_n\}, \text{Act}, \rightarrow)
\]
and a set of variables $\mathcal{X} = \{X_{p_1}, \ldots, X_{p_n}, \ldots\}$ that contains (at least) as many variables as there are states in the transition system. Intuitively $X_p$ is the syntactic symbol for the characteristic formula for $p$ and its meaning will be given in terms of an equational system.

A characteristic formula for a process has to describe both which actions the process can perform, which action it cannot perform and what happens to it after it has performed each action. The following example illustrates these issues.

**Example 6.9** If a coffee machine is given by Figure 6.4, we can construct a characteristic formula for it as follows.

![Figure 6.4: The nice coffee machine $gkm$](image)

Let $gkm$ be the initial state of the coffee machine. Then we see that $gkm$ can perform an $m$-action and that this is the only action it can perform in this state. The picture also shows us that, by performing the $m$ action, $gkm$ will necessarily end up in state $q$. This can be expressed as follows:

1. $gkm$ can perform $m$ and become $q$.
2. No matter how $gkm$ performs $m$ it becomes $q$.
3. $gkm$ cannot perform any action other than $m$.

If we let $X_{gkm}$ and $X_q$ denote the characteristic formula for $q$ and $gkm$ respectively, $X_{gkm}$ can be expressed as

$$X_{gkm} \equiv \langle m \rangle X_q \land [m]X_q \land \{\overline{t}, \overline{k}\}]ff,$$

where $\langle m \rangle X_q$ expresses property 1 above, $[m]X_q$ expresses property 2 and $\{\overline{t}, \overline{k}\}]ff$ expresses property 3. To obtain the characteristic formula for $gkm$ we have to define a recursive formula for $X_q$ following the same strategy. We observe that $q$ can perform two actions, namely $\overline{t}$ and $\overline{k}$, and in both cases it becomes $gkm$. $X_q$ can therefore be expressed as

$$X_q \equiv \langle \overline{t} \rangle X_{gkm} \land \langle \overline{k} \rangle X_{gkm} \land \{\overline{t}, \overline{k}\}]X_{gkm} \land [m]ff.$$
In the recursive formula above, the first conjunct \( \langle t \rangle X_{gkm} \) states that a process that is bisimilar to \( q \) should be able to perform a \( t \)-labelled transition and thereby end up in a state that is bisimilar to \( gkm \)—that is, that satisfies the characteristic property \( X_{gkm} \) for state \( gkm \). The interpretation of the second conjunct is similar. The third conjunct instead states that all of the outgoing transitions from a state that is bisimilar to \( q \) that are labelled with \( t \) or \( k \) will end up in a state that is bisimilar to \( gkm \). Finally, the fourth and last conjunct says that a process that is bisimilar to \( q \) cannot perform action \( m \).

Now we can generalize the strategy employed in the above example as follows. Let

\[
\text{Der}(a, p) = \{ p' \mid p \xrightarrow{a} p' \}
\]

be the set of states that can be reached from \( p \) by performing action \( a \). If \( p' \in \text{Der}(a, p) \) and \( p' \) has a characteristic property \( X_{p'} \), then \( p \) has the property \( \langle a \rangle X_{p'} \). We therefore have that

\[
p \models \bigwedge_{a, p', p \xrightarrow{a} p'} \langle a \rangle X_{p'}.
\]

Furthermore, if \( p \xrightarrow{a} p' \) then \( p' \in \text{Der}(a, p) \). Therefore \( p \) has the property

\[
[a] \bigvee_{p', p \xrightarrow{a} p'} X_{p'},
\]

for each action \( a \). The above property states that, by performing action \( a \), process \( p \) (and any other process that is bisimilar to it) must become a process satisfying the characteristic property of a state in \( \text{Der}(a, p) \). (Note that if \( p \xrightarrow{a} \), then \( \text{Der}(a, p) \) is empty. In that case, since an empty disjunction is just the formula \( \mathbf{ff} \), the above formula becomes simply \( [a] \mathbf{ff} \)—which is what we would expect.)

Since action \( a \) is arbitrary, we have that

\[
p \models \bigwedge_{a} [a] \bigvee_{p', p \xrightarrow{a} p'} X_{p'}.
\]

If we summarize the above requirements, we have that

\[
p \models \bigwedge_{a, p', p \xrightarrow{a} p'} \langle a \rangle X_{p'} \land [a] \bigvee_{p', p \xrightarrow{a} p'} X_{p'}.
\]

As this property is apparently a complete description of the behaviour of process \( p \), this is our candidate for its characteristic property. \( X_{p} \) is therefore defined as
a solution to the equational system obtained by giving the following equation for each \( q \in \text{Proc} \):

\[
X_q \equiv \bigwedge_{a,q',q \xrightarrow{a} q'} \langle a \rangle X_{q'} \land \bigwedge_a \big[ a \big] \lor \bigvee_{q',q \xrightarrow{a} q'} X_{q'} . \tag{6.14}
\]

The solution can either be the least or the largest one (or something in between for what we know at this stage).

The following example shows that the least solution to (6.14) in general does not yield the characteristic property for a process.

**Example 6.10** Let \( p \) be the process given in Figure 6.5. In this case, assuming for the sake of simplicity that \( a \) is the only action, the equational system obtained by using (6.14) will have the form

\[
X_p \overset{\text{min}}{=} \langle a \rangle X_p \land [a] X_p .
\]

Since \( \langle a \rangle \emptyset = \emptyset \), you should be able to convince yourselves that \([X_p] = \emptyset\) is the least solution to this equation. This corresponds to taking \( X_p = \text{ff} \) as the characteristic formula for \( p \). However, \( p \) does not have the property \( \text{ff} \), which therefore cannot be the characteristic property for \( p \). ♦

In what follows we will show that the largest solution to (6.14) yields the characteristic property for all \( p \in \text{Proc} \). (Those amongst you who read Section 4.3 will notice that this is in line with our characterization of bisimulation equivalence as the largest fixed point of a suitable monotonic function.) This is the content of the following theorem, whose proof you can skip unless you are interested in the mathematical developments.

**Theorem 6.4** Let \((\text{Proc}, \text{Act}, \rightarrow)\) be a finite transition system and, for each \( p \in \text{Proc} \), let \( X_p \) be defined by

\[
X_p \overset{\text{max}}{=} \bigwedge_{a,p',p \xrightarrow{a} p'} \langle a \rangle X_{p'} \land \bigwedge_a \big[ a \big] \lor \bigvee_{p',p \xrightarrow{a} p'} X_{p'} . \tag{6.15}
\]
CHAPTER 6. HML WITH RECURSION

Then $X_p$ is the characteristic property for $p$—that is, $q \models X_p$ iff $p \sim q$, for each $q \in \text{Proc}$.

The assumption that $\text{Proc}$ and $\text{Act}$ be finite ensures that there is only a finite number of variables involved in the definition of the characteristic formula and that we only obtain a formula with finite conjunctions and disjunctions on the right-hand side of each equation.

In the proof of the theorem we will let $D_K$ be the declaration defined by

$$D_K(X_p) = \bigwedge_{a,p \xrightarrow{a} p'} \langle a \rangle X_{p'} \land \bigwedge_{a} [a] \bigvee_{p', p \xrightarrow{a} p'} X_{p'}.$$  

From our previous discussion, we have that $X_p$ is the characteristic property for $p$ if and only if for the largest solution $[X_p]$, where $p \in \text{Proc}$, we have that $[[X_p]] = [p]_\sim$. In what follows, we write $q \models_{\text{max}} X_p$ if $q$ belongs to $[X_p]$ in the largest solution for $D_K$.

In order to prove Theorem 6.4, we shall establish the following two statements separately, for each process $q \in \text{Proc}$:

1. if $q \models_{\text{max}} X_p$, then $p \sim q$, and
2. if $p \sim q$, then $q \models_{\text{max}} X_p$.

As the first step in the proof of Theorem 6.4, we prove the following lemma to the effect that the former statement holds.

**Lemma 6.1** Let $X_p$ be defined as in (6.15). Then, for each $q \in \text{Proc}$, we have that

$$q \models_{\text{max}} X_p \Rightarrow p \sim q.$$  

**Proof:** Let $R = \{ (p, q) \mid q \models_{\text{max}} X_p \}$. We will prove that $R$ is a bisimulation, and thus that $p \sim q$ whenever $q \models_{\text{max}} X_p$. To this end, we have to prove the following two claims, where $b$ is an arbitrary action in $\text{Act}$ and $p_1, q_1$ are processes in $\text{Proc}$.

a) $(p, q) \in R$ and $p \xrightarrow{b} p_1 \Rightarrow \exists q_1. q \xrightarrow{b} q_1$ and $(p_1, q_1) \in R$.

b) $(p, q) \in R$ and $q \xrightarrow{b} q_1 \Rightarrow \exists p_1. p \xrightarrow{b} p_1$ and $(p_1, q_1) \in R$.

We prove these two claims separately.

**a)** Assume that $(p, q) \in R$ and $p \xrightarrow{b} p_1$. This means that

$$q \models_{\text{max}} X_p \text{ and } p \xrightarrow{b} p_1.$$  

From the definitions of $X_p$ and $D_K$ it follows that

$$q \models_{\text{max}} \left( \bigwedge_{a,p' \vdash a'} \langle a \rangle X_{p'} \right) \wedge \left( \bigwedge_{a} \bigvee_{p',p' \vdash a'} X_{p'} \right).$$

As $p \xrightarrow{b} p_1$ we get that $q \models_{\text{max}} \langle b \rangle X_{p_1}$, which means that, for some $q_1 \in \text{Proc}$,

$$q \xrightarrow{b} q_1 \text{ and } q_1 \models_{\text{max}} X_{p_1}.$$

By the definition of $R$, we have that there is a $q_1$ such that

$$q \xrightarrow{b} q_1 \text{ and } (p_1, q_1) \in R,$$

which was to be shown.

b) Assume that $(p, q) \in R$ and $q \xrightarrow{b} q_1$. This means that

$$q \models_{\text{max}} X_p \text{ and } q \xrightarrow{b} q_1.$$

As before, since $q \models_{\text{max}} X_p$, we have that

$$q \models_{\text{max}} \left( \bigwedge_{a,p,p' \vdash a'} \langle a \rangle X_{p'} \right) \wedge \left( \bigwedge_{a} \bigvee_{p',p' \vdash a'} X_{p'} \right).$$

In particular, it follows that

$$q \models_{\text{max}} \bigvee_{p',p' \xrightarrow{b} p'} X_{p'}.$$

As we know that $q \xrightarrow{b} q_1$, we obtain that

$$q_1 \models_{\text{max}} \bigvee_{p',p' \xrightarrow{b} p'} X_{p'}.$$

Therefore there must exist a $p_1$ such that $q_1 \models_{\text{max}} X_{p_1}$ and $p \xrightarrow{b} p_1$.

We have therefore proven that

$$\exists p_1 \text{ such that } p \xrightarrow{b} p_1 \text{ and } (p_1, q_1) \in R,$$

which was to be shown.
We have now shown that $R$ is a bisimulation, and therefore that

\[ q \models_{\text{max}} X_p \text{ implies } p \sim q . \]

This proves the lemma.

The following lemma completes the proof of our main theorem of this section. In the statement of this result, and in its proof, we assume for notational convenience that $\text{Proc} = \{p_1, \ldots, p_n\}$.

**Lemma 6.2** \([p_1] \sim, \ldots, [p_n] \sim \subseteq [D_K]([p_1] \sim, \ldots, [p_n] \sim).\)

**Proof:** Assume that $q \in [p] \sim$, where $p$ is one of $p_1, \ldots, p_n$. To prove our claim, it is sufficient to show that $q \in \bigcap a, p \stackrel{a}{\to} p' \cdot \cdot \cdot \cdot \cdot [\cdot a \cdot] \cdot \cdot \cdot \cdot [p'] \sim \cdot \cdot \cdot \cdot \cdot \cdot \cdot$. (Can you see why?) The proof can be divided into two parts, namely:

1) $q \in \bigcap_a [a \cdot] \cdot [p] \sim \cdot$ and
2) $q \in \bigcup_{a \cdot p \stackrel{a}{\to} p'} [p'] \sim \cdot$ .

We proceed by proving these claims in turn.

1) We recall that $q \sim p$. Assume that $p \stackrel{a}{\to} p'$ for some action $a$ and process $p'$. Then there is a $q'$, where $q \stackrel{a}{\to} q'$ and $q' \sim p'$. We have therefore shown that, for all $a$ and $p'$, there is a $q'$ such that

\[ q \stackrel{a}{\to} q' \text{ and } q' \in [p'] \sim . \]

This means that, for each $a$ and $p'$ such that $p \stackrel{a}{\to} p'$, we have that

\[ q \in [a \cdot] [p'] \sim . \]

We may therefore conclude that

\[ q \in \bigcap_{a, p' \stackrel{a}{\to} p'} [a \cdot] [p'] \sim , \]

which was to be shown.
2) Let $a \in \text{Act}$ and $q \xrightarrow{a} q'$. We have to show that $q' \in \bigcup_{p', p \xrightarrow{a} p'} [p']_\sim$. To this end, observe that, as $q \xrightarrow{a} q'$ and $p \sim q$, there exists a $p'$ such that $p \xrightarrow{a} p'$ and $p' \sim q'$. For this $q'$ we have that $q' \in [p']_\sim$. We have therefore proven that, for all $a$ and $q'$,

$$q \xrightarrow{a} q' \Rightarrow \exists p'. p \xrightarrow{a} p' \text{ and } q \in [p']_\sim,$$

which is equivalent to

$$q \in \bigcap_a \left[ X_p \right] \bigcup_p \left[ p' \right]_\sim.$$

Statements 1) and 2) above give that:

$$([p_1]_\sim, \ldots, [p_n]_\sim) \subseteq \left[ D_K \right] ([p_1]_\sim, \ldots, [p_n]_\sim),$$

which was to be shown. \hfill \Box

Theorem 6.4 can now be expressed as the following lemma, whose proof completes the argument for that result.

**Lemma 6.3** For each $p \in \text{Proc}$ we have that $[X_p] = [p]_\sim$.

**Proof:** By Lemma 6.2 we get that

$$([p_1]_\sim, \ldots, [p_n]_\sim) \subseteq ([X_{P_1}], \ldots, [X_{P_n}]),$$

which means that

$$[p]_\sim \subseteq [X_p]$$

for each $p \in \text{Proc}$. (Why?) Furthermore Lemma 6.1 gives that $[X_p] \subseteq [p]_\sim$ for every $p \in \text{Proc}$, which proves the statement of the lemma. \hfill \Box

**Exercise 6.12** What is the characteristic formula for the processes $p$ and $q$ in Figure 6.1? ♦

**Exercise 6.13** Define characteristic formulae for the simulation and ready simulation preorders as defined in Definitions 3.17 and 3.18, respectively. ♦
6.7 Mixing largest and least fixed points

Assume that we are interested in using HML with recursive definitions to specify the following property of systems:

It is possible for the system to reach a state which has a livelock.

We already saw on page 120 how to describe a property of the form ‘it is possible for the system to reach a state satisfying $F$’ using the template formula $\text{Pos}(F)$, namely

$$\text{Pos}(F) = F \lor (\text{Act})\text{Pos}(F).$$

Therefore, all that we need do to specify the above property using HML with recursion is to ‘plug in’ a specification of the property ‘the state has a livelock’ in lieu of $F$. How can we describe a property of the form ‘the state has a livelock’ using HML with recursion? A livelock is an infinite sequence of internal steps of the system. So a state $p$ in a labelled transition system has a livelock if it affords a computation of the form

$$p = p_0 \overset{\tau}{\to} p_1 \overset{\tau}{\to} p_2 \overset{\tau}{\to} p_3 \overset{\tau}{\to} \cdots$$

for some sequence of states $p_1, p_2, p_3, \ldots$. In other words, a state $p$ has a livelock now if it affords a $\tau$-labelled transition leading to a state $p_1$ which has a livelock now. This immediately suggests the following recursive specification of the property $\text{LivelockNow}$:

$$\text{LivelockNow} = (\tau)\text{LivelockNow}.$$

As usual, we are faced with a choice in selecting a suitable solution for the above equation. Since we are specifying a state of affairs that should hold forever, in this case we should select the largest solution to the equation above. It follows that our HML specification of the property ‘the state has a livelock’ is

$$\text{LivelockNow}^{\text{max}} = (\tau)\text{LivelockNow}.$$

Exercise 6.14 What would be the least solution of the above equation?

Exercise 6.15 (Mandatory) Consider the labelled transition system below.

$$
\begin{array}{c}
s \xrightarrow{a} p \xrightarrow{\tau} q \xrightarrow{\tau} r \\
\tau
\end{array}
$$
Use the iterative algorithm for computing the set of states in that labelled transition system that satisfies the formula LivelockNow defined above.

Exercise 6.16 This exercise is for those amongst you who feel they need more practice in computing fixed points using the iterative algorithm.

Consider the labelled transition system below.

\[
\begin{array}{c}
s \xrightarrow{\tau} \ xarrow{\tau} s_1 \xleftarrow{a} s_2 \xrightarrow{\tau} s_3 \xrightarrow{\tau} p
\end{array}
\]

Use the iterative algorithm for computing the set of states in that labelled transition system that satisfies the formula LivelockNow defined above.

In light of the above discussion, a specification of the property mentioned at the beginning of this section using HML with recursive definitions can be given using the following system of equations:

\[
\begin{align*}
\text{Pos}(\text{LivelockNow})_{\text{min}} &= \text{LivelockNow} \lor \langle \text{Act} \rangle \text{Pos}(\text{LivelockNow}) \\
\text{LivelockNow}_{\text{max}} &= \langle \tau \rangle \text{LivelockNow}.
\end{align*}
\]

This looks natural and innocuous. However, first appearances can be deceiving! Indeed, the equational systems we have considered so far have only allowed us to express formulae purely in terms of largest or least solutions to systems of recursion equations. (See Section 6.5.) For instance, in defining the characteristic formulae for bisimulation equivalence, we only used systems of equations in which the largest solution was sought for all of the equations in the system.

Our next question is whether we can extend our framework in such a way that it can treat systems of equations with mixed solutions like the one describing the formula \( \text{Pos}(\text{LivelockNow}) \) above. How can we, for instance, compute the set of processes in the labelled transition system

\[
\begin{array}{c}
s \xrightarrow{a} p \xrightarrow{\tau} q \xrightarrow{\tau} r \xrightarrow{\tau} p
\end{array}
\]

that satisfy the formula \( \text{Pos}(\text{LivelockNow}) \)? In this case, the answer is not overly difficult. In fact, you might have already noted that we can compute the set of processes satisfying the formula \( \text{Pos}(\text{LivelockNow}) \) once we have in our hands the collection of processes satisfying the formula LivelockNow. As you saw in Exercise 6.15, the only state in the above labelled transition system satisfying the
formula LivelockNow is \( p \). Therefore, we may obtain the collection of states satisfying the formula \( \text{Pos}(\text{LivelockNow}) \) as the least solution of the set equation

\[
S = \{ p \} \cup \langle \cdot \text{Act} \cdot \rangle S ,
\]

where \( S \) ranges over subsets of \( \{ s, p, q, r \} \). We can calculate the least solution of this equation using the iterative methods we introduced in Section 6.2.

Since we are looking for the least solution of the above equation, we begin by obtaining our first approximation \( S^{(1)} \) to the solution by computing the value of the expression on the right-hand side of the equation when \( S = \emptyset \), which is the least element in the complete lattice consisting of the subsets of \( \{ s, p, q, r \} \) ordered by inclusion. We have that

\[
S^{(1)} = \{ p \} \cup \langle \cdot \text{Act} \cdot \rangle \emptyset = \{ p \} .
\]

Intuitively, we have so far discovered the (obvious!) fact that \( p \) has a possibility of reaching a state where a livelock may arise because \( p \) has a livelock now.

Our second approximation \( S^{(2)} \) is obtained by computing the set obtained by evaluating the expression on the right-hand side of equation (6.16) when \( S = S^{(1)} = \{ p \} \). The result is

\[
S^{(2)} = \{ p \} \cup \langle \cdot \text{Act} \cdot \rangle \{ p \} = \{ s, p \} .
\]

Intuitively, we have now discovered the new fact that \( s \) has a possibility of reaching a state where a livelock may arise because \( s \) has a transition leading to \( p \), which, as we found out in the previous approximation, has itself a possibility of reaching a livelock.

You should now be able to convince yourselves that the set \( \{ s, p \} \) is indeed a fixed point of equation (6.16)—that is, that

\[
\{ s, p \} = \{ p \} \cup \langle \cdot \text{Act} \cdot \rangle \{ s, p \} .
\]

It follows that \( \{ s, p \} \) is the least solution of equation (6.16), and that the states \( s \) and \( p \) are the only ones in our example labelled transition system that satisfy the formula \( \text{Pos}(\text{LivelockNow}) \). This makes perfect sense intuitively because \( s \) and \( p \) are the only states in that labelled transition system that afford a sequence of transitions leading to a state from which an infinite computation consisting of \( \tau \)-labelled transitions is possible. (In case of \( p \), this sequence is empty since \( p \) can embark in a \( \tau \)-loop immediately.)

Note that we could find the set of states satisfying \( \text{Pos}(\text{LivelockNow}) \) by first computing \( \llbracket \text{LivelockNow} \rrbracket \), and then using this set to compute

\[
\llbracket \text{Pos}(\text{LivelockNow}) \rrbracket ,
\]
because the specification of the formula LivelockNow was independent of that $Pos(LivelockNow)$. In general, we can apply this strategy when the collection of equations can be partitioned into a sequence of ‘blocks’ such that

- the equations in the same block are all either largest fixed point equations or least fixed equations, and
- equations in each block only use variables defined in that block or in preceding ones.

The following definition formalizes this class of systems of equations.

**Definition 6.2** A $n$-nested mutually recursive equational system $E$ is an $n$-tuple

$$\langle (D_1, X_1, m_1), (D_2, X_2, m_2), \ldots, (D_n, X_n, m_n) \rangle,$$

where the $X_i$s are pairwise disjoint, finite sets of variables, and, for each $i \leq n$,

- $D_i$ is a declaration mapping the variables in the set $X_i$ to formulae in HML with recursion that may use variables in the set $\bigcup_{1 \leq j \leq i} X_j$,
- $m_i = \max$ or $m_i = \min$, and
- $m_i \neq m_{i+1}$.

We refer to $(D_i, X_i, m_i)$ as the $i$th block of $E$ and say that it is a maximal block if $m_i = \max$ and a minimal block otherwise.

Observe that our earlier specification of the formula $Pos(LivelockNow)$ is given in terms of a 2-nested mutually recursive equational system. In fact, take $X_1 = \{LivelockNow\}$ and $X_2 = \{Pos(LivelockNow)\}$. You can now easily check that the constraints in the above definition are met. On the other hand, the mixed equational system

$$X \overset{\text{max}}{=} \langle a \rangle Y$$
$$Y \overset{\text{min}}{=} \langle b \rangle X$$

does not meet these requirements because the variables $X$ and $Y$ are both defined in mutually recursive fashion, and their definitions refer to different types of fixed points. If we allow fixed points to be mixed completely freely we obtain the modal $\mu$-calculus (Kozen, 1983a), which was mentioned in Section 6.1. In this book we shall however not allow a full freedom in mixing fixed points in declarations but restrict ourselves to systems of equations satisfying the constraints in Definition 6.2. Note that using the approach described above using our running example in this
section, such systems of equations have a unique solution, obtained by solving the first block and then recursively proceeding with the others using the solutions already obtained for the preceding blocks.

Finally if $F$ is a Hennessy-Milner formula defined over a set of variables $\mathcal{Y} = \{Y_1, \ldots, Y_k\}$ that are declared by an $n$-nested mutually recursive equational system $E$, then $[F]$ is well-defined and can be expressed by

$$[F] = O_F([Y_1], \ldots, [Y_k]),$$

(6.17)

where $[Y_1], \ldots, [Y_k]$ are the sets of states satisfying the recursively defined formulae associated with the variables $Y_1, \ldots, Y_k$.

**Exercise 6.17** Consider the labelled transition system in Exercise 6.16. Use equation (6.17) to compute the set of states satisfying the formula

$$F = \langle \text{Act} \rangle \text{Pos(LivelockNow)}.$$

**Exercise 6.18** Consider the following property expressed in natural language:

*It is always the case that each request is eventually followed by a grant.*

Express this property using HML with recursion. Next, construct a rooted labelled transition system that satisfies the property and one that does not. Check your constructions by computing the set of states in the labelled transition systems you have built that satisfy the formula.

### 6.8 Further results on model checking

We shall now present an overview of results connected to model checking for various modal and temporal logics over several classes of processes, as we have done for equivalence checking problems in Chapter 3.6.

We consider only the logics mentioned in the above text. They form the following natural expressiveness hierarchy:

- Hennessy-Milner logic (HML),
- Hennessy-Milner logic with one recursively defined variable (1HML), and
- the modal $\mu$-calculus (Kozen, 1983b), i.e., Hennessy-Milner logic with arbitrarily many nested and recursively defined variables.
These logics are typical representatives of the so called branching-time logics. The view of time taken by these logics is that each moment in time may branch into several distinct possible futures. Therefore, the structures used for interpreting branching-time logics can be viewed as computation trees. This means that in order to check for the validity of a formula, one has to consider a whole tree of states reachable from the root. Another typical and well-known branching-time logic is the computational tree logic or CTL (Clarke and Emerson, 1981), which uses (nested) until as the only temporal operator, the next-time modality X and existential/universal path quantifiers.

Another collection of temporal logics is that of the so called linear-time logics. The view of time taken by these logics is that each moment in time has a unique successor. Suitable models for formulae in such logics are therefore computation sequences. Here the validity of a formula is determined for a particular (fixed) trace of the system and possible branching is not taken into account. A process satisfies a linear-time formula if all of its computation sequences satisfy it. Linear temporal logic or LTL (Pnueli, 1977b) is probably the most studied logic of this type, in particular with the connection to the automata-theoretic approach (Vardi, 1995) and its implementation in tools like SPIN (Holzmann, 2003) and COSPAN (Har'El and Kurshan, 1987).

We shall first have a look at the decidability of model checking for the logics HML, 1HML and the modal μ-calculus over finite labelled transition systems. The model checking problem for the μ-calculus, which is the most expressive of those three logics, is decidable and it belongs both to the class NP and co-NP. In fact it was proved by Jurdziński (Jurdziński, 1998) that the problem is even in UP ∩ co-UP, which is the class of problems that can be decided by polynomial time nondeterministic Turing machines with the extra restriction that, for each input, there is at most one accepting computation of the Turing machine. It has been widely conjectured that the problem is indeed decidable in deterministic polynomial time. However, this is still one of the major open questions in this theory. The logics HML and 1HML are fragments of the μ-calculus. Their model checking problems are both decidable in polynomial (more precisely in linear) time on finite labelled transition systems (Cleaveland and Steffen, 1992). It is worth remarking here that the model checking problem for LTL over finite labelled transition systems is instead PSPACE-complete (Sistla and Clarke, 1985).

The aforementioned results on the complexity of model checking are based on the use of labelled transition systems as our model for reactive systems. However, in practice, most reactive systems contain several communicating components, and may be modelled as parallel compositions of (restricted classes of) labelled transition systems. As equivalence checking, model checking suffers from the so-called state explosion problem in the presence of concurrency. Hence a characterization
of the complexity of the model checking problem in the presence of concurrency yields a more realistic assessment of the hardness of the task of model checking reactive systems. The picture that emerges from the results presented in the literature on the complexity of model checking when the size of a concurrent process is measured in terms of the ‘length of its description’, rather than in the size of the labelled transition system that describes all of its possible computations, is somewhat bleak. The complexity of CTL model checking and of reachability for concurrent programs is PSPACE-complete (Kupferman, Vardi and Wolper, 2000; Kozen, 1977), and that of the (alternation-free) $\mu$-calculus is EXPTIME-complete (Kupferman et al., 2000).

If we consider the classes of sequential systems with infinitely many reachable states like, e.g., pushdown automata, the model checking problem for the $\mu$-calculus remains decidable. More precisely it is EXPTIME-complete (Walukiewicz, 2001).

In fact even more powerful logics like monadic second order logic—see, for instance, (Libkin, 2004, Chapter 7) for a textbook introduction—are still decidable over sequential infinite-state systems (Caucał, 1996; Muller and Schupp, 1985). The EXPTIME-hardness of model checking $\mu$-calculus formulae over pushdown automata is valid even in the case that the size of the formula is assumed to be constant (fixed). On the other hand, for fixed formulae and the BPA class (pushdown automata with a single control state), the problem is decidable in polynomial time (Burkart and Steffen, 1997; Walukiewicz, 2001). Model checking of HML is PSPACE-complete for BPA, but, for a fixed formula, this problem is again in P (Mayr, 1998).

The situation is, however, not that promising once we move from sequential infinite-state systems to parallel infinite-state systems. Both for the class of Petri nets (PN) and for its communication-free fragment BPP (CCS with parallel composition, recursion and action prefixing only) essentially all branching-time logics with at least one recursively defined variable are undecidable. More precisely, the EG logic which can express the property whether there exists a computation during which some HML formula is invariantly true is undecidable for BPP (Esparza and Kiehn, 1995) (and hence also for PN). The EF logic which can essentially express reachability properties is decidable for BPP (Esparza, 1997) but undecidable for PN (Esparza, 1994). On the other hand, the linear time logic LTL (with a certain restriction) is decidable for Petri nets and BPP (Esparza, 1994). This is an example when LTL turns out to be more tractable than branching-time logics. A thorough discussion of the relative merits of linear- and branching-time logics from a complexity theoretic perspective may be found in, e.g., the paper (Vardi, 2001).

For further references and more detailed overviews we refer the reader for example to (Burkart et al., 2001; Burkart and Esparza, 1997).
Chapter 7

Modelling and analysis of mutual exclusion algorithms

In the previous chapters of this book, we have illustrated the use of the ingredients in our methodology for the description and analysis of reactive systems by means of simple, but hopefully illustrative, examples. As we have mentioned repeatedly, the difficulty in understanding and reasoning reliably about even the simplest reactive systems has long been recognized. Apart from the intrinsic scientific and intellectual interest of a theory of reactive computation, this realization has served as a powerful motivation for the development of the theory we have presented so far, and of its associated verification techniques.

In order to offer you further evidence for the usefulness of the theory you have learned so far in the modelling and analysis of reactive systems, we shall now use it to model and analyze some well known mutual exclusion algorithms. These algorithms are amongst the most classic ones in the theory of concurrent algorithms, and have been investigated by many authors using a variety of techniques—see, for instance, the classic papers (Dijkstra, 1965; Knuth, 1966; Lamport, 1986). Here, they will give us the opportunity to introduce some modelling and verification techniques that have proven their worth in the analysis of many different kinds of reactive systems.

In order to illustrate concretely the steps that have to be taken in modelling and verification problems, we shall consider a very elegant solution to the mutual exclusion problem proposed by Peterson and discussed in (Peterson and Silberschatz, 1985).

In Peterson’s algorithm for mutual exclusion, there are two processes $P_1$ and $P_2$, two boolean variables $b_1$ and $b_2$ and an integer variable $k$ that may take the values 1 and 2. The boolean variables $b_1$ and $b_2$ have initial value false, whereas the
initial value of the variable $k$ can be arbitrary. In order to ensure mutual exclusion, each process $P_i$ ($i \in \{1, 2\}$) executes the following algorithm, where we use $j$ to denote the index of the other process.

```plaintext
while true do
begin
 'noncritical section';
 $b_j := \text{true}$;
 $k := j$;
 while ($b_j \text{ and } k = j$) do skip;
 'critical section';
 $b_i := \text{false}$;
end
```

As many concurrent algorithms in the literature, Peterson’s mutual exclusion algorithm is presented in pseudocode. Therefore one of our tasks, when modelling the above algorithm, is to translate the pseudocode description of the behaviour of the processes $P_1$ and $P_2$ into the model of labelled transition systems or into Milner’s CCS. Moreover, the algorithm uses variables that are manipulated by the processes $P_1$ and $P_2$. Variables are not part of CCS because, as discussed in Chapter 1.2, process calculi like CCS are based on the message passing paradigm, and not on shared variables. However, this is not a major problem. In fact, following the message passing paradigm, we can view variables as processes that are willing to communicate with other computing agents in their environment that need to read and/or write them.

By way of example, let us consider how to represent the boolean variable $b_1$ as a process. This variable will be encoded as a process with two states, namely $B_{1t}$ an $B_{1f}$. The former state will describe the ‘behaviour’ of the variable $b_1$ holding the value true, and the latter the ‘behaviour’ of the variable $b_1$ holding the value false. No matter what its value is, the variable $b_1$ can be read (yielding information on its value to the reading process) or written (possibly changing the value held by the variable). We need to describe these possibilities in CCS. To this end, we shall assume that processes read and write variables by communicating with them using suitable communication ports. For instance, a process wishing to read the value true from variable $b_1$ will try to synchronize with the process representing that variable on a specific communication channel, say $b1rt$—the acronym means ‘read the value true from $b_1$’. Similarly, a process wishing to write the value false into variable $b_1$ will try to synchronize with the process representing that variable on the communication channel $b1wf$—‘write false into $b_1$’.

Using these ideas, the behaviour of the process describing the variable $b_1$ can be represented by the following CCS expressions:
CHAPTER 7. MODELLING MUTUAL EXCLUSION ALGORITHMS

\[ B_{1f} \stackrel{\text{def}}{=} \text{blrf.B}_{1f} + \text{blwf.B}_{1f} + \text{b1wt.B}_{1t} \]
\[ B_{1t} \stackrel{\text{def}}{=} \text{blrt.B}_{1t} + \text{blwf.B}_{1f} + \text{b1wt.B}_{1t} . \]

Intuitively, when in state \( B_{1t} \), the above process is willing to tell its environment that its value is true, and to receive writing requests from other processes. The communication of the value of the variable to its environment does not change the state of the variable, whereas a writing request from a process in the environment may do so.

The behaviour of the process describing the variable \( b_2 \) can be represented in similar fashion thus:

\[ B_{2f} \stackrel{\text{def}}{=} \text{brf.B}_{2f} + \text{b2wf.B}_{2f} + \text{b2wt.B}_{2t} \]
\[ B_{2t} \stackrel{\text{def}}{=} \text{brt.B}_{2t} + \text{b2wf.B}_{2f} + \text{b2wt.B}_{2t} . \]

The CCS representation of the behaviour of the variable \( k \) is as follows:

\[ K_{1} \stackrel{\text{def}}{=} \text{k1.K}_{1} + \text{kw1.K}_{1} + \text{kw2.K}_{2} \]
\[ K_{2} \stackrel{\text{def}}{=} \text{k2.K}_{2} + \text{kw1.K}_{1} + \text{kw2.K}_{2} . \]

Again, the process representing the variable \( k \) has two states, denoted by the constants \( K_{1} \) and \( K_{2} \) above, because the variable \( k \) can only take the two values 1 and 2.

**Exercise 7.1** You should now be in a position to generalize the above examples. Assume that we have a variable \( v \) taking values over a data domain \( D \). Can you represent this variable using a CCS process? ♦

Having described the variables used in Peterson’s algorithm as processes, we are now left to represent the pseudocode algorithms for the processes \( P_1 \) and \( P_2 \) as CCS expressions. Note that, in doing so, we are making a step of formalization because pseudocode is a semi-formal notation without a precise syntax and semantics, whereas both the syntax and the semantics of CCS are unambiguously specified.

In our CCS formalization of the behaviour of processes \( P_1 \) and \( P_2 \), we shall ignore what the processes do outside and within their critical sections, and focus on their entering and exiting the critical section. After all, this is the interesting part of their behaviour as far as ensuring mutual exclusion is concerned! Moreover,
we shall assume, for the sake of simplicity, that processes cannot fail or terminate within the critical section. Under these assumptions, the initial behaviour of process $P_1$ can be described by the following CCS expression:

$$P_1 \overset{\text{def}}{=} \text{blw}.k2.P_{11}.$$ 

The above expression says that process $P_1$ begins by writing true in variable $b_1$ and 2 in variable $k$. Having done so, it will enter a new state that will be represented by the constant $P_{11}$. This new constant will intuitively describe the behaviour of process $P_1$ while it is executing the following line of pseudocode:

$$\text{while } (b_j \text{ and } k = j) \text{ do skip.}$$

To simulate this ‘busy waiting’ behaviour, we expect that process $P_{11}$ will

- read the value of the variables $b_j$ and $k$,
- loop back to $P_{11}$ if $b_j$ is true and $k$ is equal to 2, and
- move to a new state, say $P_{12}$, otherwise. In state $P_{12}$, we expect that process $P_1$ will enter and then exit the critical section.

The first thing to note here is that we need to make a decision as to the precise semantics of the informal pseudocode expression

$$b_j \text{ and } k = j.$$ 

How is this boolean conjunction evaluated? Is it evaluated from left to right, or from right to left? Assuming that it is evaluated from left to right, is the second conjunct evaluated if the first turns out to yield false? Different answers to these questions will produce different CCS processes. In what follows, we shall present a CCS description for process $P_{11}$ under the assumption that conjunctions are evaluated from left to right, and that the second conjunct is not evaluated if the value of the first is equal to false. Under these assumptions, we can write

$$P_{11} \overset{\text{def}}{=} \text{b2rf}.P_{12} + \text{b2rt}.(\text{kr2}.P_{11} + \text{kr1}.P_{12}) .$$

**Exercise 7.2** Would it have been a good idea to define $P_{11}$ thus:

$$P_{11} \overset{\text{def}}{=} \text{b2rf}.P_{12} + \text{b2rt}.\text{kr2}.P_{11} + \text{b2rt}.\text{kr1}.P_{12} ?$$

*Argue for your answer.*
To complete the description of the behaviour of the process $P_1$ we are left to present the defining equation for the constant $P_{12}$, describing the access to, and exit from, the critical section, and the setting of the variable $b_1$ to false:

$$P_{12} \overset{\text{def}}{=} \text{enter1.exit1.bTwf.P}_1 .$$

In the above CCS expression, we have labelled the enter and exit actions in a way that makes it clear that it is process $P_1$ that is entering and exiting the critical section.

The CCS process describing the behaviour of process $P_2$ in Peterson’s algorithm is entirely symmetric to the one we have just provided, and is defined thus:

$$P_2 \overset{\text{def}}{=} \text{b2wf.kw1.P}_{21} .$$

$$P_{21} \overset{\text{def}}{=} \text{b1rf.P}_{22} + \text{b1rt.(kr1.P}_{21} + \text{kr2.P}_{22}) .$$

$$P_{22} \overset{\text{def}}{=} \text{enter2.exit2.b2wf.P}_1 .$$

The CCS process term representing the whole of Peterson’s algorithm consists of the parallel composition of the terms describing the two processes running the algorithm, and of those describing the variables. Since we are only interested in the behaviour of the algorithm pertaining to the access to, and exit from, their critical sections, we shall restrict all of the communication channels that are used to read from, and write to, the variables. We shall use $L$ to stand for that set of channel names. Assuming that the initial value of the variable $k$ is 1, our CCS description of Peterson’s algorithm is therefore given by the term

$$\text{Peterson} \overset{\text{def}}{=} (P_1 \mid P_2 \mid B_{1f} \mid B_{2f} \mid K_1) \setminus L .$$

**Exercise 7.3 (Mandatory!)** Give a CCS process that describes the behaviour of Hyman’s ‘mutual exclusion’ algorithm. Hyman’s algorithm was proposed in (Hyman, 1966). It uses the same variables as Peterson’s.

In Hyman’s algorithm, each process $P_i$ ($i \in \{1, 2\}$) executes the algorithm in Figure 7.1, where as above we use $j$ to denote the index of the other process.

Now that we have a formal description of Peterson’s algorithm, we can set ourselves the goal to analyze its behaviour—manually or with the assistance of a software tool that can handle specifications of reactive systems given in the language CCS. In order to do so, however, we first need to specify precisely what it means for an algorithm to ‘ensure mutual exclusion’. In our formalization, it seems natural to identify ‘ensuring mutual exclusion’ with the following requirement:

At no point in the execution of the algorithm will both processes $P_1$ and $P_2$ be in their critical sections at the same time.
while true do
begin
    'noncritical section';
    \( b_i := \text{true}; \)
    while \( k \neq j \) do begin
        while \( b_j \) do skip;
        \( k := i \)
    end;
    'critical section';
    \( b_i := \text{false}; \)
end

Figure 7.1: The pseudocode for Hyman’s algorithm

How can we formalize this requirement? There are at least two options for doing so, depending on whether we wish to use HML with recursion or CCS processes/labelled transition systems as our specification formalism. In order to gain experience in the use of both approaches to specification and verification, in what follows we shall present specifications for mutual exclusion using HML with recursion and CCS.

7.1 Specifying mutual exclusion in HML

Hennessy-Milner logic with recursion is an excellent formalism for specifying our informal correctness condition for Peterson’s algorithm. To see this, observe, first of all, that the aforementioned desideratum is really a safety property in that it intuitively states that a desirable state of affairs—namely that ‘it is not possible for both processes to be in their critical sections at the same time’—is maintained throughout the execution of the process Peterson. We already saw in Chapter 6 that safety properties can be specified in HML with recursion using formulae of the form \( \text{Inv}(F) \), where \( F \) is the ‘desirable property’ that we wish to hold at all points in the execution of the process. Recall that \( \text{Inv}(F) \) is nothing but a shorthand for the recursively defined formula

\[
\text{Inv}(F) \overset{\text{max}}{=} F \land \text{[Act]}\text{Inv}(F) .
\]

So all that we are left to do in order to formalize our requirement for mutual exclusion is to give a formula \( F \) in HML describing the requirement that:
It is not possible for both processes to be in their critical sections at the same time.

In light of our CCS formalization of the processes $P_1$ and $P_2$, we know that process $P_i$ ($i \in \{1, 2\}$) is in its critical section precisely when it can perform action $\text{exit}_i$. So our formula $F$ can be taken to be

$$F \overset{\text{def}}{=} [\text{exit}_1] f f \lor [\text{exit}_2] f f.$$ 

The formula $\text{Inv}(F)$ now states that it is invariantly the case that either $P_1$ is not in the critical section or that $P_2$ is not in the critical section, which is an equivalent formulation of our correctness criterion.

Throughout this chapter, we are interpreting the modalities in HML over the transition system whose states are CCS processes, and whose transitions are weak transitions of the form $\alpha \Rightarrow$ for any action $\alpha$ including $\tau$. So a formula like $[\text{exit}_1] f f$ is satisfied by all processes that do not afford a $\text{exit}_1\Rightarrow$-labelled transition—that is, by those processes that cannot perform action $\text{exit}_1$ no matter how many internal steps they do before.

**Exercise 7.4** Consider the formula $\text{Inv}(G)$, where $G$ is

$$(\langle \text{enter}_1 \rangle [\text{enter}_2] f f) \land (\langle \text{enter}_2 \rangle [\text{enter}_1] f f).$$

Would such a formula be a good specification for our correctness criterion? What if we took $G$ to be the formula

$$(\langle \text{enter}_1 \rangle [\text{enter}_2] f f) \land (\langle \text{enter}_2 \rangle [\text{enter}_1] f f)?$$

*Argue for your answers!*

Now that we have a formal description of Peterson’s algorithm, and a specification of a correctness criterion for it, we could try to establish whether process Peterson satisfies the formula $\text{Inv}(F)$ or not.

With some painstaking effort, this could be done manually either by showing that the set of states of the process Peterson is a post-fixed point of the set function associated with the mapping

$$S \mapsto [F] \cap [\cdot \text{Act} \cdot]S,$$

or by iteratively computing the largest fixed point of the above mapping. The good news, however, is that we do not need to do so! One of benefits of having formal specifications of systems and of their correctness criteria is that, at least in
principle, they can be used as inputs for algorithms and tools that do the analysis for us.

One such verification tool for reactive systems that is often used for educational purposes is the so-called Edinburgh Concurrency Workbench (henceforth abbreviated to CWB) that is freely available at

\[ http://homepages.inf.ed.ac.uk/perdita/cwb/ \]

The CWB accepts inputs specified in CCS and HML with recursive definitions, and implements, amongst others, algorithms that check whether a CCS process satisfies a formula in HML with recursion or not. One of its commands (namely, `checkprop`) allows us to check, at the press of a button, that Peterson does indeed satisfy property \( \text{Inv}(F) \) above, and therefore that it preserves mutual exclusion, as its proposer intended.

**Exercise 7.5** Use the CWB to check whether Peterson satisfies the two candidate formulae \( \text{Inv}(G) \) in Exercise 7.4.

**Exercise 7.6 (Mandatory)** Use the CWB to check whether the CCS process for Hyman's algorithm that you gave in your answer to Exercise 7.3 satisfies the formula \( \text{Inv}(F) \) specifying mutual exclusion.

### 7.2 Specifying mutual exclusion using CCS itself

In the previous section, we have seen how to specify and verify the correctness of Peterson’s mutual exclusion algorithm using HML with recursion, and the model checking approach to the correctness problem. We have also hinted at the usefulness of an automatic verification tool like the CWB in the verification of even rather simple concurrent algorithms like Peterson’s algorithm. (Process Peterson has 69 states, and cannot be considered a ‘large reactive system’. However, its manual analysis already requires a fair amount of work and care.)

As mentioned previously in this book (see Chapter 3), implementation verification is another natural approach to the specification and verification of reactive systems. Recall that, in implementation verification, both actual systems and their specifications are represented as terms in the same model of concurrent computation—for instance as CCS terms or labelled transition systems. The correctness criterion in this setting is that, in some suitable formal sense, the term describing the implementation is equivalent to, or a suitable approximation of, that standing for the specification of the desired behaviour. As we have seen in Chapter 3, in this approach an important ingredient in the theory of reactive systems
7.2. SPECIFYING MUTUAL EXCLUSION USING CCS ITSELF

is therefore a notion of behavioural equivalence or approximation between process descriptions. Such a notion of equivalence can be used as our yardstick for correctness.

Unfortunately, however, there is no single notion of behavioural equivalence that fits all purposes. We have already met notions of equivalence like trace equivalence (Chapter 3.2), strong bisimilarity (Chapter 3.3) and observational equivalence (Chapter 3.4). Moreover, this is just the tip of the iceberg of ‘reasonable’ notions of equivalence or approximation between reactive systems. (The interested, and very keen, reader may wish to consult van Glabbeek’s encyclopaedic studies (Glabbeek, 1990; Glabbeek, 1993; Glabbeek, 2001) for an in-depth investigation of the notions of behavioural equivalence that have been proposed in the literature on concurrency theory.) So, when using implementation verification to establish the correctness of an implementation, such as our description of Peterson’s mutual exclusion algorithm, we need to

1. express our specification of the desired behaviour of the implementation using our model for reactive systems—in our setting as a CCS term, and

2. choose a suitable notion of behavioural equivalence to be used in checking that the model of the implementation is correct with respect to the chosen specification.

As you can see, in both of these steps we need to make creative choices—putting paid to the usual perception that verifying the correctness of computing systems is a purely mechanical endeavour.

So let us try and verify the correctness of Peterson’s algorithm for mutual exclusion using implementation verification. According to the above checklist, the first thing we need to do is to express the desired behaviour of a mutual exclusion algorithm using a CCS process term.

Intuitively, we expect that a mutual exclusion algorithm like Peterson’s initially allows both processes $P_1$ and $P_2$ to enter their critical sections. However, once one of the two processes, say $P_1$, has entered its critical section, the other can only enter after $P_1$ has exited its critical section. A suitable specification of the behaviour of a mutual exclusion algorithm seems therefore to be given by the CCS term

$$\text{MutexSpec} \overset{\text{def}}{=} \text{enter}_1.\text{exit}_1.\text{MutexSpec} + \text{enter}_2.\text{exit}_2.\text{MutexSpec}.$$  \hspace{1cm} (7.1)

Assuming that this is our specification of the expected behaviour of a mutual exclusion algorithm, our next task is to prove that the process Peterson is equivalent to, or a suitable approximation of, MutexSpec. What notion of equivalence or approximation should we use for this purpose?
You should be able to convince yourselves readily that strong bisimilarity or trace equivalence as presented in Chapter 3.2 will not do. (Why?) One possible approach would be to use observational equivalence (Definition 3.4) as our formal embodiment of the notion of correctness. Unfortunately, however, this would not work either! Indeed, you should be able to check easily that the process Peterson affords the weak transition

\[ \text{Peterson} \xrightarrow{\tau} (P_{12} \mid P_{21} \mid B_{11} \mid B_{21} \mid K_1) \setminus L, \]

and that the state that is the target of that transition affords an enter\(_1\)-labelled transition, but cannot perform a weak enter\(_2\)-labelled transition. On the other hand, the only state that process MutexSpec can reach by performing internal transitions is itself, and in that state both enter transitions are always enabled. It follows that Peterson and MutexSpec are not observationally equivalent.

**Exercise 7.7** What sequence of \(\tau\)-transitions will bring process Peterson into state \((P_{12} \mid P_{21} \mid B_{11} \mid B_{21} \mid K_1) \setminus L?\) (You’ll need five \(\tau\)-steps.)

Argue that, as we claimed above, that state affords an enter\(_1\)-labelled transition, but cannot perform a weak enter\(_2\)-labelled transition. ♦

This sounds like very bad news indeed. Observational equivalence allows us to abstract away from some of the internal steps in the evolution of process Peterson, but obviously not enough in this specific setting. We seem to need a more abstract notion of equivalence to establish the, seemingly obvious, correctness of Peterson’s algorithm with respect to our specification.

Observe that if we could show that the ‘observable content’ of each sequence of actions performed by process Peterson is a trace of process MutexSpec, then we could certainly conclude that Peterson does ensure mutual exclusion. In fact, this would mean that at no point in its behaviour process Peterson can perform two exit actions in a row—possibly with some internal steps in between them. But what do we mean precisely by the ‘observable content’ of a sequence of actions? The following definition formalizes this notion in a very natural way.

**Definition 7.1** [Weak Traces and Weak Trace Equivalence] A weak trace of a process \(P\) is a sequence \(a_1 \cdots a_k (k \geq 1)\) of observable actions such that there exists a sequence of transitions

\[ P = P_0 \xrightarrow{a_1} P_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} P_k, \]

for some \(P_1, \ldots, P_k\). Moreover, each process affords the weak trace \(\varepsilon\).

We say that a process \(P\) is a weak trace approximation of process \(Q\) if the set of weak traces of \(P\) is included in that of \(Q\). Two processes are weak trace equivalent if they afford the same weak traces. ♦
Note that the collection of weak traces coincides with that of traces for processes that, like MutexSpec, do not afford internal transitions. (Why?)

We claim that the processes Peterson and MutexSpec are weak trace equivalent, and therefore that Peterson does meet our specification of mutual exclusion modulo weak trace equivalence. This can be checked automatically using the command `mayeq` provided by the CWB. (Do so!) This equivalence tells us that not only each weak trace of process Peterson is allowed by the specification MutexSpec, but also that process Peterson can exhibit as a weak trace each of the traces permitted by the specification.

If we are just satisfied with checking the pure safety condition that no trace of process Peterson violates the mutual exclusion property, then it suffices only to show that Peterson is a weak trace approximation of MutexSpec. A useful proof technique that can be used to establish this result is given by the notion of weak simulation. (Compare with the notion of simulation defined in Exercise 3.17.)

Definition 7.2 [Weak Simulation] Let us say that a binary relation $R$ over the set of states of an LTS is a weak simulation iff whenever $s_1 \Rightarrow s_2$ and $\alpha$ is an action:

- if $s_1 \xrightarrow{\alpha} s'_1$, then there is a transition $s_2 \Rightarrow s'_2$ such that $s'_1 \Rightarrow s'_2$.

We say that $s'$ weakly simulates $s$ iff there is a weak simulation $R$ with $s \Rightarrow s'$. ♦

Proposition 7.1 For all states $s, s', s''$ in a labelled transition system, the following statements hold.

1. State $s$ weakly simulates itself.
2. If $s'$ weakly simulates $s$, and $s''$ weakly simulates $s'$, then $s''$ weakly simulates $s$.
3. If $s'$ weakly simulates $s$, then each weak trace of $s$ is also a weak trace of $s'$.

In light of the above proposition, to show that Peterson is a weak trace approximation of MutexSpec, it suffices only to build a weak simulation that relates Peterson with MutexSpec. The existence of such a weak simulation can be checked using the command `pre` offered by the CWB. (Do so!)

Exercise 7.8 Prove Proposition 7.1.

Exercise 7.9 Assume that $s'$ weakly simulates $s$, and $s$ weakly simulates $s'$. Is it true that $s$ and $s'$ are observationally equivalent? Argue for your answer.

Exercise 7.10 Assume that the CCS process $Q$ weakly simulates $P$. Show that $Q + R$ weakly simulates $P$ and $P + R$, for each CCS process $R$. ♦
Exercise 7.11

1. Show that the processes \( \alpha.P + \alpha.Q \) and \( \alpha.(P + Q) \) are weak trace equivalent for each action \( \alpha \), and terms \( P, Q \).

2. Show that weak trace equivalence is preserved by all of the operators of CCS.

7.3 Testing mutual exclusion

Another approach to establishing the correctness of Peterson’s algorithm is to use a notion of ‘testing’. Recall that what we mean by ensuring mutual exclusion is that at no point in the execution of process Peterson both processes will be in their critical section at the same time. Such a situation would arise if there is some execution of process Peterson in which two enter actions occur one after the other without any exit action in between them. For instance, process \( P_1 \) might perform action enter1, and the next observable action might be enter2—causing both processes to be in their critical section at the same time. A way to check whether this undesirable situation can ever occur in the behaviour of process Peterson is to make it interact with a ‘monitor process’ that observes the behaviour of process Peterson, and reports an error if and when the undesirable situation arises. This is a conceptually simple, but very useful, technique that has arisen in various forms over and over again in the study of verification techniques for reactive systems, and probably finds its most theoretically satisfying embodiment in the classic automata-theoretic approach to verification—see, for instance, the references (Vardi, 1991; Vardi and Wolper, 1994).

So, how can we construct a monitor process that reports a failure in ensuring mutual exclusion if any arises? Intuitively, such a process would observe the enter and exit actions performed by process Peterson. Whenever an enter action is observed, the monitor process reaches a state in which it is ready to report that something bad has happened if it observes that the other process can now enter its critical section as well. If our monitor process observes the relevant exit action as expected, it gladly returns to its initial state, ready to observe the next round of the execution of the algorithm. A CCS process term describing the above behaviour is, for instance,

\[
\begin{align*}
\text{MutexTest} & \stackrel{\text{def}}{=} \text{enter}_1.\text{MutexTest}_1 + \text{enter}_2.\text{MutexTest}_2 \\
\text{MutexTest}_1 & \stackrel{\text{def}}{=} \text{exit}_1.\text{MutexTest} + \text{enter}_2.\text{bad}.0 \\
\text{MutexTest}_2 & \stackrel{\text{def}}{=} \text{exit}_2.\text{MutexTest} + \text{enter}_1.\text{bad}.0 ,
\end{align*}
\]
7.3. TESTING MUTUAL EXCLUSION

where we have assumed that our monitor process outputs on channel name bad, 
when it discovers that two enter actions have occurred without an intervening exit.

In order to check whether process Peterson ensures mutual exclusion, it is now 
sufficient to let it interact with MutexTest, and ask whether the resulting system 
(Peterson | MutexTest) \ {enter₁, enter₂, exit₁, exit₂}
can initially perform the action bad. Indeed, we have the following result:

**Proposition 7.2** Let \( P \) be a CCS process whose only visible actions are contained 
in the set \( L' = \{\text{enter}_1, \text{enter}_2, \text{exit}_1, \text{exit}_2\} \). Then \((P | \text{MutexTest}) \setminus L' \overset{\text{bad}}{\Rightarrow} \) iff 
either \( P \overset{\sigma}{\Rightarrow} P' \overset{\text{enter}_1}{\Rightarrow} P'' \overset{\text{enter}_2}{\Rightarrow} \) or \( P \overset{\sigma}{\Rightarrow} P' \overset{\text{enter}_2}{\Rightarrow} P'' \overset{\text{enter}_1}{\Rightarrow} \), for some \( P', P'' \) and 
sequence of actions \( \sigma \) in the regular language \((\text{enter}_1\text{exit}_1 + \text{enter}_2\text{exit}_2)^*\).

**Proof:** For the ‘if implication’, assume, without loss of generality, that 
\( P \overset{\sigma}{\Rightarrow} P' \overset{\text{enter}_1}{\Rightarrow} P'' \overset{\text{enter}_2}{\Rightarrow} P''' \),
for some \( P', P'', P''' \) and sequence of actions \( \sigma \in (\text{enter}_1\text{exit}_1 + \text{enter}_2\text{exit}_2)^* \). We shall argue that \((P | \text{MutexTest}) \setminus L' \overset{bad}{\Rightarrow} \). To see this, note that, using induction on 
the length of the sequence \( \sigma \), it is not hard to prove that 
\((P | \text{MutexTest}) \setminus L' \overset{\tau}{\Rightarrow} (P' | \text{MutexTest}) \setminus L' \).

Since \( P' \overset{\text{enter}_1}{\Rightarrow} P'' \overset{\text{enter}_2}{\Rightarrow} P''' \), we have that 
\((P' | \text{MutexTest}) \setminus L' \overset{\tau}{\Rightarrow} (P'' | \text{MutexTest}) \setminus L' \overset{\tau}{\Rightarrow} (P''' | \text{bad} \cdot 0) \setminus L' \overset{\text{bad}}{\Rightarrow} \).

Combining the above sequences of transitions, we may conclude that 
\((P | \text{MutexTest}) \setminus L' \overset{\text{bad}}{\Rightarrow} \),
which was to be shown.

Conversely, assume that \((P | \text{MutexTest}) \setminus L' \overset{\text{bad}}{\Rightarrow} \). Since \text{bad} \cdot 0 is the only state 
of process MutexTest that can perform a bad-action, this means that, for some \( P''' \), 
\((P | \text{MutexTest}) \setminus L' \overset{\tau}{\Rightarrow} (P''' | \text{bad} \cdot 0) \setminus L' \overset{\text{bad}}{\Rightarrow} \).

Because of the way MutexTest is constructed, this must be because, for some \( P' \) 
and \( P'' \) such that either \( P' \overset{\text{enter}_1}{\Rightarrow} P'' \overset{\text{enter}_2}{\Rightarrow} P''' \) or \( P' \overset{\text{enter}_2}{\Rightarrow} P'' \overset{\text{enter}_1}{\Rightarrow} P''' \),
\((P | \text{MutexTest}) \setminus L' \overset{\tau}{\Rightarrow} (P' | \text{MutexTest}) \setminus L' \overset{\tau}{\Rightarrow} (P'' | \text{MutexTest}) \setminus L' \overset{\tau}{\Rightarrow} (P''' | \text{bad} \cdot 0) \setminus L' \).
Using induction on the length of the transition

\[(P \mid \text{MutexTest}) \setminus L' \xrightarrow{\tau} (P' \mid \text{MutexTest}) \setminus L'\]

you can now argue that \(P \xrightarrow{\sigma} P'\), for some sequence of actions \(\sigma\) in the regular language \((\text{enter}_1\text{exit}_1 + \text{enter}_2\text{exit}_2)^*\). This completes the proof. \(\square\)

**Exercise 7.12** Fill in the details in the above proof.

**Aside: testable formulae in Hennessy-Milner logic**

This section is for the theoretically minded readers, who would like a glimpse of some technical results related to testing formulae in HML with recursion, and is meant as a pointer for further self-study.

Those amongst you that solved Exercise 7.4 might have already realized that, intuitively, the monitor process MutexTest is ‘testing’ whether the process it observes satisfies the formula \(\text{Inv}(G)\), where \(G\) is

\[([\text{enter}_1][\text{enter}_2]ff) \land ([\text{enter}_2][\text{enter}_1]ff)\]

A natural question to ask is whether each formula in the language HML with recursion can be tested as we just did with the above formula \(\text{Inv}(G)\). In order to make this question precise, we need to define the collection of allowed tests and the notion of property testing. Informally, testing involves the parallel composition of the tested process (described by a state in a labelled transition system or by a CCS process) with a test. Following the spirit of the classic approach of De Nicola and Hennessy (De Nicola and Hennessy, 1984; Hennessy, 1988), and our developments above, we say that the tested state fails a test if the distinguished reject action \(\text{bad}\) can be performed by the test while it interacts with it, and passes otherwise. The formal definition of testing then involves the definition of what a test is, how interaction takes place and when the test has failed or succeeded. We now proceed to make these notions precise.

**Definition 7.3** [Tests] A test is a finite, rooted LTS \(T\) over the set of actions \(\text{Act} \cup \{\text{bad}\}\), where \(\text{bad}\) is a distinguished channel name not occurring in \(\text{Act}\). We use \(\text{root}(T)\) to denote the start state of the LTS \(T\).

As above, the idea is that a test acts as a monitor that ‘observes’ the behaviour of a process and reports any occurrence of an undesirable situation by performing a \(\text{bad}\)-labelled transition.
7.3. TESTING MUTUAL EXCLUSION

In the remainder of this section, tests will often be concisely described using the regular fragment of Milner’s CCS—that is the fragment of CCS given by the following grammar:

\[ T ::= 0 \mid \alpha.T \mid T + T \mid X, \]

where \( \alpha \) can be any action in \( \text{Act} \) as well as the distinguished action \( \text{bad} \), and \( X \) is a constant drawn from a given, finite set of process names. The right-hand side of the defining equations for a constant can only be a term generated by the above grammar. For example, the process MutexTest we specified above is a regular CCS process, but the term

\[ X \overset{\text{def}}{=} a.(b.0 \mid X) \]

is not.

We now proceed to describe formally how tests can be used to check whether a process satisfies a formula expressed in HML with recursion.

**Definition 7.4** [Testing Properties] Let \( F \) be a formula in HML with recursion, and let \( T \) be a test.

- For every state \( s \) of an LTS, we say that \( s \) passes the test \( T \) iff

\[ (s \mid \text{root}(T)) \not\rightarrow \text{bad} \]

(Recall that \( \mathcal{L} \) stands for the collection of observable actions in CCS.) Otherwise we say that \( s \) fails the test \( T \).

- We say that the test \( T \) tests for the formula \( F \) (and that \( F \) is testable) iff for every LTS \( T \) and every state \( s \) of \( T \),

\[ s \models F \iff s \text{ passes the test } T. \]

- A collection of formulae in HML with recursion is testable iff each of the formulae in it is.

\[ \blacklozenge \]

**Example 7.1** The formula \( [a] ff \) is satisfied by those processes that do not afford an \( \overset{a}{\Rightarrow} \)-transition. We therefore expect that a suitable test for such a property is

\[ T \equiv \overline{a}.\text{bad}.0. \]

Indeed, the reader will easily realize that \( (s \mid T) \not\rightarrow \text{bad} \) iff \( s \not\Rightarrow \), for every state \( s \). The formula \( [a] ff \) is thus testable, in the sense of Definition 7.4.

The formula defined by the recursion equation

\[ F \overset{\text{max}}{=} [a] ff \land [b] F \]
is satisfied by those states which cannot perform a $a \Rightarrow$-transition, no matter how they engage in a sequence of $b \Rightarrow$-transitions. (Why?) A suitable test for such a property is
\[ X \overset{\text{def}}{=} \overline{a.bad.0} + \overline{b}.X , \]
and the recursively defined formula $F$ is thus testable.

\textbf{Exercise 7.13} Consider the following labelled transition system:

\begin{center}
\begin{tikzpicture}
    \node (p) at (0,0) {$p$};
    \node (q) at (1,0) {$q$};
    \node (r) at (2,0) {$r$};
    \node (s) at (1,-1) {$s$};
    \node (a) at (1,-2) {$a$};
    \node (b) at (2,-2) {$b$};
    \draw (p) -- node[above] {$b$} (q);
    \draw (q) -- node[above] {$b$} (r);
    \draw (s) -- node[below] {$a$} (p);
    \draw (s) -- node[below] {$b$} (r);
    \end{tikzpicture}
\end{center}

Compute the set of states in this labelled transition system that satisfy the property
\[ F \overset{\text{max}}{=} [a].ff \land [b].F . \]
Which of the states in that labelled transition system passes the test
\[ X \overset{\text{def}}{=} \overline{a.bad.0} + \overline{b}.X ? \]
Argue for your answers!

\textbf{Exercise 7.14} Prove the claims that we have made in the above example.

In Example 7.1, we have met two examples of testable formulae. But, can each formula in HML with recursion be tested in the sense introduced above? The following instructive result shows that even some very simple HML properties are not testable in the sense of Definition 7.4.

\textbf{Proposition 7.3} [Two Negative Results]

1. For every action $a$ in $\mathcal{L}$, the formula $\langle a \rangle \# \#$ is not testable.
2. Let $a$ and $b$ be two distinct actions in $\mathcal{L}$. Then the formula $[a].ff \lor [b].ff$ is not testable.

\textbf{Proof:} We prove each statement in turn.
7.3. TESTING MUTUAL EXCLUSION

- **Proof of (1).** Assume, towards a contradiction, that the test $T$ tests for the formula $\langle a \rangle \top$. Since $T$ tests for $\langle a \rangle \top$ and $0 \not\models \langle a \rangle \top$, we have that

$$ (0 \mid \text{root}(T)) \not\models \text{bad} \Rightarrow. $$

Consider now the term $P = a.0 + \tau.0$. As $P \not\models a$, the process $P$ satisfies the formula $\langle a \rangle \top$. However, $P$ fails the test $T$ because

$$ (P \mid \text{root}(T)) \not\models (0 \mid \text{root}(T)) \not\models \text{bad} \Rightarrow. $$

This contradicts our assumption that $T$ tests for $\langle a \rangle \top$.

- **Proof of (2).** Assume, towards a contradiction, that the test $T$ tests for the formula $[a].\mathbf{f} \lor [b].\mathbf{f}$, with $a \neq b$. Since the state $a.0 + b.0$ does not satisfy the formula $[a].\mathbf{f} \lor [b].\mathbf{f}$, it follows that

$$ ((a.0 + b.0) \mid \text{root}(T)) \not\models \text{bad} \Rightarrow. \quad (7.2) $$

We now proceed to show that this implies that either the state $a.0$ fails the test $T$ or $b.0$ does. This we do by examining the possible forms transition (7.2) may take.

- **Case:** $((a.0 + b.0) \mid \text{root}(T)) \not\models \text{bad} \Rightarrow$ because $\text{root}(T) \not\models \text{bad} \Rightarrow$. In this case, every state of an LTS fails the test $T$, and we are done.

- **Case:** $((a.0 + b.0) \mid \text{root}(T)) \not\models (0 \mid t) \not\models \text{bad} \Rightarrow$, because $\text{root}(T) \not\models \text{bad} \Rightarrow$. In this case, we may infer that

$$ (a.0 \mid \text{root}(T)) \not\models (0 \mid t) \not\models \text{bad} \Rightarrow $$

and thus that $a.0$ fails the test $T$.

- **Case:** $((a.0 + b.0) \mid \text{root}(T)) \not\models (0 \mid t) \not\models \text{bad} \Rightarrow$, because $\text{root}(T) \not\models \text{bad} \Rightarrow$. In this case, reasoning as above, it is easy to see that $b.0$ fails the test $T$.

Hence, as previously claimed, either $a.0$ fails the test $T$ or $b.0$ does. Since both $a.0$ and $b.0$ satisfy the formula $[a].\mathbf{f} \lor [b].\mathbf{f}$, this contradicts our assumption that $T$ tests for it.

The proof is now complete. $\square$
The collection of formulae in safety HML is the set of formulae in HML with recursion that do not contain occurrences of $\lor$, $\langle \alpha \rangle$ and variables defined using least fixed point recursion equations.

**Exercise 7.15 (Strongly Recommended)** *Can you build a test (denoted by a process in the regular fragment of CCS) for each formula in safety HML without recursion? Hint: Use induction on the structure of formulae.*

It turns out that, with the addition of recursive formulae defined using largest fixed points, the collection of testable formulae in HML with recursion is precisely the one you built tests for in the previous exercise! This is the import of the following result from (Aceto and Ingólfsdóttir, 1999).

**Theorem 7.1** The collection of formulae in safety HML is testable. Moreover, every testable property in HML with recursion can be expressed in safety HML.

Thus we can construct tests for safety properties expressible in HML with recursion. We refer the interested readers to (Aceto and Ingólfsdóttir, 1999) for more details, further developments and references to the literature.
Part II

A Theory of Real-time Systems
Chapter 8

Introduction

In the first part of this book, we have motivated and developed a general purpose theory that can be used to describe, and reason about, reactive systems. The key ingredients in our approach were

- an algebraic language, namely Milner’s CCS, for the syntactic description of reactive systems,
- automata/labelled transition systems for describing the dynamic behaviour of process terms,
- Structural Operational Semantics, allowing us to associate systematically a labelled transition system with each process term in a syntax directed fashion,
- notions of behavioral equivalence to compare process behaviours, and
- modal and temporal logics to specify desired properties of reactive systems.

These ingredients gave the foundations for the formal modelling and verification of reactive systems, and are the bedrock for the development of (semi-)automatic verification tools for reactive systems.

The theory that we have developed so far, however, does not allow us to describe naturally all of the important aspects in reactive computation. Consider, for instance, some by now ubiquitous examples of reactive systems, namely embedded systems like the ABS and air bags in cars, cruise control systems, digital watches, mobile phones, the monitors of your computers, production lines and video game consoles. These are all examples of real-time systems. A real-time system is a system whose correct behaviour does not only depend on the logical order in which
events are performed, but also on their timing. Think for a moment about the expected behaviour of an air bag system in a car. Such a system is intended to inflate the air bags in case of a car crush, but this behaviour is not just expected to occur “eventually”. Rather, we should like to have some (hopefully small) hard bounds on the timing of its occurrence. A suitable correctness criterion for such a system might therefore be that

If the car crushes, the airbag must be inflated within 50 milliseconds,
say, rather than a not so reassuring “the air bag will be eventually inflated”.
Another instructive, and suggestive, example of a real-time system is that of a control program that we already met when introducing the general notion of reactive system. Recall that, at a high level of abstraction, the behaviour of a control program can be seen to be governed by the following pseudo-code algorithm skeleton.

```plaintext
loop
    read the sensors' values at regular intervals
    depending on the sensors' values trigger the relevant actuators
forever
```

In the above description, we have some implicit and qualitative description of the real-time behaviour of a control program. Such a program can be thought of as being in an “idle mode” in between consecutive readings of the values of the sensors. When the “idle interval” is over, the system polls the values of the sensors, triggers the relevant actions interacting with its environment, and then re-enters its “idle mode”. Such a system is a typical example of an hybrid system—that is, a discrete system that interacts with a continuously evolving one, namely its environment. (In the jargon of control theory, the environment is usually referred to as the plant.) As we shall see in what follows, the expected behaviour of the aforementioned control program will serve as a useful example to explain some of the choices that researchers have made in designing appropriate models for the description of reactive, real-time computation.
9.1 Intuition

All of the types of reactive systems that we have mentioned above should give us a sufficient motivation to describe and analyze formally real-time reactive computations. In the first part of the book, we have introduced a collection of languages and models based on the flexible and intuitive idea of communicating state machines, and argued, by means of several examples, that the resulting formalisms can be used to describe and analyze non-trivial reactive systems. When real-time constraints become important to the proper functioning of reactive systems, we should like to continue building on the time-honoured formalisms we have introduced previously. But, are those formalisms sufficiently powerful to describe timing constraints in computation? Can we use them to specify, for instance, features like time-outs?

Consider, by way of example, a light switch that has the following behaviour:

If the switch is off, and is pressed once, then the light will turn on. If the switch is then pressed again “soon after” the light was turned on, then the light becomes brighter. Otherwise, the light is turned off by the next button press.

A way of describing this behaviour using CCS is to construct a process with three states—say, Off, Light and Bright—describing the three possible states mentioned in the above English description of the behaviour of the system. Modelling the behaviour of the switch in the Off or Bright states is easy:

\[
\begin{align*}
\text{Off} & \overset{\text{def}}{=} \text{press.Light} \quad \text{and} \\
\text{Bright} & \overset{\text{def}}{=} \text{press.Off}
\end{align*}
\]
How can we describe the behaviour of the system in the Light state? One approach would be to write the following CCS term:

\[
\text{Light} \overset{\text{def}}{=} \tau.\text{press.Off} + \text{press.Bright},
\]

which describes the possible effects that pressing the button can have when the system is in the Light state. Note, however, that the above description does not capture the requirement that if the user presses the button “quickly” after the light is on, then the light will become brighter. (Rather, it intuitively states that the system may internally choose to switch off the light at the next button press.) This is a timing requirement on consecutive button presses, and CCS offers no facility for describing it. What kind of constructs can we add to CCS in order to describe systems like the light switch above, whose behaviour is time dependent?

One of the most important principles underlying the development of CCS, and of all of the models and languages we have met so far, is that of parsimony. These models are built on a small collection of operators that are sufficient to describe the computational phenomena under study. Using this principle, we should like to extend a language like CCS, and the model of labelled transition systems, with the least amount of machinery that allows us to describe time dependent behaviour. As we shall see, the lessons we learned from the behaviour in time of the skeleton control program will help us a lot in deciding how to extend CCS and labelled transition systems with timing.

Intuitively, we have argued that one can view a real-time system like the aforementioned skeleton of a control program as having a two-phase behaviour. In fact, that system alternates between phases in which the system is idle—that is, it does not perform any action—and time passes, and those in which the system performs sequences of actions triggering the relevant actuators and possibly chooses between different courses of action before returning to its idle mode. As in CCS, it is a useful abstraction to consider the sequence of actions performed by the system as being instantaneous in time.

We already know how to describe syntactically action occurrences by means of the action prefixing operator of CCS. What is the minimum amount of machinery that we can add to our language in order to give a faithful description of the passage of time? A possible, conservative answer is to view the passage of time as being some kind of “action” that a system may perform. This action can be specified by means of a new prefixing operator that describes time delays—say $\epsilon(d)$, where $d$ is a real number that specifies the amount of time that needs to elapse before the “idling time” is over. For example, using this new type of prefixing operator, the behaviour of the light switch in state Light could be described by means of the
equation
\[ \text{Light} \overset{\text{def}}{=} \varepsilon(1.4).\text{press.Off} + \text{press.Bright}, \] (9.1)
assuming that pressing the button “quickly” means doing so within 1.4 seconds, say. The reason why we included here also the action \( \tau \) might not be completely obvious at this moment and will be explained later on. Intuitively, the reader should think of \( \tau \) as an action that cannot be delayed and must be performed as soon as it becomes available. This introduces the notion of urgency, which is useful for describing features like time-outs.

The term on the right-hand side of the above equation, however, needs to be given a semantics in order for its behaviour to be understood. In particular, this involves giving the formal semantics of the delay operator and describing how it interacts with the other constructs of the language. Moreover, we need to choose a suitable semantic model that can be used to describe formally the behaviour of terms in a timed version of CCS. Last but not least, a choice has to be made as to the structure to be used to model Time. In what follows, we shall use the set \( \mathbb{R}_{\geq 0} \) of non-negative real numbers as our time domain. This appears as a natural choice, if we think of the flow of time as being continuous. However, some researchers prefer to work with a discrete notion of time, and this can be modelled by using the set of natural numbers \( \mathbb{N} \) as time domain.

9.2 Timed Labelled Transition Systems

In the first part of this book, we have used labelled transition systems to model the behaviour of reactive systems. Since this model of computation is very intuitive and flexible, we should like to use a variation on it in order to give semantics to real-time reactive systems. In light of the above discussion, it is natural to assume that we can describe the passage of time by adding special “delay” transitions to the model. Such transitions could, for instance, be used to give the formal semantics of the delay prefixing operators \( \varepsilon(d) \), with \( d \in \mathbb{R}_{\geq 0} \), used in equation (9.1). The resulting structure is a timed version of the model of LTSs. This we now proceed to describe formally.

**Definition 9.1** A *timed (labelled) transition system* (TLTS) is a triple
\[
(\text{Proc}, \text{Lab}, \{ a \overset{\alpha}{\rightarrow} \mid a \in \text{Lab}\}),
\]
where
- \( \text{Proc} \) is a set of states (or processes),
• $\text{Lab} = \text{Act} \cup \mathbb{R}_{\geq 0}$ is a set of labels (consisting of actions and time delays), and

• $\rightarrow \subseteq \text{Proc} \times \text{Proc}$, for each $a \in \text{Lab}$, is a binary relation on states called the transition relation.

As usual, we write

• $s \xrightarrow{a} s'$ if $a \in \text{Act}$ and $(s, s') \in \rightarrow$, and

• $s \xrightarrow{d} s'$ if $d \in \mathbb{R}_{\geq 0}$ and $(s, s') \in \rightarrow$.

Transitions of the type $s \xrightarrow{a} s'$ are ordinary transitions that are due to the performance of actions, and those of the form $s \xrightarrow{d} s'$, with $d \in \mathbb{R}_{\geq 0}$, are time-elapsing transitions describing how a system evolves as time passes. A little reflection, however, leads us to conclude that not all structures of the above kind reflect our intuition as to the passage of time. Assume, for instance, that a state $s$ in a timed LTS affords a transition of the form $s \xrightarrow{\text{1.4}} s'$. (Such a state could, for instance, describe the timing behaviour of the process Light in equation (9.1).) That transition tells us that a system in state $s$ can wait for 1.4 units of time, and thereby evolve into $s'$. However, if a system can wait for 1.4 seconds, say, then it is natural to expect that it can delay for, say, 0.8 seconds and thereby reach a state that can then proceed to wait for 0.6 seconds, and become $s'$ in doing so. This means that time is additive, and that the transition relation of a TLTS ought to satisfy the following time additivity requirement:

$$\text{if } s \xrightarrow{d} s' \text{ and } 0 \leq d' \leq d \text{ then } s \xrightarrow{d'} s'' \xrightarrow{d-d'} s', \text{ for some state } s''.$$  \hspace{1cm} (9.2)

Note that $d'$ could be 0. Since it is reasonable to assume that a state can only reach itself without delay, we also postulate that

$$s \xrightarrow{0} s, \text{ for each state } s.$$  \hspace{1cm} (9.3)

A final requirement that is imposed on TLTSs captures the way a system can evolve just by idling as time passes. In order to motivate this requirement, consider the control program we have already discussed. When in idle mode, that program is intuitively just updating the value of some timer that measures the amount of time that needs to elapse before the system polls the values of its sensors next. Each time delay therefore brings the system to a unique next state, in which intuitively the value of the timer has decreased by the amount of delay. This example indicates
that delay transitions are deterministic—that is, for all \( s, s', s'' \) and for each \( d \in \mathbb{R}_{\geq 0} \),

\[
\text{if } s \xrightarrow{d} s' \text{ and } s \xrightarrow{d} s'' \text{ then } s' = s''.
\]

(9.4)

In what follows, we shall restrict our attention to TLTSs that satisfy requirements (9.2)–(9.4).

**Example 9.1** Consider the following timed transition system

\[
(\text{Proc}, \text{Lab}, \{ \xrightarrow{a} \mid a \in \text{Lab} \})
\]

where \( \text{Proc} = \mathbb{R}_{\geq 0} \), \( \text{Lab} = \{ a \} \cup \mathbb{R}_{\geq 0} \) such that \( \xrightarrow{a} = \{(5, 0)\} \) and for all \( d \in \mathbb{R}_{\geq 0} \) we define \( \xrightarrow{d} = \{(d', d'') \in \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \mid d' + d = d''\} \). The picture shows a fragment of the timed transition system defined above. Note that we have included only very few timed transitions, in fact there are infinitely (or rather uncountably) many transitions that should be added to the picture.

It is easy to verify that the transition system above satisfies conditions (9.2), (9.3) and (9.4). The only ordinary action is called \( a \) and whenever the system is at time point 5, it can move the control to the initial state.

### 9.3 Syntax and SOS rules

In the remainder of this section, we shall specify TLTSs using a timed version of the language CCS that is essentially Wang Yi’s Timed CCS (henceforth abbreviated to TCCS) (Wang, 1990; Wang, 1991a; Wang, 1991b). As indicated in our discussion in Section 9.1, the only constructs that we shall add to the syntax of the language CCS presented in Definition 2.3 are the delay prefixing operators \( \varepsilon(d) \), with \( d \in \mathbb{R}_{\geq 0} \).

Formally, the collection \( \mathcal{P} \) of Timed CCS expressions is given by the grammar for CCS expressions given in Definition 2.3 extended with the following formation rule

\[
\text{if } P \text{ is in } \mathcal{P} \text{ and } d \in \mathbb{R}_{\geq 0}, \text{ then } \varepsilon(d).P \text{ is in } \mathcal{P}.
\]
In what follows, we shall not distinguish the terms $P$ and $\varepsilon(0).P$.

As for standard CCS, we assume that the behaviour of each process constant is given by a defining equation

$$K \overset{\text{def}}{=} P.$$  

**Definition 9.2** An occurrence of a constant $K$ in an expression $P$ is *guarded* if it occurs within a sub-expression of $P$ of the form $\alpha.Q$. ♦

In what follows, we shall restrict ourselves to considering only processes involving constants whose defining equations contain only guarded occurrences of constants. These expressions are called *guarded*.

**Example 9.2** Consider the following timed CCS expression.

$$(a.K_1 + (K_2 \mid b.K_3) + K_1) \mid (\varepsilon(4.2).(K_4 \mid 0) + \varepsilon(1.2).K_3)$$

The first occurrence (from the left) of the constant $K_1$ is guarded, while its second occurrence is not guarded. The single occurrence of $K_2$ is not guarded either. Both occurrences of $K_3$ are guarded, as well as the single occurrence of $K_4$. All together, the whole expression is not guarded.

**Exercise 9.1** Convince yourselves that the specification of the process Light given in (9.1) is a guarded Timed CCS expression. ♦

By analogy with standard CCS, we expect that the behaviour of the process Light given in (9.1) is determined by that of the right-hand side of its defining equation, namely the expression

$$\varepsilon(1.4).\tau.\text{press.Off} + \text{press.Bright}.$$  

The action transitions of this expression are determined by the same SOS rules we used for standard CCS. (See Table 2.2.) Moreover, since we identify $P$ with $\varepsilon(0).P$, we have that

$$P \xrightarrow{\alpha} P' \quad \varepsilon(0).P \xrightarrow{\alpha} P'$$

So, as you can easily check,

$$\varepsilon(1.4).\tau.\text{press.Off} + \text{press.Bright} \xrightarrow{\text{press}} \text{Bright}.$$  

(9.5)

Since the other press-labelled transition is only available after 1.4 units of time have elapsed and an internal action has occurred, we expect that this is the only
9.3. SYNTAX AND SOS RULES

Table 9.1: SOS Rules for TCCS ($d, d' \in \mathbb{R}_{\geq 0}$)

\[
\begin{align*}
\frac{P \xrightarrow{d} P'}{\varepsilon(d).P \xrightarrow{d+d'} P'} & \quad \frac{\varepsilon(d).P \xrightarrow{d} \varepsilon(d-d').P}{d' \leq d} \\
\frac{P \xrightarrow{d} P'}{K \xrightarrow{d} P'} & \quad \frac{\alpha.P \xrightarrow{d} \alpha.P}{\alpha \neq \tau} \\
\frac{P \xrightarrow{d} P'}{P[f] \xrightarrow{d} P'[f]} & \quad \frac{P \xrightarrow{d} P'}{P \setminus L \xrightarrow{d} P' \setminus L} \\
\end{align*}
\]

action transition that is initially possible for that term. However, the above term can delay some amount of time $d \leq 1.4$ by virtue of the transition

\[\varepsilon(1.4) \cdot \tau.\text{press.Off} + \text{press.Bright} \xrightarrow{d} \varepsilon((1.4 - d)) \cdot \tau.\text{press.Off} + \text{press.Bright} .\]

Note that, unlike action transitions, delay transitions like the one above do not resolve nondeterministic choices. This is in line with our intuition that delay transitions ought to be deterministic. In the above example, the transition (9.5) is available immediately, and it remains available after $d$ units of time have elapsed. When $d$ equals 0, we expect that, after the internal action has occurred, the press-labelled transition leading to the Off state becomes enabled, and therefore that the transitions

\[\varepsilon(0) \cdot \tau.\text{press.Off} + \text{press.Bright} \xrightarrow{\tau} \text{press.Off} \xrightarrow{\text{press.Off}} \text{Off}\]

are possible.

To formally capture our intuitive understanding of the effect of delay transitions over TCCS expressions, we introduce the collection of SOS rules in Table 9.1. (For the sake of clarity, we restrict for the moment to expressions that contain neither occurrences of the parallel composition operator nor of $\tau$ prefixes.) A transition $P \xrightarrow{d} Q$, with $d \in \mathbb{R}_{\geq 0}$, holds for TCCS expressions $P, Q$ if, and only if, it can be proven using these rules.

The SOS rules for standard actions are the same as for CCS.

Exercise 9.2 Prove the transition

\[\text{Light} \xrightarrow{d} \varepsilon((1.4 - d)) \cdot \tau.\text{press.Off} + \text{press.Bright} ,\]

with $d \leq 1.4$, using the above rules.
Exercise 9.3 (Strongly Recommended) Prove the following persistency property of action transitions:

For all processes $P, Q$, action $a$ and delay $d$, if $P \xrightarrow{a} P' \xrightarrow{d} Q$, then $Q \xrightarrow{a}$.

For simplicity, you may restrict yourselves to considering process terms that do not contain occurrences of constants.

Consider a process expression of the form

$$\sum_{i \in I} \varepsilon(d_i).\alpha_i.P_i.$$ 

Intuitively, we may think of the operational rules in Table 9.1 as implementing the following idea of the expected behaviour of this process. The process has a stopwatch, or clock, that is used to measure the amount of time that has elapsed since it last embarked in an action. As time progresses and the process idles, the value of the stopwatch increases. When this value is greater than, or equal to, $d_i$ for some $i \in I$, then action $\alpha_i$ becomes enabled and can be performed. If this happens, the process enters state $P_i$, the stopwatch is reset, and the future behaviour is determined according to the same approach. Note that this intuitive description of the dynamics of processes is fully in line with our view that the behaviour of processes consists of two alternating phases consisting of idling (when time passes) and action (when actions are performed instantaneously). Moreover, in the absence of parallel composition, the above approach to the description of the behaviour of processes can be “implemented” by using only one clock.

However, parallel composition is fundamental for the description of interacting reactive systems, and we therefore now examine what happens when we consider real-time parallel systems.

9.4 Parallel composition

Assume now that we have a user of our light switch whose behaviour is described by the following expression

$$\text{FastUser} \overset{\text{def}}{=} \text{press}.\varepsilon(0.3).\text{press}.\text{FastUser}.$$ 

We expect that the FastUser will be able to synchronize with the switch in the Off state immediately, resulting in the transition

$$(\text{FastUser} \parallel \text{Off}) \setminus \text{press} \xrightarrow{\tau} ((\varepsilon(0.3).\text{press}.\text{FastUser}) \parallel \text{Light}) \setminus \text{press}.$$
In the target state of the above transition, the press-transition of the left-hand expression is only available after a delay of 0.3 time units. As we have already seen in Exercise 9.2, the process Light can delay as much and reach the state

\[
\epsilon(1.1).\tau.\text{press.Off} + \text{press.Bright}.
\]

It is natural to expect the whole system can therefore perform the delay transition

\[
\left(\left(\epsilon(0.3)\cdot\text{press.FastUser} \mid \text{Light}\right) \setminus \text{press}\right) \xrightarrow{0.3} \left(\left(\text{press.FastUser} \mid (\epsilon(1.1)\cdot\tau.\text{press.Off} + \text{press.Bright})\right) \setminus \text{press}\right).
\]

(Recall that we identify an expression \( P \) with \( \epsilon(0).P \).) In the target state of the above transition, the FastUser is eager to press the button once more, and the press-transition of the switch leading to the Bright state is enabled. This means that the two transitions can synchronize without further delay, yielding the \( \tau \)-transition

\[
\left(\left(\text{press.FastUser} \mid (\epsilon(1.1)\cdot\tau.\text{press.Off} + \text{press.Bright})\right) \setminus \text{press}\right) \xrightarrow{\tau} \text{FastUser} \mid \text{Bright} \setminus \text{press}. \tag{9.6}
\]

(9.7)

This is nicely in agreement with our intuition.

Note, however, that both parallel components of the expression

\[
\left(\text{press.FastUser} \mid (\epsilon(1.1)\cdot\tau.\text{press.Off} + \text{press.Bright})\right)
\]

can delay indefinitely. How can we formally capture our intuition that the \( \tau \)-transition (9.6) must occur immediately, and therefore that the above term cannot delay even though both of its parallel components can? The solution adopted by Wang Yi in the design of TCCS and by the researchers who developed other process calculi for real-time systems (see, e.g., (Hennessy and Regan, 1995; Nicollin and Sifakis, 1994; Schneider, 1995)) is to postulate that the evolution of processes obeys the so-called maximal progress assumption. Intuitively, this means that if a process is ready to perform an action that is entirely under its control immediately, then it will do so without further delay. In the setting of Timed CCS, the only action that is entirely under the control of a process is the internal \( \tau \)-action. Therefore, the maximal progress assumption for this calculus can be formalized like:

For each TCCS process \( P \), if \( P \xrightarrow{\tau} \) then \( P \xrightarrow{d} \) for any \( d > 0 \).

In particular, in light of (9.6), this means that the expression

\[
\left(\left(\text{press.FastUser} \mid (\epsilon(1.1)\cdot\tau.\text{press.Off} + \text{press.Bright})\right) \setminus \text{press}\right)
\]
cannot delay any positive amount of time.

The maximal progress assumption is built in the operational semantics of Timed CCS by means of the following rules:

\[
\begin{align*}
\tau.P & \to \tau.P \\
P \xrightarrow{d} P' & \text{ and } Q \xrightarrow{d} Q' \text{ and NoSync}(P, Q, d) \\
P | Q \xrightarrow{d} P' | Q'
\end{align*}
\]

where the predicate NoSync\((P, Q, d)\) intuitively expresses that no synchronization between \(P\) and \(Q\) becomes available by delaying less than \(d\) time units. Formally, NoSync\((P, Q, d)\) holds if, and only if,

For each \(0 \leq d' < d\) and expressions \(P', Q'\), if \(P \xrightarrow{d'} P'\) and \(Q \xrightarrow{d'} Q'\), then \(P' | Q' \not\xrightarrow{\tau}\).

**Exercise 9.4** Argue, using the SOS rules for Timed CCS, that

\[
((\varepsilon(0.3).\text{press.FastUser}) | \text{Light}) \setminus \text{press} \xrightarrow{0.4}
\]

and that the expression

\[
((\text{press.FastUser}) | (\varepsilon(1.1).\tau.\text{press.Off} + \text{press.Bright})) \setminus \text{press}
\]

cannot delay a positive amount of time.

How long can the expression \(\varepsilon(\pi).\tau.0 + a.0\) delay? ♦

In order to familiarize ourselves better with the role played by the maximal progress assumption in describing the dynamics of processes, let us consider the possible interplay between the switch and a SlowUser whose behaviour is described by the following expression

\[
\text{SlowUser} \overset{\text{def}}{=} \text{press.}\varepsilon(1.7).\text{press.SlowUser}.
\]

As before, the SlowUser will be able to synchronize with the switch in the Off state immediately, resulting in the transition

\[
(\text{SlowUser} | \text{Off}) \setminus \text{press} \xrightarrow{\tau} ((\varepsilon(1.7).\text{press.SlowUser}) | \text{Light}) \setminus \text{press}.
\]

In the target state of the above transition, the press-transition of the left-hand expression is only available after a delay of 1.7 time units. However, because of the maximal progress assumption, the process Light can delay at most 1.4 units of time. (Why?) Delaying that much yields the transition

\[
((\varepsilon(1.7).\text{press.SlowUser}) | \text{Light}) \setminus \text{press} \xrightarrow{1.4}
((\varepsilon(0.3).\text{press.SlowUser}) | (\varepsilon(0).\tau.\text{press.Off} + \text{press.Bright})) \setminus \text{press}.
\]
Since the right-hand process can perform a \( \tau \)-transition, no further delay is possible, and the slow user cannot press the button leading to the Bright state. The system can now internally reach the state

\[
((\varepsilon(0.3) . \text{press}.\text{SlowUser}) \mid \text{press}.\text{Off}) \setminus \text{press}
\]

You should now be able to argue that the next button press issued by the user will bring the switch in the Off state.

**How Many Clocks?**

Consider, by way of example, a process expression of the form

\[
(\sum_{i \in I} \varepsilon(d_i) . \alpha_i . P_i) \mid (\sum_{j \in J} \varepsilon(e_j) . \beta_j . Q_i)
\]

We have already seen that, intuitively, the timing behaviour of each of the two parallel components can be described by using a single clock that is reset each time the relevant component performs one action. So, intuitively, the behaviour of the above parallel process can be described by using two local clocks, say \( x \) and \( y \), as follows:

- The process can idle \( d \) units of time, for some \( d \in \mathbb{R}_{\geq 0} \), provided that the following conditions are met:
  - If \( d_i \) is smaller than the value of \( x \) plus \( d \), then \( \alpha_i \neq \tau \)—that is, no \( \tau \)-transition of the left-hand expression becomes enabled by delaying less than \( d \) units of time—,
  - If \( e_j \) is smaller than the value of \( y \) plus \( d \), then \( \beta_j \neq \tau \)—that is, no \( \tau \)-transition of the right-hand expression becomes enabled by delaying less than \( d \) units of time—, and
  - If \( d_i \) is smaller than the value of \( x \) plus \( d \), and \( e_j \) is smaller than the value of \( y \) plus \( d \), then \( \alpha_i \) and \( \beta_j \) are not complementary—that is, no synchronization becomes enabled as the processes delay less than \( d \) units of time.
- If the process has idled \( d \) units of time, then
  - Update the values of the clocks \( x \) and \( y \) by adding \( d \) to them,
  - Perform one of the actions that becomes enabled after this delay, if any, and
– Reset the clock of the processes that were involved in that action. In particular, if the action was a synchronization, then both $x$ and $y$ are reset.

In fact, one clock is not sufficient to implement the above procedure. Moreover, as shown in (Godskesen and Larsen, 1992), in general the more parallel components we have in our system, the more clocks we need to “implement” its behaviour. A formal proof of this fact is beyond the scope of this introductory text.

9.5 Discussion

At least syntactically, Timed CCS is a simple extension of Milner’s original CCS that, as we have seen, can be used to describe some aspects of time dependent behaviour in the evolution of reactive systems. For instance, we have described how to specify time-outs in the calculus by means of a combination of urgent actions (the internal action $\tau$, in the setting of TCCS) and the maximal progress assumption. From an expressiveness viewpoint, however, the resulting calculus is not completely satisfactory. For example, the simple timed labelled transition system depicted in Example 9.1 cannot be described, up to isomorphism, using TCCS. To see this, recall that in Exercise 9.3, you showed that a sub-class of TCCS process terms affords the following persistency property of action transitions:

For all processes $P, Q$, action $a$ and delay $d$, if $P \xrightarrow{a}$ and $P \xrightarrow{d} Q$, then $Q \xrightarrow{a}$.

In fact, a slightly more elaborate argument shows that each TCCS process term affords the above property. On the other hand, the timed labelled transition system depicted in Example 9.1 does not have the above persistency property. In fact, state 5 in that timed labelled transition system has an outgoing $a$-labelled transition, but none of the states that it can reach by delaying a positive amount of time has that transition.

This lack of expressiveness of TCCS is somewhat unsatisfactory since the timed labelled transition system depicted in Example 9.1 can intuitively be captured by the following very informal, but hopefully natural and unambiguous, process description that uses a stopwatch:

1. Set the stopwatch to 0;
2. Let time pass—the amount of time that has elapsed is recorded by the stopwatch;
3. If the value of the stopwatch is 5, then action $a$ may be performed. In that case, go to step 1 above, and continue from there. If action $a$ is not performed at time 5, then idle forever.

If our desideratum is to have a formalism for the specification of real-time systems in which that type of process can be described, then TCCS falls short of our expectations. Moreover, one can argue that the use of urgent actions and of the maximal progress assumption in the implementation of features like time-outs is somewhat artificial. It would be more intuitive to have a formalism in which features like time-outs can be described without recourse to assumptions like action urgency or maximal progress.

The above example and our previous discussions seem to suggest that an automaton based formalism with some explicit notion of clocks (or stopwatches) that can be used to determine when transitions are available and when they are disabled might be a natural and powerful specification formalism for real-time behaviours. In the following section, we shall introduce one such formalism—namely, that of timed automata that were introduced by Alur and Dill in their seminal paper (Alur and Dill, 1994).
Chapter 10

Timed Automata

XXX Kim XXX could write some brief historical remarks about timed automata?

10.1 Motivation

Timed automata are essentially nondeterministic finite automata equipped with a finite number of real valued clocks so that transitions can be conditioned on clock values and performing a particular transition can reset selected clocks. We shall now intuitively introduce the formalism, showing how the light switch from the previous section can be described using the formalism of timed automata without recourse to assumptions like urgency of some actions or maximal progress. Graphically, we may suggest to model the light switch like in Figure 10.1.

Initially the switch is in the control location Off and, under certain circumstances, it can reach the other two locations Light and Bright. From the initial location, performing the action press will cause the clock $x$ to be reset to zero. This means that the value of the clock $x$ will be zero whenever we enter location

Figure 10.1: Light switch
Light. Therefore, the value of the clock $x$ represents the amount of time that has passed since we entered that location. The expressions $x > 1.4$ and $x \leq 1.4$ are called guards and they restrict the possibility of performing a transition. If the time that has elapsed since we entered the location Light is smaller than or equal to 1.4 then, after pressing the switch, we move to the location Bright. If the clock value is greater than 1.4 and the switch is pressed, we have to move to the location Off, because the guard labelling the edge from Light to Bright is not satisfied, and that edge is therefore disabled.

In what follows, we shall formalize this simple and intuitive model and develop the basic theory of timed automata.

### 10.2 Syntax of Timed Automata

As we have seen above, the main ingredients that timed automata add to the standard model of nondeterministic finite automata are the clocks, clock constraints (guards) and clock resets. These notions can be formally defined as follows.

Let us fix a finite set $C = \{x, y, \ldots \}$ which represents the clock names that we can use in the automaton.

**Definition 10.1** The set $B(C)$ of clock constraints (or guards) over the set of clocks $C$ is defined by the abstract syntax

$$g, g_1, g_2 ::= x \triangleright n \mid g_1 \land g_2$$

where $x \in C$ is a clock, $n \in \mathbb{N}$ and $\triangleright \in \{\leq, <, =, >, \geq\}$.

**Example 10.1** The following expressions belong to $B(C)$, where $C = \{x, y, z\}$:

- $x \leq 5$,
- $x \geq 0 \land x < 5$ and
- $x > 3 \land y = 2$.

We shall often write the constraints in the usual mathematical notation. For example $x \geq 0 \land x < 5$ can be written as $0 \leq x < 5$.

**Exercise 10.1** Is $x \leq 1.4$ a syntactically correct constraint?

Each clock from the set $C$ is assumed to store the amount of time elapsed from the last moment when the clock was reset. This can be formally expressed as a function $v : C \rightarrow \mathbb{R}_{\geq 0}$, which we shall call a (clock) valuation. The value of a
particular clock \( x \) is denoted by \( v(x) \). Assume, for example, that \( C = \{ x, y \} \), and consider a valuation \( v \) such that \( v(x) = 1.34 \) and \( v(y) = 5.333 \). We shall often denote the valuation \( v \) simply by \( [x = 1.34, y = 5.333] \).

Note that the values of clocks can be arbitrary non-negative real numbers. So, for instance, \( [x = \pi, y = \sqrt{2}] \) is a valuation.

In what follows, we will need two important operations, called delay and reset, which help us to manipulate clock valuations. Let \( v \) be a clock valuation. By \( v + d \) we denote a clock valuation where the value of every clock is increased by a given real number \( d \). For a given subset \( r \) of clocks, we use \( v[r] \) to denote the clock valuation where the values of clocks from \( r \) are set to zero and the values of the other clocks are the same as in \( v \). Formally,

- for each \( d \in \mathbb{R}_{\geq 0} \), the valuation \( v + d \) is defined by
  \[
  (v + d)(x) = v(x) + d, \quad \text{for each } x \in C;
  \]

- for each \( r \subseteq C \), the valuation \( v[r] \) is defined by
  \[
  v[r](x) = \begin{cases} 
  0 & \text{if } x \in r \\
  v(x) & \text{otherwise}.
  \end{cases}
  \]

**Remark 10.1** In case that \( r = \{ x \} \) is a singleton set, we shall often use an alternative notation for reset such that instead of \( v[\{ x \}] \) we write \( v[x \mapsto 0] \).

Now that we have in place the notions of clock constraints and clock valuations, we can naturally define when a clock constraint satisfies a given valuation, or alternatively how the constraint evaluates under the valuation.

**Definition 10.2** Let \( g \in \mathcal{B}(C) \) be a clock constraint for a given set of clocks \( C \) and let \( v : C \to \mathbb{R}_{\geq 0} \) be a clock valuation. **Evaluation** of clock constraints \( (v \models g) \) is defined inductively on the structure of \( g \) by

\[
  v \models x \bowtie n \quad \text{iff} \quad v(x) \bowtie n
\]

\[
  v \models g_1 \land g_2 \quad \text{iff} \quad v \models g_1 \quad \text{and} \quad v \models g_2
\]

where \( x \in C \) is a clock, \( n \in \mathbb{N}, g_1, g_2 \in \mathcal{B}(C) \) and \( \bowtie \in \{ \leq, <, =, >, \geq \} \). Note that the sign \( \bowtie \) on the left-hand side is a purely syntactic symbol while \( \bowtie \) on the right-hand side represents the standard corresponding arithmetic comparison on \( \mathbb{R}_{\geq 0} \).

If \( v \models g \) holds, we often write that “\( v \) satisfies \( g \)”. As usual, we write \( v \not\models g \) when \( v \) does not satisfy the constraint \( g \).
Example 10.2 Let \( C = \{x, y\} \), and consider the valuation \( v = [x = 1.2, y = 3.01] \). We can easily see that

- \( v \models x > 1 \land x \leq 2 \),
- \( v \models x > 0 \land y \geq 3 \) and
- \( v \not\models y \leq 3 \land x \geq 1 \).

Exercise 10.2 Can you give an example of a clock constraint that is satisfied by every valuation? What about one that is satisfied by no valuation?

Definition 10.3 Two clock constraints \( g_1 \) and \( g_2 \) are equivalent if, and only if, they are satisfied by the same valuations—that is, for each valuation \( v \),

\[
v \models g_1 \iff v \models g_2 .
\]

Example 10.3 The clock constraints \( x \leq 5 \land x \geq 5 \) and \( x = 5 \) are equivalent, and so are \( y \geq 5 \land y \geq 0 \) and \( y \geq 5 \). (Why?)

Exercise 10.3 (Strongly Recommended)

1. A constraint \( g \) is interval closed if, for each valuation \( v \) and non-negative real number \( d \), it holds that \( v \models g \) and \( v + d \models g \) imply that \( v + d' \models g \), for each \( 0 \leq d' \leq d \).

   Prove that each constraint in \( B(C) \) is interval closed.

2. Show that there is no constraint in \( B(C) \) expressing that \( x \) is not equal to \( 2 \).

3. Let us define the relation \( \leq \) over the collection of valuations as:

   \[
   v \leq v' \iff v(x) \leq v'(x), \text{ for each } x \in C .
   \]

   Prove that the relation so defined is a partial order.

4. A constraint \( g \) is downward closed if \( v' \models g \) and \( v \leq v' \) imply \( v \models g \), for all valuations \( v, v' \).

   Give examples of constraints that are downward closed. Are there constraints that are not downward closed?
5. Show that constraints of the form \( x < n \) are not definable in terms of the other guards in \( B(C) \). That is, argue that a constraint of the form \( x < n \) is not equivalent to any constraint generated by the following grammar:

\[
g, g_1, g_2 ::= x \preceq n \mid g_1 \land g_2
\]

where \( x \in C \) is a clock, \( n \in \mathbb{N} \) and \( \preceq \in \{\leq, =, >, \geq\} \).

We are now ready to provide a formal definition of timed automata.

**Definition 10.4** A timed automaton over a finite set of clocks \( C \) and a finite set of actions \( \text{Act} \) is a quadruple

\[(L, \ell_0, E, I)\]

where

- \( L \) is a finite set of locations, ranged over by \( \ell \),
- \( \ell_0 \in L \) is the initial location,
- \( E \subseteq L \times B(C) \times \text{Act} \times 2^C \times L \) is a finite set of edges, and
- \( I : L \to B(C) \) assigns invariants to locations.

We usually write \( \ell \overset{g, a, r}{\longrightarrow} \ell' \) instead of \((\ell, g, a, r, \ell') \in E\). For such an edge, \( \ell \) is called the source location, \( g \) is the guard, \( a \) is the action, \( r \) is the set of clocks to be reset and \( \ell' \) is the target location.

Timed automata are often given in their graphical representation like in Figure 10.1. Locations are drawn as nodes in the graph, and the initial location is marked with a double circle. Edges in the graph have attributes: the beginning of an edge is assigned a guard, in the middle of the edge there is an action name and resets are written at the end of the edge using the notation \( x := 0 \) for each clock that should be reset. Invariants are placed next to their corresponding locations. (Their role in the behaviour of timed automata will become clear in what follows. For the moment, you may think of them as imposing restrictions on the values that the clocks may have in control locations.) Irrelevant guards and invariants—that is, those that are always satisfied—are omitted in the picture.

**Example 10.4** The light switch from Figure 10.1 can be formally described as follows (let \( g_l = x \geq 0 \) be a guard that is true in any valuation):
• \( C = \{ x \} \)
• \( L = \{ \text{Off}, \text{Light}, \text{Bright} \} \)
• \( \ell_0 = \text{Off} \)
• \( E = \{ \begin{array}{l} \text{Off} \xrightarrow{\text{gt}, \text{press}, \{ x \}} \text{Light}, \text{Light} \xrightarrow{x > 14, \text{press}, \emptyset} \text{Off}, \\
\text{Light} \xrightarrow{x \leq 14, \text{press}, \emptyset} \text{Bright}, \text{Bright} \xrightarrow{\text{gt}, \text{press}, \emptyset} \text{Off} \end{array} \} \)
• \( I(\text{Off}) = I(\text{Light}) = I(\text{Bright}) = \text{gt}. \)

**Remark 10.2** Note that in the example above we have replaced the time point 1.4 with 14. This is due to the requirement that constants in the guards be natural numbers. This is, however, not a real restriction. We can also consider guards where constants are rational numbers (irrational numbers in guards would be impossible to write in a finite way). In this case, we can always multiply all the constants in the guards by an appropriate number in order to raise them to natural numbers. Such stretching of time does not have any significant influence on the behaviour, it is essentially like saying that, instead of a clock value being equal to 0.145 seconds, it is equal to 145 milliseconds.

We shall now discuss the intended behaviour of timed automata. Intuitively, a timed automaton can be in exactly one of its control locations at each stage of its computation. However, knowing the present control location is not enough to determine which of the outgoing edges can be taken next, if any. A snapshot of the current state of the computation should also remember the present clock values. Therefore, a suitable notion of state of the computation of a timed automaton consists of a pair \((\ell, v)\), with \(\ell\) being the control location the automaton is in, and \(v\) being the valuation determined by the current clock values. The pair \((\ell, v)\) is a legal state of the timed automaton only if the valuation \(v\) satisfies the invariant of location \(\ell\). (Initially, the control location is \(\ell_0\) and the value of each clock is 0.) If there is an edge whose source location equals the current location \(\ell\), and whose guard is satisfied by the current valuation \(v\), then we can follow that edge, thereby changing the current location to the target location of the edge and resetting the set of clocks labelling the edge. Another possibility is to delay in the current location by increasing simultaneously the value of all clocks by a given amount of time \(d\), without changing the control location. This is possible only if the invariant of the current location is satisfied by the valuation \(v + d\). (Since invariants are interval closed—see Exercise 10.3—, this also means that the invariant is satisfied by all of the intermediate valuations \(v + d'\) with \(0 \leq d' \leq d\).) For example, if the current
value of the clock $x$ is 0, and the invariant of the present location is $x \leq 1$, then the timed automaton can delay 1 time unit, but not 1.00001 units of time.

The reader might have already observed that essentially we have intuitively defined a timed transition system generated by a given timed automaton. These ideas can be formalized as follows.

**Definition 10.5** Let $A = (L, \ell_0, E, I)$ be a timed automaton over a set of clocks $C$ and a set of actions $\text{Act}$. We define the timed transition system $T(A)$ generated by $A$ as $T(A) = (\text{Proc}, \text{Lab}, \{ \overset{a}{\rightarrow} \mid a \in \text{Lab} \})$ where

- $\text{Proc} = \{ (\ell, v) \mid (\ell, v) \in L \times (C \rightarrow \mathbb{R}_{\geq 0})$ and $v \models I(\ell) \}$—i.e., states are of the form $(\ell, v)$ where $\ell$ is a location of the timed automaton and $v$ is a valuation that satisfies the invariant of $\ell$,
- $\text{Lab} = \text{Act} \cup \mathbb{R}_{\geq 0}$ is the set of labels, and
- the transition relation is defined as follows:

  - $(\ell, v) \overset{a}{\rightarrow} (\ell', v')$ if there is an edge $(\ell \overset{a, r}{\rightarrow} \ell') \in E$ such that $v \models g$, $v' = v[r]$ and $v' \models I(\ell')$
  - $(\ell, v) \overset{d}{\rightarrow} (\ell, v + d)$ for all $d \in \mathbb{R}_{\geq 0}$ such that $v \models I(\ell)$ and $v + d \models I(\ell)$.

Let $v_0$ denote the valuation such that $v_0(x) = 0$ for all $x \in C$. If $v_0$ satisfies the invariant of the initial location $\ell_0$, we shall call $(\ell_0, v_0)$ the initial state (or initial configuration) of $T(A)$.

**Example 10.5** Consider the timed automaton $A$ defined in the following picture (there is one edge labeled by $a$ with the guard $x \leq 1$ which resets the clock $x$ and the invariant in the location $\ell_0$ is $x \leq 2$).

$$x \leq 2 \xrightarrow{a} \ell_0 \xrightarrow{a} x \leq 1$$

A fragment of the transition system $T(A)$ follows (there are in fact infinitely many different reachable states for every $x$ in the interval $[0, 2]$).

$$\overset{a}{\rightarrow} (\ell_0, [x = 0]) \overset{0.6}{\rightarrow} (\ell_0, [x = 0.6]) \overset{0.4}{\rightarrow} (\ell_0, [x = 1]) \overset{0.3}{\rightarrow} (\ell_0, [x = 1.3]) \overset{0.7}{\rightarrow} (\ell_0, [x = 2])$$
Note that from the state \((\ell_0, [x = 1.3])\) it is not possible to perform the transition under the action \(a\) and the state \((\ell_0, [x = 2])\) is essentially stuck as the only available transition is a time elapsing step with the time delay 0.

There is a fundamental difference between the situations where a clock constraint is used in the guard or in the invariant. This can be demonstrated by means of the simple example in Figure 10.2. In the timed automaton a) on the left, \(x \leq 1\) is a guard. There is no restriction on time elapsing steps and hence arbitrarily long delays are possible. This means that, as long as the value of the clock \(x\) is smaller than or equal to 1, we can perform the transition \(a\) and reset the clock \(x\). However, if the total time delay after the last reset of the clock is strictly greater than 1 then it is not possible to enable the action \(a\) in the future and the only available transitions are the delay steps. In the timed automaton b) on the right, \(x \leq 1\) is used in the invariant. This means that it is never possible to delay more than 1 time unit and hence during each execution the action \(a\) is always available.

**Exercise 10.4** Let \(A\) be a timed automaton. Prove that \(T(A)\) is a TLTS in the sense of Definition 9.1.

**Exercise 10.5** Give a timed automaton \(A\) whose associated TLTS \(T(A)\) is, modulo a renaming of the names of the states, precisely that in Example 9.1.

**Exercise 10.6** Is there a timed automaton whose associated TLTS has only one state? Argue for your answer.

**Exercise 10.7 (For the Keenest)** Show how to translate TCCS expressions into timed automata in a syntax directed fashion. Your translation should be such that the timed labelled transitions determined by the source TCCS expression and the target timed automaton are isomorphic. Use your translation and your answer to Exercise 10.5 to argue that the formalism of timed automata is more expressive than TCCS.

**Exercise 10.8** In Exercise 10.3(2), you proved that there is no constraint in \(B(C)\) expressing that \(x \neq 2\). Assume now that, using the formalism of timed automata, we wish to model a situation in which an action \(a\) is enabled in some location \(\ell\) unless \(x = 2\). Is this possible? If so, how would you do so?

Perhaps ADD A MORE COMPLEX EXAMPLE OF TIMED AUTOMATON??
10.2. SYNTAX OF TIMED AUTOMATA

Figure 10.2: Clock constraint in the guard vs. in the invariant
10.3 Timed and Untimed Trace Equivalence

We shall now investigate behavioural aspects of timed automata. Even though we often refer to timed automata, these notions are independent of the actual formalism used and they can be defined purely in terms of timed labelled transition systems.

A natural way to begin with is to define timed traces of timed transition systems, similarly as we have considered traces for ordinary labelled transition systems.

Let $A = (L, \ell_0, E, I)$ be a timed automaton over a set of clocks $C$ and a set of actions $\text{Act}$.

**Definition 10.6** A sequence $(t_1, a_1)(t_2, a_2)(t_3, a_3)\ldots$ where $t_i \in \mathbb{R}_{\geq 0}$ and $a_i \in \text{Act}$ is called a finite or infinite timed trace of $A$ iff there is a finite or infinite transition sequence

$$
(\ell_0, v_0) \xrightarrow{d_1} (\ell_1, v_1) \xrightarrow{a_1} (\ell_2, v_2) \xrightarrow{d_2} (\ell_3, v_3) \xrightarrow{a_2} (\ell_4, v_4) \xrightarrow{d_3} (\ell_5, v_5) \xrightarrow{a_3} \ldots
$$

in $T(A)$ such that $v_0(x) = 0$ for all $x \in C$ and, for each $i$,

$$
t_i = t_{i-1} + d_i \quad \text{where } t_0 = 0.
$$

The intuition is that the real number $t_i$ represents the absolute time (the timestamp) at which $a_i$ happened since the start of the computation of the automaton $A$. Note that the sequence of time-stamps $t_1, t_2, \ldots$ is nondecreasing.

**Example 10.6** Consider the light switch from Figure 10.1. The following sequence can be easily seen to be a finite timed trace.

$$(2.3, \text{press})(2.5, \text{press})(2.51, \text{press})(5.6, \text{press})(5.6, \text{press})(7.0, \text{press}) .$$

In fact any nondecreasing sequence of time-stamps induces a timed trace of this timed automaton. (Why?)

If we consider the timed automata a) and b) from Figure 10.2 then e.g. the sequence

$$(0.2, a)(0.5, a)(1.5, a)(1.5, a)(2.0, a)$$

is a finite timed trace of both a) and b). In fact, it is not too hard to argue that those two timed automata afford the same timed traces. (Do so!)

We can now define the notion of timed language equivalence.
10.3. **TIMED AND UNTIMED TRACE EQUIVALENCE**

**Definition 10.7** The set of all finite and infinite timed traces of a timed automaton $A$ is denoted by $L(A)$ and is called the **timed language of $A$**. Timed automata $A_1$ and $A_2$ are **timed language equivalent** iff $L(A_1) = L(A_2)$.

As remarked above, the two timed automata in Figure 10.2 are timed language equivalent.

Sometimes we would like to abstract away from the particular time points when actions happen and consider only the action sequences that can be performed from the initial configuration of a given timed automaton. For this purpose, we shall define untimed traces and untimed trace equivalence over timed automata.

**Definition 10.8** We say that $a_1a_2a_3\ldots$ is an **untimed trace of $A$** iff there exist $t_1, t_2, t_3, \ldots \in \mathbb{R}_{\geq 0}$ such that $(t_1, a_1)(t_2, a_2)(t_3, a_3)\ldots$ is a timed trace of $A$.

**Definition 10.9** The set of all untimed traces of $A$ is denoted by $L_u(A)$ and called the **untimed language of $A$**. Timed automata $A_1$ and $A_2$ are **untimed language equivalent** iff $L_u(A_1) = L_u(A_2)$.

The proof of the following theorem is straightforward and follows directly from the definitions.

**Theorem 10.1** Any two timed language equivalent automata are also untimed language equivalent.

**Exercise 10.9** Prove the above theorem.

The converse of the above theorem does not hold as demonstrated by the following example.

**Example 10.7** Consider the following two timed automata.

\[ a) \quad x := 0 \quad a \quad x \leq 1 \]
\[ b) \quad x := 0 \quad a \quad x = 1 \]

We can easily observe that automaton $a$ affords the timed trace $(0, a)$, but automaton $b$ does not. Therefore the two automata are not timed language equivalent. Note, however, that they are untimed language equivalent. Moreover, each timed trace of automaton $b$ is also a timed trace of automaton $a$.

**Exercise 10.10** Prove the claims that we have made in the previous example.

As in the case of ordinary labelled transitions systems, language equivalence is not always the most suitable notion of behavioural equivalence to consider because, as argued in Sect. 3.2, it does not faithfully describe the deadlock behaviour of processes. We therefore proceed to introduce the notions of timed and untimed strong bisimilarity.
10.4 Timed and Untimed Bisimilarity

From our discussion so far in this chapter, we know that the semantics of timed automata is given in terms of timed labelled transition systems, which can, in fact, be viewed as standard labelled transition systems. Hence the first notion of bisimilarity can be naturally defined as strong bisimilarity. In the timed case, we shall call it timed bisimilarity. This implies that in timed bisimilarity both ordinary actions and time elapsing steps are considered as visible actions, which means that we can observe the precise duration of time delays.

**Definition 10.10** Timed automata $A_1$ and $A_2$ are **timed bisimilar** iff their initial states in the timed transition systems $T(A_1)$ and $T(A_2)$ generated by $A_1$ and $A_2$ are strongly bisimilar in the sense of Definition 3.2.

In order to understand better the notion of timed bisimilarity, we will present a few examples.

**Example 10.8** Consider the following timed automata with initial locations $A$ and $A'$ over the set of clocks $C = \{x\}$.

We shall argue that the given timed automata are timed bisimilar. In order to do so, we have to establish that their initial states $(A, v_0)$ and $(A', v_0)$ where $v_0(x) = 0$ are strongly bisimilar. This can be demonstrated, e.g., by defining a relation $R$ of strong bisimulation as follows:

$$
\{(A, [x = d]), (A', [x = d]) \mid d \in \mathbb{R}_{\geq 0}\} \cup \\
\{(B, [x = d + 1]), (B', [x = d]) \mid d \in \mathbb{R}_{\geq 0}\} \cup \\
\{((C, [x = d]), (C', [x = d'])) \mid d, d' \in \mathbb{R}_{\geq 0}\}.
$$

One can easily see that $((A, v_0), (A', v_0)) \in R$ and we leave it to the reader to verify that $R$ is indeed a strong bisimulation. (Do so!)
Example 10.9 Consider the following two timed automata.

We will demonstrate that the initial states of those automata are not timed bisimilar by finding a universal winning strategy for the attacker in the strong bisimulation game starting from the pair \((A, [x = 0])\) and \((A', [x = 0])\). In the first round the attacker plays the delay transition \((A, [x = 0]) \xrightarrow{1.7} (A, [x = 1.7])\) and the defender can only answer by \((A', [x = 0]) \xrightarrow{1.7} (A', [x = 1.7])\). The game continues in the second round from the pair \((A, [x = 1.7])\) and \((A', [x = 1.7])\). Now the attacker can switch sides and play \((A', [x = 1.7]) \xrightarrow{a} (B', [x = 0])\). The defender cannot answer to this move from \((A, [x = 1.7])\). Hence the attacker has a universal winning strategy. This implies that \((A, [x = 0]) \not\sim (A', [x = 0])\), and the given timed automata are not timed bisimilar.

The reason why the two transition systems above are not timed bisimilar is that in case of timed bisimilarity a particular time elapsing step in one automaton has to be matched by a timed elapsing step of exactly the same duration in the other. This might sometimes be too strict an assumption since delays can be arbitrary real numbers, and one might try to relax this requirement. One possibility is to require that a time delay in one process has to be matched by a time delay in the other process but possibly of a different duration. We shall call this notion untimed bisimilarity and proceed to define it formally.

There are essentially two equivalent ways to define untimed bisimilarity: we can either modify the notion of strong bisimilarity or modify the underlying timed transition systems. We will sketch the second possibility here.

Let \(T = (\text{Proc}, \text{Lab}, \{a \rightarrow T \mid a \in \text{Lab}\})\) be a timed transition system where \(\text{Lab} = \text{Act} \cup \mathbb{R}_{\geq 0}\). Assume that \(\varepsilon\) is a new action such that \(\varepsilon \notin \text{Lab}\). We construct the untimed labelled transition system

\[ T_\varepsilon = (\text{Proc}, \text{Act} \cup \{\varepsilon\}, \{a \rightarrow \mid a \in \text{Act} \cup \{\varepsilon\}\}) \]

thus:

- for each transition \(s \xrightarrow{a} T s'\) in \(T\), where \(a \in \text{Act}\), we add the transition \(s \xrightarrow{a} \varepsilon s'\) also to \(T_\varepsilon\), and
• for each transition \( s \xrightarrow{d} T s' \) in \( T \), where \( d \in \mathbb{R}_{\geq 0} \), we add the transition \( s \xrightarrow{\varepsilon} s' \) for the new action \( \varepsilon \) to \( T_\varepsilon \).

(Note that \( s \xrightarrow{\varepsilon} s \) holds for each state \( s \). Can you see why?) This means that the label of each time elapsing transition in \( T \) is replaced by \( \varepsilon \) and all the other standard transitions are preserved.

**Definition 10.11** Let \( A_1 \) and \( A_2 \) be timed automata with untimed transition systems \( T_\varepsilon(A_1) \) and \( T_\varepsilon(A_2) \). We say that \( A_1 \) and \( A_2 \) are *untimed bisimilar* iff their initial states in the untimed transition systems \( T_\varepsilon(A_1) \) and \( T_\varepsilon(A_2) \) are strongly bisimilar in the sense of Definition 3.2.

**Example 10.10** We will demonstrate that the timed nonbisimilar automata from Example 10.9 are equivalent w.r.t. untimed bisimilarity. Consider the following relation \( R \).

\[
\{(A, [x = d]), (A', [x = d'])\} \cup \{(A, [x = d]), (A', [x = d'])\} \cup \{(B, [x = d]), (B', [x = d'])\} \mid d, d' \in \mathbb{R}_{\geq 0}\}.
\]

It remains to verify that \( R \) is a strong bisimulation over the corresponding untimed transition systems and that the initial states belong to \( R \). We will examine one case of the analysis necessary to show that \( R \) is a strong bisimulation. Consider the pair \((A, [x = d])\) and \((A', [x = d'])\) for some \( d \) and \( d' \) such that \( 0 \leq d \leq 1 \) and \( 0 \leq d' \leq 2 \). There are three possible types of moves from \((A, [x = d])\) (the situation from \((A', [x = d'])\) is symmetric).

- **The move** \((A, [x = d]) \xrightarrow{a} (B, [x = 0])\) can be matched by \((A', [x = d']) \xrightarrow{a} (B', [x = 0])\) and the resulting pair surely belongs to \( R \).

- **The move** \((A, [x = d]) \xrightarrow{d''} (A, [x = d + d'])\) such that \( d + d'' \leq 1 \) can be matched by \((A', [x = d']) \xrightarrow{0} (A', [x = d'])\) and the resulting pair belongs to \( R \). This answer is possible because all time elapsing steps are observed only as the action \( \varepsilon \).

- **The move** \((A, [x = d]) \xrightarrow{d''} (A, [x = d + d'])\) such that \( d + d'' > 1 \) can be matched by, e.g., \((A', [x = d']) \xrightarrow{3} (A', [x = d' + 3])\) for the same reasons as above, and the resulting pair of states belongs to \( R \).

The reader is invited to finish the analysis of the remaining cases.
10.4. TIMED AND UNTIMED BISIMILARITY

As illustrated by the above example, there are untimed bisimilar timed automata that are not timed bisimilar. On the other hand, one can observe the validity of the following theorem.

**Theorem 10.2** Any two timed bisimilar timed automata are also untimed bisimilar.

**Proof:** Any relation which is a timed bisimulation is also an untimed bisimulation. One can easily argue for this fact, e.g., by using bisimulation games. Whenever the defender has an answer to an attack under a time elapsing action $d$ then the same defence is also valid when the time elapsing label is replaced with $\varepsilon$. $\square$

**Exercise 10.11** Argue for each of the following claims in the positive case or give a counter example in the negative case.

- If two timed automata are timed bisimilar then they are also timed trace equivalent.
- If two timed automata are timed bisimilar then they are also untimed trace equivalent.
- If two timed automata are untimed bisimilar then they are also untimed trace equivalent.
- If two timed automata are untimed bisimilar then they are also timed trace equivalent.
- If two timed automata are timed trace equivalent then they are also untimed bisimilar.

**Exercise 10.12** Let $T$ be a timed transition system. Let us consider a labelled transition system $T'$ where every time-delay action $d \in \mathbb{R}_{\geq 0}$ is replaced with the silent action $\tau$. We now define that two states $p$ and $q$ from the timed transition system $T$ are time abstracted bisimilar if and only if $p$ and $q$ are weakly bisimilar in $T'$ in the sense of Definition 3.4.

- Is the notion of time abstracted bisimilarity equivalent to untimed bisimilarity?
- If yes, prove your claim. If no, give a counter example.
10.5 Region Graph

Even the simplest timed automata generate timed transition systems with infinitely (even uncountably) many reachable states. This is due to the fact that states of timed automata contain not only the control location but also the particular valuation of clocks. In general we have uncountably many different valuations already in the situation when only one clock is considered (the value of this clock can be in general any number from $\mathbb{R}_{\geq 0}$).

Since good formalisms for the description of reactive systems should also support methods for their algorithmic analysis, a natural question the reader may ask at this point is the following:

Is any automatic verification approach like bisimilarity checking, model checking or reachability analysis possible at all over timed automata?

Surprisingly, the answer to this question is positive, and in what follows we shall discuss a fundamental approach—due to Alur and Dill (Alur and Dill, 1990; Alur and Dill, 1992; Alur and Dill, 1994) and called the region graph technique—, which will enable us to draw such a conclusion. XXX Kim XXX add some reference here, please.

The key idea behind the region technique is very simple: even though the collection of valuations for a given timed automaton is uncountably infinite, it can be partitioned into finitely many equivalence classes in such a way that any two valuations from the same equivalence class will not create any “significant difference” in the behaviour of the system.

Assume a given timed automaton $A$ over a set of clocks $C$. Formally, our goal will be to define effectively an equivalence relation $\equiv$ over clock valuations, $\equiv \subseteq (C \rightarrow \mathbb{R}_{\geq 0}) \times (C \rightarrow \mathbb{R}_{\geq 0})$, such that

1. $v \equiv v'$ implies that the states $(\ell, v)$ and $(\ell, v')$ are untimed bisimilar for each location $\ell$ of the automaton $A$, and

2. $\equiv$ has only finitely many equivalence classes—i.e., the set

$$\{ [v]_{\equiv} \mid v \in (C \rightarrow \mathbb{R}_{\geq 0}) \} ,$$

where $[v]_{\equiv} = \{ v' \mid v' \equiv v \}$, is finite. We shall call $[v]_{\equiv}$ the equivalence class represented by $v$.

Before embarking in the definition of the equivalence relation $\equiv$, we first introduce some necessary notation.
Definition 10.12 Let $d \in \mathbb{R}_{\geq 0}$ be a real number. By $\lfloor d \rfloor$ we denote the integer part of $d$ and $\text{frac}(d)$ stands for the fractional part of $d$. Any $d \in \mathbb{R}_{\geq 0}$ can be now written as $d = \lfloor d \rfloor + \text{frac}(d)$.

Example 10.11 By the above definition, $\lfloor 2.345 \rfloor = 2$ and $\text{frac}(2.345) = 0.345$.

We shall now proceed to motivate the definition of the relation $\equiv$ in stepwise fashion.

Let $A$ be a timed automaton and let $x \in C$ be one of its clocks. We define $c_x \in \mathbb{N}$ as the largest constant against which the clock $x$ is ever compared either in the guards or in the invariants present in $A$. For instance, $c_x$ is 2 (respectively, 1) for the timed automaton on the left (respectively, on the right) in Example 10.9.

The first observation we can make is that the specific value of the clock $x$ is irrelevant for the behaviour of the timed automaton, if that value is strictly greater than the constant $c_x$. In other words, for each clock constraint $g$ in the automaton $A$, the satisfiability of $g$ does not depend on the concrete value of the clock $x$ as long as it is greater than $c_x$. For instance, in the timed automaton on the left in Example 10.9 a valuation that assigns a value greater than 2 to $x$ satisfies none of the clock constraints in that timed automaton.

Hence a first approximation of the relation $\equiv$ can be as follows:

\[ v \equiv v' \text{ if, and only if, } v(x) = v'(x), \text{ whenever } v(x) \leq c_x \text{ or } v'(x) \leq c_x. \]

Equivalently, and perhaps more explicitly, we have that $v \equiv v'$ holds whenever, for each clock $x$, either $v(x) = v'(x)$ or both $v(x)$ and $v'(x)$ are greater than $c_x$.

The above argument used to motivate the definition of $\equiv$ implies that $v \equiv v'$ means that, for each location $\ell$ in the timed automaton, the states $(\ell, v)$ and $(\ell, v')$ are indeed untimed bisimilar. (Prove this!) However, the equivalence relation $\equiv$ still has infinitely many equivalence classes, and therefore does not satisfy the second requirement listed above. (Why?)

In order to motivate our further improvement upon the definition of $\equiv$, let us consider the following timed automaton:

In this automaton $c_x = 2$. Consider the following two configurations of the timed automaton above: $(\ell_0, [x = 0.12])$ and $(\ell_0, [x = 0.97])$. A short reflection makes us realize that these configurations are untimed bisimilar. (Argue for this claim!) Similarly $(\ell_0, [x = 1.23])$ and $(\ell_0, [x = 1.467])$ are surely untimed bisimilar. This
might motivate us to claim that the fractional parts of the clocks are irrelevant and to formulate the following refinement of condition (10.1):

\[ v \equiv v' \text{ if, and only if, for each } x \in C, \]
\[ [v(x)] = [v'(x)], \text{ or both } v(x) \text{ and } v'(x) \text{ are greater than } c_x. \]  

We can now see that \( \equiv \) has finitely many equivalence classes as required (only the integer parts of the clocks up to a given constant matter). The question is whether \( v \equiv v' \) does indeed imply that, for each location \( \ell \), the states \( (\ell, v) \) and \( (\ell, v') \) are untimed bisimilar. The answer is, unfortunately, negative. For example, the valuations \([x = 1]\) and \([x = 1.4]\) should be equivalent according to our definition of \( \equiv \), but in the timed automaton \( A \) the state \((\ell_0, [x = 1])\) can still perform the action \( a \) while \((\ell_0, [x = 1.4])\) cannot.

This leads us to the observation that if the fractional part of a clock is equal to 0, we should consider this as a “special” situation because the guards can distinguish such a clock value from a clock which has the same integer part but non-zero fractional part. Therefore we add the following requirement to condition (10.2):

\[ v \equiv v' \text{ only if, for each } x \in C \text{ such that } v(x) \leq c_x, \]
\[ (\frac{v(x)}{v'(x)}) = 0 \iff (\frac{v'(x)}{v'(x)}) = 0. \]

We can now see that \( \equiv \) has finitely many equivalence classes as required (only the integer parts of the clocks up to a given constant matter). The question is whether \( v \equiv v' \) does indeed imply that, for each location \( \ell \), the states \( (\ell, v) \) and \( (\ell, v') \) are untimed bisimilar. The answer is, unfortunately, negative. For example, the valuations \([x = 1]\) and \([x = 1.4]\) should be equivalent according to our definition of \( \equiv \), but in the timed automaton \( A \) the state \((\ell_0, [x = 1])\) can still perform the action \( a \) while \((\ell_0, [x = 1.4])\) cannot.

This leads us to the observation that if the fractional part of a clock is equal to 0, we should consider this as a “special” situation because the guards can distinguish such a clock value from a clock which has the same integer part but non-zero fractional part. Therefore we add the following requirement to condition (10.2):

\[ v \equiv v' \text{ only if, for each } x \in C \text{ such that } v(x) \leq c_x, \]
\[ (\frac{v(x)}{v'(x)}) = 0 \iff (\frac{v'(x)}{v'(x)}) = 0. \]

Exercise 10.13 Prove the claim we have just made.

The number of equivalences classes of \( \equiv \) still remains finite because we are refining an equivalence relation that already had finitely many equivalence classes. However, the equivalence relation \( \equiv \) is still too coarse as demonstrated by the following timed automaton:

Consider the valuations \( v_1 = [x = 0.8, y = 0.3] \) and \( v_2 = [x = 0.5, y = 0.9] \). You should be able to convince yourselves that, by the requirements (10.2) and (10.3), we have that \( v_1 \equiv v_2 \). However, the states \((\ell_0, v_1)\) and \((\ell_0, v_2)\) are not untimed bisimilar. This can be easily seen since state \((\ell_0, v_1)\) can first delay 0.2 time units, perform the action \( a \), further delay 0.5 time units, and finally perform the action \( b \). On the other hand the configuration \((\ell_0, v_2)\) cannot match this behaviour because in order to perform the action \( a \) we have to delay exactly 0.5 time units, which
means that the value of the clock $y$ necessarily grows to 1.4, disabling action $b$ forever.

The reason why we have this problem is due to the fact that $v_1(x) > v_1(y)$ while $v_2(x) < v_2(y)$. This indicates that the ordering of the fractional parts between two different clocks plays an important role. This motivates us to add the following requirement on top of (10.2) and (10.3):

$$v \equiv v' \text{ only if, for all } x, y \in C \text{ with } v(x) \leq c_x \text{ and } v(y) \leq c_y,$$

$$(\frac{v(x)}{\mu} \leq \frac{v(y)}{\mu}) \iff \frac{v'(x)}{\mu} \leq \frac{v'(y)}{\mu}.$$ 

In fact, this is all that we have to do in order to establish our goal that $\equiv$ has finitely many equivalence classes and still preserves untimed bisimilarity. We now provide a summary of the final definition, and we also state the main theorem to the effect that the equivalence relation $\equiv$ has the desired properties (full proof can be found e.g. in citexxx XXX Kim XXX which reference to put here?).

**Definition 10.13** Let $A$ be a timed automaton. We say that two clock valuations $v$ and $v'$ are equivalent, and write $v \equiv v'$, iff

1. for each $x \in C$, we have that either both $v(x)$ and $v'(x)$ are greater than $c_x$ or

$$\lfloor v(x) \rfloor = \lfloor v'(x) \rfloor;$$

2. for each $x \in C$ such that $v(x) \leq c_x$ we have

$$\frac{v(x)}{\mu} = 0 \iff \frac{v'(x)}{\mu} = 0; \text{ and}$$

3. for all $x, y \in C$ such that $v(x) \leq c_x$ and $v(y) \leq c_y$ we have

$$\frac{v(x)}{\mu} \leq \frac{v(y)}{\mu} \iff \frac{v'(x)}{\mu} \leq \frac{v'(y)}{\mu}.$$ 

$\diamond$

**Remark 10.3** Note that if $v \equiv v'$, then, for all $x, y \in C$ such that $v(x) \leq c_x$ and $v(y) \leq c_y$, we have that

$$v(x) = v(y) \iff v'(x) = v'(y).$$

Can you justify this claim? $\diamond$
Exercise 10.14 Let $A$ be a timed automaton and let $g$ be one of its clock constraints. Assume that $v \equiv v'$ and $v \models g$. Show that $v' \models g$ also holds.

Would this property continue to hold if we allowed for clock constraints of the form $x - y \bowtie n$, where $x, y \in C$ are clocks, $n \in \mathbb{N}$ and $\bowtie \in \{\leq, <, =, >, \geq\}$? Argue for your answer.

Theorem 10.3 Let $A$ be a timed automaton. The equivalence relation $\equiv$ partitions the clock valuations of $A$ into finitely many equivalence classes. Moreover, whenever $v$ and $v'$ are in the same equivalence class (that is, $v \equiv v'$ holds), then, for any location $\ell$ of $A$, the configurations $(\ell, v)$ and $(\ell, v')$ are untimed bisimilar.

In what follows we shall refer to the equivalence classes induced by $\equiv$ as regions.

Definition 10.14 An $\equiv$-equivalence class $[v]_{\equiv}$ represented by some clock valuation $v$ is called a region.

Each region can be uniquely characterized by a finite collection of clock constraints that it satisfies. For instance, consider the valuation $v$ over two clocks $x, y$ such that $v(x) = \sqrt{2}$ and $v(y) = 1.3$. Assume that both $c_x$ and $c_y$ are equal to 2. Then, each valuation $v'$ that is equivalent to $v$ satisfies the constraint $(1 < y < x < 2)$, and we will use $[1 < y < x < 2]_{\equiv}$ to denote the region $[v]_{\equiv}$.

Exercise 10.15 Check that each valuation $v'$ that is equivalent to the valuation $v$ introduced in the paragraph above does satisfy the constraint $(1 < y < x < 2)$, as claimed.

Example 10.12 Consider a timed automaton with only one clock $x$ such that $c_x = 3$. There are exactly 8 regions consisting of 4 corner points, 3 closed line segments and 1 open line segment, namely

- $[x = 0]_{\equiv}$, $[x = 1]_{\equiv}$, $[x = 2]_{\equiv}$, $[x = 3]_{\equiv}$
- $[0 < x < 1]_{\equiv}$, $[1 < x < 2]_{\equiv}$, $[2 < x < 3]_{\equiv}$
- $[3 < x]_{\equiv}$

Graphically, we can draw a picture like this.

Exercise 10.16 Consider a timed automaton with only one clock $x$, and constraints $x > 0$ and $x \leq 2$. How many regions does this determine? What are they?
Example 10.13 Consider a timed automaton with two clocks $x$ and $y$ such that $c_x = 2$ and $c_y = 1$. There are exactly 28 regions consisting of 6 corner points, 9 closed line segments, 5 open line segments, 4 closed areas and 4 open areas, namely

- $[x = 0, y = 0]_\equiv, [x = 1, y = 0]_\equiv, [x = 2, y = 0]_\equiv, [x = 0, y = 1]_\equiv, [x = 1, y = 1]_\equiv, [x = 2, y = 1]_\equiv$

- $[0 < x < 1, y = 0]_\equiv, [1 < x < 2, y = 0]_\equiv, [0 < x < 1, y = 1]_\equiv, [1 < x < 2, y = 1]_\equiv, [x = 0, 0 < y < 1]_\equiv, [x = 1, 0 < y < 1]_\equiv, [x = 2, 0 < y < 1]_\equiv$

- $[0 < x < 1, 0 < y < 1, x = y]_\equiv, [1 < x < 2, 1 < y < 2, x = y]_\equiv$

- $[2 < x, y = 0]_\equiv, [2 < x, y = 1]_\equiv, [x = 0, 1 < y]_\equiv, [x = 1, 1 < y]_\equiv, [x = 2, 1 < y]_\equiv$

- $[0 < x < 1, 0 < y < 1, x > y]_\equiv, [0 < x < 1, 0 < y < 1, x < y]_\equiv, [1 < x < 2, 0 < y < 1, \frac{x}{y} > \frac{y}{x}]_\equiv, [1 < x < 2, 0 < y < 1, \frac{x}{y} \leq \frac{y}{x}]_\equiv$

- $[2 < x, 0 < y < 1]_\equiv, [0 < x < 1, 1 < y]_\equiv, [1 < x < 2, 1 < y]_\equiv, [2 < x, 1 < y]_\equiv$

This can be graphically depicted as follows.

Exercise 10.17 The regions for a timed automaton with two clocks $x$ and $y$ such that $c_x = 3$ and $c_y = 2$ look as follows.
How many regions are there in the picture?

As indicated by the above examples, each region of a timed automaton $A$ can be uniquely represented by specifying the following items of information:

- for each clock $x$, one constraint from the set
  \[
  \{ x = n \mid n \in \{0, 1, \ldots, c_x\}\} \cup \\
  \{ n < x < n + 1 \mid n \in \{0, 1, \ldots, c_x - 1\}\} \cup \{c_x < x\},
  \]

- for each pair of distinct clocks $x$ and $y$ that, for some $n < c_x$ and $m < c_y$, satisfy constraints of the form $n < x < n + 1$ and $m < y < m + 1$, an indication of whether $\text{frac}(v(x))$ is smaller than or equal to $\text{frac}(v(y))$ or not, for each valuation $v$ in that region.

Exercise 10.18 Assume a timed automaton with a set of clocks $C = \{x, y\}$ and the corresponding constants $c_x$ and $c_y$. Find a general expression which describes the number of regions for the given constants $c_x$ and $c_y$. (Hint: count the number of possible combinations of constraints of the above form.)

We shall now define the fundamental concept of a region graph. The main idea is that every configuration of the form $(\ell, v)$ will be replaced by a so called symbolic state $(\ell, [v])$ in the region graph where $[v]$ is the region represented by $v$. Whenever we have a time elapsing or a standard transition between two configurations, we shall also have a transition between the corresponding symbolic states. This can be formally described as follows.
Definition 10.15 The region graph of a timed automaton $A$ over a set of clocks $C$ and actions $\text{Act}$ is a labelled transition system $T_r(A) = (\text{Proc}, \text{Act} \cup \{\varepsilon\}, \{\Rightarrow\} | a \in \text{Act} \cup \{\varepsilon\})$ where

- $\text{Proc} = \{ (\ell, [v]_{\equiv}) \mid \ell \in L, \ v : C \rightarrow \mathbb{R}_{\geq 0} \}$, i.e., states are symbolic states, and

- $\Rightarrow$ on symbolic states is defined as follows:
  - for each label $a \in \text{Act}$, we have $(\ell, [v]_{\equiv}) \stackrel{a}{\Rightarrow} (\ell', [v']_{\equiv})$ if, and only if, $(\ell, v) \xrightarrow{a} (\ell', v')$, and
  - $(\ell, [v]_{\equiv}) \stackrel{\varepsilon}{\Rightarrow} (\ell, [v']_{\equiv})$ if, and only if, $(\ell, v) \xrightarrow{d} (\ell, v')$, for some $d \in \mathbb{R}_{\geq 0}$.

Exercise 10.19 (Recommended) Prove that the relation $\Rightarrow$ in $T_r(A)$ is reflexive and transitive.

Example 10.14 Consider the following timed automaton $A$.

The regions with assigned numbers look as follows.
A fragment of the region graph reachable from the initial configuration \((\ell_0, 1) = (\ell_0, [x = y = 0]_\equiv)\) is depicted below.

In order to make the picture simple, the reflexive closure of the \(\varepsilon\)-transitions is omitted in the drawing. In particular, there should be also the following \(\varepsilon\)-transitions: \((\ell_0, 1) \xrightarrow{\varepsilon} (\ell_0, 4), (\ell_0, 1) \xrightarrow{\varepsilon} (\ell_0, 18), (\ell_0, 9) \xrightarrow{\varepsilon} (\ell_0, 18), (\ell_1, 2) \xrightarrow{\varepsilon} (\ell_1, 11), (\ell_1, 2) \xrightarrow{\varepsilon} (\ell_1, 18), (\ell_1, 16) \xrightarrow{\varepsilon} (\ell_1, 18)\), and \(\varepsilon\)-loops in every state.

**Exercise 10.20** Construct the region graph of the following timed automaton.

**Theorem 10.4** The region graph \(T_r(A)\) of any timed automaton \(A\) is finite and can be algorithmically constructed. Moreover, for each location \(\ell\) and valuation \(v\), it is the case that \((\ell, v)\) in the untimed transition system \(T_u(A)\) is strongly bisimilar to \((\ell, [v]_\equiv)\) in the transition system \(T_r(A)\).

**Proof:** A proof to the effect that the region graph \(T_r(A)\) of any timed automaton \(A\) is finite and can be algorithmically constructed may be found in the classic reference (Alur and Dill, 1994). That \((\ell, v)\) in the untimed transition system \(T_u(A)\) is strongly bisimilar to \((\ell, [v]_\equiv)\) in the transition system \(T_r(A)\) follows from Theorem 10.3 and from the fact that strong bisimilarity is an equivalence relation (Theorem 3.1(1)).

**Corollary 10.1** Untimed bisimilarity between two timed automata is decidable.
Proof: Theorem 10.4 shows that untimed bisimilarity between two timed automata can be algorithmically reduced to strong bisimilarity between the corresponding region graphs. Since the region graph of an arbitrary timed automaton is finite, the claim follows.

Another important application of region graphs is to probably to the most studied question in the theory and practice of timed automata—namely, the reachability problem.

Given a timed automaton $A$ we write $(\ell, v) \rightarrow (\ell', v')$ whenever

- $(\ell, v) \xrightarrow{a} (\ell', v')$ for some action $a$, or
- $(\ell, v) \xrightarrow{d} (\ell', v')$ for some $d \in \mathbb{R}_{\geq 0}$.

The reachability problem for timed automata is defined as follows. We are given a timed automaton $A = (L, \ell_0, E, I)$ over a set of clocks $C$ and a configuration $(\ell, v)$. The question is to decide whether $(\ell, v)$ is reachable from the initial configuration, i.e., whether $(\ell_0, v_0) \rightarrow^* (\ell, v)$ where $v_0(x) = 0$ for all $x \in C$.

Having the region graph technique at hand, we can now see the validity of the following lemma (for a detailed proof we refer the reader to citexxx XXX Kim XXX which reference to put here?).

**Lemma 10.1** Let $A$ be a timed automaton and $(\ell, v)$ a configuration. It holds that $(\ell_0, v_0) \rightarrow^* (\ell, v)$ in $A$ if and only if $(\ell_0, [v]_\equiv) \Rightarrow^* (\ell, [v]_\equiv)$ in its region graph $T_r(A)$.

A direct corollary of the above lemma is that, as first shown by Alur and Dill in (Alur and Dill, 1994), the reachability problem for timed automata is decidable.

**Corollary 10.2** The reachability problem for timed automata is decidable.

Region graphs provide a finite and elegant abstraction of infinite timed transition systems generated by timed automata, which enables us to prove decidability of, e.g., reachability, timed and untimed bisimilarity, untimed language equivalence and language emptiness (see the next section for references).

On the other hand, region graphs have very large state spaces. The state-space explosion is exponential in the number of clocks and in the maximal constants appearing in the guards. Indeed, as shown by Alur and Dill in (Alur and Dill, 1994), we get the following bound on the size.

**Proposition 10.1** Let $A$ be a timed automaton with $n$ clocks. Let $C$ be the set of clocks in $A$. The number of regions of $A$ is smaller than, or equal to,

$$n! \cdot 2^n \cdot \Pi_{x \in C}(2c_x + 2).$$
Researchers Kim, please add a reference therefore developed a more compact representation of regions by means of so-called zones. A zone is described by a clock constraint \( g \in \mathcal{B}(C) \), and it defines a set of valuations as follows:

\[
\{ v \mid v \models g \}.
\]

In a similar way as we defined a region graph, we can now introduce a so-called zone graph. Essentially all tools that build on the theory of timed automata—for example, Kronos (Bozga, Daws, Maler, Olivero, Tripakis and Yovine, 1998) and UPPAAL (Behrmann et al., 2004)—nowadays use zones in their verification engines. Let us conclude this section by mentioning that zones are often stored in the memory in a data structure called Difference Bound Matrix (Bellman, 1957; Dill, 1989; Yannakakis and Lee, 1993) (or simply DBM). We refer the interested reader to (Dill, 1989; Yannakakis and Lee, 1993) for further details about DBM related issues.

**Exercise 10.21** Show that each zone \( \{ v \mid v \models g \} \) is convex—that is, argue that, for all valuations \( v, v' \) that satisfy \( g \) and each real number \( 0 < \lambda < 1 \), the valuation \( v'' \) defined by

\[
v''(x) = \lambda v(x) + (1 - \lambda) v'(x), \quad \text{for each clock } x,
\]

also satisfies \( g \).

10.6 Overview of the Main Results

Perhaps Kim could provide a more educated summary of the main results here?

**Theorem 10.5 (Alur and Dill (Alur and Dill, 1994))** Timed language equivalence is undecidable.

**Theorem 10.6 (Čeráns (Čeráns, 1993))** Timed bisimilarity for timed automata is decidable in \( \text{EXPTIME} \) (deterministic exponential time).

**Theorem 10.7 (Larsen and Yi (Larsen and Wang, 1997))** Untimed bisimilarity for timed automata is decidable in \( \text{EXPTIME} \) (deterministic exponential time).

**Theorem 10.8 (Alur, Dill'94)** Untimed language equivalence for timed automata is decidable. (Kim XXX complexity and reference???)

**Theorem 10.9 (Alur, Dill)** Reachability for timed automata is decidable in \( \text{PSPACE} \) (polynomial space). (Kim XXX reference)
10.7 Networks of Timed Automata

Many real-life systems consist of a number of independent components running in parallel and communicating whenever necessary. For example, a production line may consist of a number of independent sensors and actuators for single purpose operation that have to synchronize in order to complete the whole production task. Such a composed behaviour, of course, depends also on timing features and we should like to be able to model it by suitably combining descriptions of its components.

Process algebras like CCS and its TCCS extension with timing features provide this possibility by means of the operation of parallel composition. The communication is implemented by one parallel component raising a synchronization request on a particular channel and another component accepting the request on the same channel. Both components can then simultaneously perform the communication transitions and we assume that the duration of the synchronization action is 0 time units—that is, communication is instantaneous. This form of communication is also called hand-shake synchronization. If we want to force the communication on a particular channel, we can further use the restriction operator on that channel as demonstrated earlier in Section 2.2.

In the case of timed automata, our formalism so far enables us to model only a single component. Having in place the inspiration from CCS and TCCS, we shall develop a more general model consisting of a collection of timed automata running in parallel with one another. Such automata may also synchronize with each other. We shall call this kind of system a network of timed automata.

By way of example, consider the following two timed automata with initial locations A and B.

\[ A \]

\[ x \geq 3 \]

\[ a! \]

\[ A' \]

\[ B \]

\[ x \geq 4 \]

\[ a? \]

\[ B' \]

Note that the actions labelling the edges in the two timed automata are now taking two forms. The action \( a! \) means that by performing the transition the process \( A \) wants to synchronize on channel \( a \) with some other process (in our case with the process \( B \)) offering the action \( a? \) in exchange. (By convention, we think of action \( a! \) as standing for an “output on channel \( a \)” , whereas \( a? \) stands for an ‘input on channel \( a \)’.) The processes can then communicate using the hand-shake synchronization. Moreover, all channels are implicitly assumed to be restricted
at the highest level; hence the synchronization is always forced in networks of timed automata. In fact, we can describe the above behaviour in terms of TCCS as \((A \mid B) \setminus \{a\}\) where \(A \equiv \varepsilon(3).a.A'\) and \(B \equiv \varepsilon(4).a.B'\).

Let us now consider another example, namely the light switch and the fast user from Section 9.4. The TCCS definition of the light switch can be directly rewritten into the timed automaton in the upper part of the picture. The fast user (in the lower part of the picture) is performing the action press! exactly every 3 time units.

Note that we have “stretched” the time by multiplying all constants by 10 in order to have only integer constants in our model (see Remark 10.2). We are also using a new clock \(y\) instead of \(x\) in the automaton modelling the fast user. This is because we want to avoid a clash among the parallel components (should the user be using the same clock \(x\), its reset would also influence the behaviour of the light switch).

The intended behaviour of the above network is as follows. The fast user presses the switch at time 0 and the upper automaton changes its location to Light. At the same moment the user enters the location \(U'\). After another 3 time units the user presses again the switch and the automata enter simultaneously the locations Bright and \(U'\), respectively. After yet another 3 time units the user presses the switch again and both automata synchronize and enter the locations Off and \(U'\), respectively. The system behaviour then continues in a similar manner. Not surprisingly, the semantics of the network can be given via a timed labelled transition system. The structure of the states is now richer, as it contains pairs of locations of the respective timed automata together with information on the current value of the clocks in both automata, and, as in the case of CCS, the communication appears for an external observer as the action \(\tau\). A fragment of the system looks as follows, where we have omitted the values of the clocks for the sake of readability.
Exercise 10.22 Add the information on the clock values to the states in the fragment of the timed labelled transition system above.

We shall now proceed to formalize the notion of a network of timed automata. Assume that our set of actions consists of a finite set of channel names \(\text{Chan}\) (followed by the symbol ‘!’ or ‘?’, which indicates whether the action uses the channel for output or input, respectively) and of a finite set \(N\) of ordinary action names, formally
\[
\text{Act} = \{c! \mid c \in \text{Chan}\} \cup \{c? \mid c \in \text{Chan}\} \cup N.
\]

By analogy with CCS, we say that the actions \(c!\) and \(c?\) are complementary. We shall use \(\alpha, \beta\) to range over \(\text{Act}\).

Definition 10.16 Let \(n\) be a positive integer and, for each \(i \in \{1, \ldots, n\}\), let
\[
A_i = (L_i, \ell_i^0, E_i, I_i)
\]
be timed automata over a set of clocks \(C\) and a set of actions \(\text{Act}\). We call their composition \(A = A_1 \mid A_2 \mid \cdots \mid A_n\) a network of timed automata with \(n\) parallel components.

As we already mentioned above, the semantics to the parallel composition \(A = A_1 \mid A_2 \mid \cdots \mid A_n\) will be given by means of a timed labelled transition system. The following definition formalizes the behaviour of a network of timed automata.

Definition 10.17 Let \(A = A_1 \mid A_2 \mid \cdots \mid A_n\), where \(A_i = (L_i, \ell_i^0, E_i, I_i)\) for each \(i \in \{1, \ldots, n\}\), be a network of timed automata over a set of clocks \(C\) and actions \(\text{Act} = \{c! \mid c \in \text{Chan}\} \cup \{c? \mid c \in \text{Chan}\} \cup N\). We define the timed labelled transition system \(T(A)\) generated by the network \(A\) as \(T(A) = (\text{Proc}, \text{Lab}, \{a \rightarrow \mid a \in \text{Lab}\})\) where

- \(\text{Proc} = \{ (\ell_1, \ell_2, \ldots, \ell_n, v) \mid (\ell_1, \ell_2, \ldots, \ell_n, v) \in L_1 \times L_2 \times \cdots \times L_n \times (C \rightarrow \mathbb{R}_{\geq 0}) \text{ and } v = \bigwedge_{i \in \{1, \ldots, n\}} I_i(\ell_i) \}\)——i.e., states are of the form \((\ell_1, \ldots, \ell_n, v)\), where each \(\ell_i\) is a location in the component (timed automaton) \(A_i\) and \(v\) is a valuation over the set of clocks \(C\) that satisfies the invariants of all locations \(\ell_i\) present in the state,
\textbf{Lab} = \mathcal{N} \cup \{\tau\} \cup \mathbb{R}_{\geq 0} \text{ is the set of labels, and}

the transition relation is defined as follows:

1. \( (\ell_1, \ldots, \ell_i, \ldots, \ell_n, v) \xrightarrow{a} (\ell_1, \ldots, \ell'_i, \ldots, \ell_n, v') \) if \( a \in \mathcal{N} \) and there is an edge \( (\ell_i g_i, a, r_i, \ell'_i) \in E_i \) in the \( i \)'th component automaton such that
   \begin{itemize}
   \item \( v \models g, v' = v[r] \) and
   \item \( v' = I_i(\ell'_i) \land \bigwedge_{k \neq i} I_k(\ell_k) \);
   \end{itemize}

2. \( (\ell_1, \ldots, \ell_i, \ldots, \ell_j, \ldots, \ell_n, v) \xrightarrow{\tau} (\ell_1, \ldots, \ell'_i, \ldots, \ell'_j, \ldots, \ell_n, v') \) if \( i \neq j \) and there are edges \( (\ell_i g_i, \alpha, r_i, \ell'_i) \in E_i \) and \( (\ell_j g_j, \beta, r_j, \ell'_j) \in E_j \) such that
   \begin{itemize}
   \item \( \alpha \) and \( \beta \) are complementary,
   \item \( v \models g_i \land g_j, v' = v[r_i \cup r_j] \) and
   \item \( v' = I_i(\ell'_i) \land I_j(\ell'_j) \land \bigwedge_{k \neq i,j} I_k(\ell_k) \);
   \end{itemize}

3. \( (\ell_1, \ldots, \ell_n, v) \xrightarrow{d} (\ell_1, \ldots, \ell_n, v + d) \) for all \( d \in \mathbb{R}_{\geq 0} \) such that the \( v + d' \models \bigwedge_{i \in \{1, \ldots, n\}} I_i(\ell_i) \) for each real number \( d' \) in the interval \([0, d]\).

Let \( v_0 \) denote the valuation such that \( v_0(x) = 0 \) for all \( x \in \mathcal{C} \). If \( v_0 \) satisfies the invariants of all the initial locations \( \ell'_0 \), we shall call \( (\ell'_1, \ell'_1, \ldots, \ell'_n, v_0) \) the initial state (or initial configuration) of \( T(A) \).

Even though the definition of the transition relation for networks of timed automata might look more technical than usual, the intuition behind it is easy to understand. The first part says that any component can make an independent move as long as it happens under an ordinary action from \( \mathcal{N} \), the move is enabled by the guard in the corresponding component, all clocks that are reset in the component are reset also in the composed state and we do not violate any invariant. The second part of the definition is for the situation when two components are willing to perform an input and output action on a particular channel. The guards have to be satisfied as before, the collection of reset clocks is the union of the clocks that are reset in both automata and the invariants have to be satisfied as before. Moreover, in analogy with CCS and TCCS, this transition is supposed to be internal and is visible only under the silent action \( \tau \). Finally, the last part of the definition allows arbitrary time delays as long as the invariants of all components are not violated as time progresses.
10.7. NETWORKS OF TIMED AUTOMATA

Exercise 10.23 (Mandatory) Use the formal definition above to draw a fragment of the timed labelled transition system for the network of timed automata consisting of the light switch and the fast user. Determine all of the transitions in the timed labelled transition system for the network of timed automata on page 217.

Networks of timed automata are a very useful extension of the model of timed automata, and nowadays essentially all available tools for verification of timed automata use the above defined network (or a variant of it) to model real-life real-time systems. Among such tools, let us mention at least UPPAAL (Behrmann et al., 2004), KRONOS (Bozga et al., 1998) and CMC (Laroussinie and Larsen, 1998), and we refer the reader to the relevant literature for further details. The notions of timed and untimed bisimilarity as well as of region graphs, zones and the reachability algorithms based on them can be directly transferred to the network scenarios. The advantages of networks of timed automata for modelling of systems are accompanied also by a collection of pleasing theoretical results. For example, the reachability problems for a single timed automaton as well as for a network are both PSPACE-complete (the reader is referred to (Aceto and Laroussinie, 2002) for further details on the complexity of verification problems for networks of timed automata), so—at least from the theoretical complexity point of view—the increase in the complexity of verification problems is not as dramatic as the reader might have thought at the first glance. The theoretically-minded reader might wish to compare this situation to the one that arises in the verification of untimed concurrent systems. In that setting, the modelling power that is gained moving from single automata to networks of automata has a price: the complexity of verification problems increases dramatically over networks. You can find a discussion of this issue, and pointers to further reading in the paper (Aceto and Laroussinie, 2002).

Exercise 10.24 Let \( A = A_1 \mid A_2 \mid \cdots \mid A_n \), where \( A_i = (L_i, l_{i0}, E_i, I_i) \) for each \( i \in \{1, \ldots, n\} \), be a network of timed automata over a set of clocks \( C \) and actions \( \text{Act} = \{c! \mid c \in \text{Chan}\} \cup \{c? \mid c \in \text{Chan}\} \cup N \). Can you define a timed automaton \( B \) such that \( T(A) \) and \( T(B) \) are isomorphic? How many locations does \( B \) have if each timed automaton \( A_i \) has, say, ten nodes?

Exercise 10.25 Implement the network of timed automata consisting of the fast user and the light switch in the verification tool UPPAAL available at


Simulate the behaviour of the system using the tool, and use the tool to check whether this network contains deadlocks.
Chapter 11

Hennessy-Milner Logic with Time

In Sections 10.3–10.4, we have introduced some notions of behavioural equivalence over real-time systems specified by means of timed automata. These equivalences are based on various adaptations to the timed setting of the classic notions of trace equivalence and strong bisimilarity over labelled transition systems—as presented in Sections 3.2–3.3 of this book—and may be used to perform implementation verification for real-time systems. This is useful because, at least in principle, a formalism like that of timed automata can be used to describe both actual systems and their specifications, and, as we have seen in Section 10.6, these notions of behavioural equivalence are decidable over (networks) of time automata with the notable exception of timed trace equivalence (see Theorem 10.5).

However, as we already noted in the setting of modelling and verification for classic, untimed reactive systems, when establishing the correctness of our system with respect to a specification using the methodology of implementation verification, we are somehow forced to specify the overall behaviour of the system under consideration. In a real-time setting, this often means that our specifications need to consider many details pertaining to the timing behaviour of the implementation under analysis. This may lead to overly complex and subtle specifications. Moreover, sometimes we are only interested in specifying the expected behaviour of the system in certain specific circumstances.

Suppose, for instance, that all we want to know about our system is that each $a$-labelled transition is followed by a $b$-labelled transition within 2 time units. Expressing this requirement, and similar ones, in terms of observational equivalence is rather unnatural. Indeed, this aspect of the behaviour of a system seems best checked by exploring the state space of the (network of) timed automata describ-
ing the system. In the setting of classic reactive systems, we saw in Section 5 that
the so-called Hennessy-Milner Logic (HML) is a suitable formalism for specifying
properties of reactive systems. Indeed, not only does HML allow us to express
natural requirements on the behaviour of reactive systems, but, as originally shown
by Hennessy and Milner in one of the most satisfying results in the theory of con-
current processes, it captures precisely all of the behavioural properties of reactive
systems that are relevant with respect to bisimilarity—recall Theorem 5.1.

Since HML is a convenient formalism for the description of behavioural prop-
erties of reactive systems modelled semantically as labelled transition systems, and
the semantics of timed automata is given in terms of timed labelled transition sys-
tems (see Definition 9.1), it is natural to try to define a notion of HML for real-time
systems. In what follows, our aim will be to motivate and introduce this variation
on HML, argue by means of examples that it allows us to specify properties of real-
time systems modelled as timed labelled transition systems or timed automata, and
study its relationship with timed bisimilarity. As we shall see, the overall collection
of results that we shall obtain mirrors, and is just as satisfying as, that presented in
Section 5.

11.1 Basic Logic

As you might recall from our developments in Section 5, Hennessy-Milner logic
is a modal logic that is obtained by adding to the syntax of boolean logic two
modal operators that allow us to express properties of reactive systems (modelled
as labelled transition systems) that relate to the effect that performing actions has
on their behaviour. More specifically, we recall, for the sake of clarity, that

- a process satisfies a formula of the form $\langle a \rangle F$ for some $a \in \text{Act}$ iff it affords
  an $a$-labelled transition leading to a state satisfying $F$, and that

- a process satisfies a formula of the form $[a] F$ for some $a \in \text{Act}$ iff all of its
  $a$-labelled transitions lead to a state satisfying $F$.

Hence, formulae of the form $\langle a \rangle F$ express “possible behaviour” of processes,
whereas formulae of the form $[a] F$ describe their “necessary behaviour”—that is,
properties that must hold for each of their $a$-derivatives.

Timed labelled transition systems, as presented in Definition 9.1, are just ordi-
nary labelled transition systems that have transitions whose labels can be also time
delays. If we are to follow the lead of Hennessy and Milner in defining a modal
logic for describing their properties, it seems therefore reasonable to augment the
syntax of HML with two new modalities that can be used to express possible and
11.1. BASIC LOGIC

necessary behaviour of systems as time progresses. Following the notation introduced by Laroussinie, Larsen and Weise in (Laroussinie et al., 1995), these two new “time modalities” will be denoted by \( \exists \) and \( \forall \), respectively.

By analogy with the two classic action modalities, we expect that

- A process satisfies a formula of the form \( \exists F \) iff it can delay some amount of time thereby reaching a state satisfying \( F \), and that
- A process satisfies a formula of the form \( \forall F \) iff no matter how long it delays it will always reach a state satisfying \( F \).

For instance, we would expect that the initial states of the following two timed automata (introduced in Example 10.9)

\[
\begin{align*}
A & \quad x \leq 1 \\
\quad & \quad a \\
\quad & \quad x := 0 \\
B & \\
\end{align*}
\]

\[
\begin{align*}
A' & \quad x \leq 2 \\
\quad & \quad a \\
\quad & \quad x := 0 \\
B' & \\
\end{align*}
\]

both satisfy the formula \( \exists (a) \# \) (as both those states can perform action \( a \) immediately), but that neither of them satisfies the formula \( \forall (a) \# \) (as both of them can delay 2.1 timed units, say, and reach a state where the \( a \)-action is no longer possible).

However, the mere addition of these two action modalities to HML does not suffice to express all of the timing properties of systems that we should like to describe. For instance, we previously mentioned the property

“each \( a \)-labelled transition is followed by a \( b \)-labelled transition within 2 time units,”

as an example of a property that we wish to express using our variant on HML. A brief examination of the behaviour of the two timed automata above leads us to expect that the automaton on the right enjoys the following property:

“an \( a \)-labelled transition is possible after a delay of 2 time units.”

On the other hand, the initial state of the automaton on the left should not afford this property.

Both of the aforementioned properties make explicit reference to time delays, and it seems therefore reasonable to extend our variant on HML with some way of expressing “quantitative real-time constraints.” The design decisions taken by
Alur and Dill in their development of timed automata provide us with suitable inspiration here. Timed automata use clock resets and guards to specify real-time constraints on the behaviour of real-time systems. For instance, we might specify that a $b$-action should follow an $a$-action within 2 time units by resetting a clock $x$ upon the performance of an $a$-labelled edge leading to a location $\ell$, and adjoining a guard like $x \leq 2$ to $b$-labelled edges that emanate from $\ell$. We shall therefore augment our variant on HML with clock constraints (whose syntax will take the form given in Definition 10.1), and clock resets. A formula of the form

$$x \in F$$

will intuitively say that a state in a timed labelled transition system should satisfy $F$ after setting the value of $x$ to zero. For instance, the formula

$$y \in \exists(y > 1 \land \langle a \rangle \#)$$

states intuitively that it is possible to delay more than one time unit thereby reaching a state in which an $a$-labelled transition is possible. We shall for simplicity assume that the clocks used in the formulae are disjoint with the clocks that appear in timed automata (should they be used to generate the underlying timed transition systems).

We are now ready to present the syntax of Hennessy-Milner logic with time.

**Definition 11.1** The set of Hennessy-Milner formulae with time (from now on referred to as $\mathcal{M}_t$) over a set of actions $\mathcal{A}$ and a set of formula clocks $D$ is given by the following abstract syntax:

$$F ::= \# | \texttt{ff} | F \land G | F \lor G | \langle a \rangle F | \left[ a \right] F | \exists F | \forall F | x \in F | g$$

where $a \in \mathcal{A}$, $x \in D$ and $g \in \mathcal{B}(D)$.

In writing formulae, we shall sometimes use the same abbreviations introduced in Definition 5.1. In particular, we recall, for the sake of clarity, that if $A = \{a_1, \ldots, a_n\} \subseteq \mathcal{A}$ ($n \geq 0$), we use the abbreviation $\langle A \rangle F$ for the formula $\langle a_1 \rangle F \lor \cdots \lor \langle a_n \rangle F$ and $\left[ A \right] F$ for the formula $[a_1] F \land \cdots \land [a_n] F$. (If $A = \emptyset$, then $\langle A \rangle F = \text{ff}$ and $\left[ A \right] F = \bot$.)

We are interested in using the above logic to describe properties of states in a TLTS over the set of actions $\mathcal{A}$. The semantics of a formula in the language $\mathcal{M}_t$ is given by characterizing the collection of states that satisfy it. We have already presented the intuitive meaning of all of the constructs in the logic; however, there is still a subtlety that needs to be dealt with before we can present the formal definition of the semantics of our variant on HML with time.
Clock constraints are first-class formulae in our language, and we wish to be able to determine whether a state in a TLTS satisfies a clock constraint. But when does a state satisfy the constraint \( y > 1 \), say? In our example formula

\[ y \mathrel{\text{\#}} \exists (y > 1 \land \langle a \rangle \#) \]

we used this constraint as part of the formula to specify that we wish that a state from which an \( a \)-labelled transition is possible be reached after delaying more than one time unit.

The answer to the question above is classic in logic: in order to determine whether a state satisfies a guard we need to make reference to a valuation for the clocks in the set \( D \). The valuation will be used to check whether clock constraints are met or not.

The semantics of formulae is given with respect to a given timed labelled transition system

\[ (\text{Proc}, \text{Lab}, \{ \rightarrow | a \in \text{Lab} \}) \]

An extended state over \( \text{Proc} \) is a pair \((p, u)\), where \( p \) is a state in \( \text{Proc} \), and \( u \) is a time assignment for \( D \)—that is, a mapping \( D \to \mathbb{R}_{\geq 0} \). The set of extended states over \( \text{Proc} \) will be noted by \( \mathcal{E}S(\text{Proc}) \). We shall use \([ F ]\), where \( F \) is a formula in \( \mathcal{M}_t \), to denote the set of extended states over \( \text{Proc} \) that satisfy \( F \). This we now proceed to define formally.

**Definition 11.2** We define \([ F ] \subseteq \mathcal{E}S(\text{Proc})\) for \( F \in \mathcal{M}_t \) by:

\[
\begin{align*}
[\#] &= \mathcal{E}S(\text{Proc}) \\
[ff] &= \emptyset \\
[F \land G] &= [F] \cap [G] \\
[\forall F] &= \langle \varepsilon \cdot \rangle [F] \\
[\exists F] &= \langle \varepsilon \cdot \rangle [F] \\
[x \mathrel{\text{\#}} F] &= \{(p, u) \mid (p, u[x \mapsto 0]) \in [F]\} \\
[g] &= \{(p, u) \mid p \in \text{Proc}, u \models g\}
\end{align*}
\]

where we use the set operators

\[ \langle a \cdot \rangle, [a \cdot ], \langle \varepsilon \cdot \rangle, [\varepsilon \cdot ] : \mathcal{P}(\mathcal{E}S(\text{Proc})) \to \mathcal{P}(\mathcal{E}S(\text{Proc})) \]
defined by

\[
\begin{align*}
\langle a \rangle S &= \{ (p, u) \in ES(Proc) \mid \exists p'. p \xrightarrow{a} p' \text{ and } (p', u) \in S \}, \\
\lnot a S &= \{ (p, u) \in ES(Proc) \mid \forall p'. p \xrightarrow{a} p' \text{ implies } (p', u) \in S \}, \\
\langle \varepsilon \rangle S &= \{ (p, u) \in ES(Proc) \mid \exists d \in \mathbb{R}_{\geq 0}. \exists p' \in Proc. p \xrightarrow{d} p' \text{ and } (p', u + d) \in S \}, \\
\lnot \varepsilon S &= \{ (p, u) \in ES(Proc) \mid \forall d \in \mathbb{R}_{\geq 0}. \forall p' \in Proc. p \xrightarrow{d} p' \text{ implies } (p', u + d) \in S \}.
\end{align*}
\]

We write \((p, u) \models F\) iff \((p, u) \in [F]\).

Two formulae are equivalent if, and only if, they are satisfied by the same extended states in every timed labelled transition system.

**Definition 11.3** A state \(p\) in a timed labelled transition system satisfies a formula \(F\) (written \(p \models F\)) iff \((p, u_0) \models F\) where \(u_0\) is a clock valuation mapping each formula clock to zero.

Note that the above definitions apply equally well to the timed labelled transition system \(T(A)\) generated from a time automaton \(A\). For this TLTS, however, extended states take the (notationally slightly unpleasant) form \(((\ell, v), u)\), where \(v\) is a valuation for the set of clocks \(C\) in \(A\), and \(u\) is a valuation for the set of clocks \(D\) used in writing the formulae in \(\mathcal{M}_t\). From now on, we shall always tacitly assume that the set of clocks used in formulae is disjoint from that used in timed automata. This means that reset operations on clocks from one of these sets will not have any effect on clocks in the other. So, when specialized over a TLTS of the form \(T(A)\), the semantics of a formula of the form \(x \xrightarrow{\in} F\) becomes

\[
[x \xrightarrow{\in} F] = \{(\ell, v), u) \mid ((\ell, v), u[x \mapsto 0]) \in [F]\}\ .
\] (11.1)

Note how the reset operation only applies to clock \(x\), whereas the values of each clock in the automaton \(A\) remain unchanged because we are asking that

\[
((\ell, v), u[x \mapsto 0]) \in [F]
\] .

**Definition 11.4** A timed automaton \(A\) satisfies a formula \(F \in \mathcal{M}_t\) iff

\[
((\ell_0, v_0), u_0) \models F ,
\]

where \(\ell_0\) is the initial location in \(A\), and \(v_0, u_0\) are clock valuations mapping each clock variable in the automaton and in the formula to zero, respectively.
To understand better the above definition of the semantics of formulae in $\mathcal{M}_t$, it is instructive to use the formal definition of the semantics of $\mathcal{M}_t$ to establish that the initial state of the timed automaton

![Timed Automaton Diagram]

satisfies the formula $y \in \exists (3 \geq y > 1 \land \langle a \rangle \#)$. To see that this holds, we use (11.1) to derive that

$$\models (\exists (3 \geq y > 1 \land \langle a \rangle \#)).$$

Now observe that to establish that

$$\models (\exists (3 \geq y > 1 \land \langle a \rangle \#))$$

it suffices only to find a $d \in \mathbb{R}_{\geq 0}$ such that

$$\models (3 \geq y > 1 \land \langle a \rangle \#).$$

It is easy to find such a $d$. In fact, each $d$ in the interval $(1, 2]$ would do because if $d$ lies in that interval then $[y = d] \models 3 \geq y > 1$ and $(A', [x = d]) \xrightarrow{a} (B', [x = 0])$ both hold.

As you might have already noticed, the above reasoning does in fact show that any extended state of the form $((A', [x = 0]), [y = d])$ satisfies the formula $y \in \exists (3 \geq y > 1 \land \langle a \rangle \#)$, regardless of the value of $d$. This is because the use of the formula clock $y$ in the clock constraint $3 \geq y > 1$ is within the scope of a $y \in \exists$ construct. We call formulae in which each occurrence of a formula clock $z$ in a clock constraint is within the scope of a $z \in \exists$ construct closed. For example, the formula $y \in \exists \exists y = 1$ is closed, whereas $y = 1$ and $(y \in \exists \exists y = 1) \land (y \leq 2)$ are not.

If $F$ is a closed formula, then the collection of extended states satisfying it is independent of the valuation $u$ for the formula clocks. This means that, if $F$ is closed, for each state $p$ in a TLTS and valuations $u, u'$ for the formula clocks, we have that

$$(p, u) \models F \iff (p, u') \models F.$$
Therefore, when $F$ is closed it makes sense to speak of a state $s$ satisfying $F$, and we shall tacitly do so in what follows. For a timed automaton $A$ and closed formula $F$, the suggestive shorthand $A \models F$ will be used in lieu of $((\ell_0, v_0), u_0) \models F$. In what follows, whenever we say that a state in a TLTS satisfies a formula, we shall always assume that the formula is closed unless specified otherwise.

**Exercise 11.1** Prove that, as claimed above, if $F$ is a closed formula, then the collection of extended states satisfying it is independent of the valuation $u$ for the formula clocks. Does this hold for arbitrary formulae in $\mathcal{M}_t$? ♦

Let us now try to use the logic $\mathcal{M}_t$ to express formally the following property of a state in a TLTS:

“each $a$-labelled transition is followed by a $b$-labelled transition within 2 time units.”

We can express that a $b$-labelled transition is available within 2 time units by means of the closed formula

$$y \in \exists (y \leq 2 \land \langle b \rangle t t) .$$

Indeed, you should be able to convince yourselves that a state $s$ satisfies the above formula if, and only if, $s \xrightarrow{d} s' \xrightarrow{b}$ for some state $s'$ and real number $d$ in the interval $[0, 2]$. (Do so!) All we need to do now to express the desired property is to realize that the natural language requirement “each $a$-labelled transition” can be expressed in terms of the $[a]$-operator of HML. The resulting formula is therefore

$$[a](y \in \exists (y \leq 2 \land \langle b \rangle t t)) .$$

As a further example, consider the following timed automaton

$$x \leq 2 \quad x \leftarrow_{a}^{0} x \leq 1$$

We already argued in Example 10.5 that this timed automaton can delay two units of time thereby reaching a state in which no $a$-labelled transition is possible. This can be expressed in the language $\mathcal{M}_t$ by means of the formula

$$y \in \exists (y = 2 \land [a] ff) .$$

You should also be able to argue that the timed automaton above also satisfies the formula $\exists [a] ff$. (Do so!) Does it also satisfy the formula

$$[a](y \in \exists (y = 1 \land \langle a \rangle t t)) ?$$
Excercise 11.2 Use the logic $M_t$ to formulate properties of the timed automata in Example 10.9, and argue that the automata have (respectively, do not) have those properties using the semantics of the logic $M_t$. Can you give examples of properties that both timed automata afford?

Excercise 11.3

1. Consider the formulae $y \leq y = 0$ and $y \leq y > 0$. Can you offer equivalent formulations of the properties described by these formulae?

2. Argue that the formulae $\exists F$ and $\exists F$ are equivalent for each formula $F$. Are the formulae $\forall F$ and $\forall F$ also equivalent?

3. Show that $\exists (e \cdot \varepsilon \cdot E S (\text{Proc}))$ and $\exists (e \cdot \varepsilon \cdot E S (\text{Proc}))$ are both equal to $E S (\text{Proc})$. What are the formulae $\forall t$ and $\forall t$ equivalent to?

4. Argue that $(p, u) \models \forall (a) t$ iff $(p, u) \not\models \exists[a] t t$.

5. Prove that formulae $x \leq (y \leq (y \leq \exists F))$ and $y \leq (x \leq (y \leq \exists F))$ are equivalent for any formula $F$. Are the following two formulae $x \leq (y \leq (y \leq \exists F))$ and $y \leq (x \leq (y \leq \exists F))$ also equivalent? If yes, prove it—if not, give a counter example.

As we did on page 108 for standard Hennessy-Milner logic, it is sometimes useful to have an alternative characterization of the satisfaction relation $\models$ presented in Definition 11.2. This can be obtained by defining the binary relation $\models$ relating extended states to formulae by structural induction on formulae as follows.

- $(p, u) \models \#$ for each $(p, u)$
- $(p, u) \models \# F$ for no $(p, u)$
- $(p, u) \models F \land G$ iff $(p, u) \models F$ and $(p, u) \models G$
- $(p, u) \models F \lor G$ iff $(p, u) \models F$ or $(p, u) \models G$
- $(p, u) \models \langle a \rangle F$ iff $p \overset{a}{\rightarrow} p'$ for some $p'$ such that $(p', u) \models F$
- $(p, u) \models \exists F$ iff $p \overset{d}{\rightarrow} p'$ for some $p'$ and $d \in \mathbb{R}_{\geq 0}$ such that $(p', u + d) \models F$
- $(p, u) \models \forall F$ iff $(p', u + d) \models F$ for each $d \in \mathbb{R}_{\geq 0}$ and $p'$ such that $p \overset{d}{\rightarrow} p'$
- $(p, u) \models y \leq F$ iff $(p, u[y \rightarrow 0]) \models F$
• \((p, u) \models g \iff u \models g\)

**Exercise 11.4** Show that the above definition of the satisfaction relation is indeed equivalent to that given in Definition 11.2. [Hint: Use induction on the structure of formulae.]

Note that, as was the case with classic Hennessy-Milner logic (see page 109), logical negation is not one of the constructs in the abstract syntax for \(\mathcal{M}_t\). However, on page 109 we argued that the language \(\mathcal{M}\) is closed under negation, in the sense that, for each formula \(F \in \mathcal{M}\), there is a formula \(F^c \in \mathcal{M}\) that is equivalent to the negation of \(F\). This result carries over to the setting of \(\mathcal{M}_t\). The formula \(F^c\) is defined by structural induction on \(F\) by extending the clauses dealing with the constructs of classic Hennessy-Milner logic on page 109 as follows.

\[
\begin{align*}
(\exists F)^c &= \forall F^c \\
(\forall F)^c &= \exists F^c \\
(y \nleq n)^c &= y > n \\
(y < n)^c &= y \geq n \\
(y = n)^c &= (y < n) \lor (y > n) \\
(y > n)^c &= y \leq n \\
(y \geq n)^c &= y < n
\end{align*}
\]

Note that the “negation” of the clock constraint \(y = n\) is not itself a clock constraint, but rather a formula in \(\mathcal{M}_t\). This is inevitable because, as we saw in Exercise 10.3, the negation of \(y = n\) cannot be expressed as a clock constraint. (This is a good time for you to go back and solve that exercise if you have not done so already!)

**Example 11.1** The negation of the formula \(y \nleq n \exists (y = 2 \land \langle a \rangle t t)\) is the formula \(y \nleq n \forall (y < 2 \lor y > 2 \lor \langle a \rangle f f)\).  

**Exercise 11.5** Negate the formula \(\forall [a] ff \lor x \nleq n \exists (x = 1 \land \langle a \rangle t t)\).

**Proposition 11.1** Let \((\text{Proc}, \text{Lab}, \{a \rightarrow | a \in \text{Lab}\})\) be a timed labelled transition system. Then, for every formula \(F \in \mathcal{M}_t\), it holds that \([F^c] = \mathcal{E}S(\text{Proc}) \setminus [F]\).

**Proof:** The proposition can be proven by structural induction on \(F\). The details are left as an exercise to the reader.

**Exercise 11.6**
11.2. TIMED BISIMILARITY VS. HENNESSY-MILNER LOGIC WITH TIME


2. Prove, furthermore, that \((F^c)^c = F\) for every formula \(F \in \mathcal{M}_t\). [Hint: Use structural induction on \(F\).]

As a consequence of Proposition 11.1, we have that, for each extended state \((p, u)\) and formula \(F\), exactly one of \((p, u) \models F\) and \((p, u) \models F^c\) holds. In fact, each extended state is exclusively contained either in \([F]\) or in \([F^c]\).

Exercise 11.7 (Recommended) Another natural way to introduce time delay operators \(\exists\) and \(\forall\) which can specify particular time durations would be to associate these operators with time intervals. For example the formula \(\exists_{[3,5]} F\) would mean that it is possible to perform a time delay greater or equal to 3 but strictly less than 5 time units such that the formula \(F\) holds afterwards. Similarly \(\forall_{(2,7]} F\) stands for the fact that after all possible time delays, strictly between 2 and 7 time units, the formula \(F\) must hold.

1. Define formally the syntax and semantics of the above mentioned variant of HML with time.

2. Prove that for any such formula one can construct an equivalent formula from \(\mathcal{M}_t\). [Hint: use structural induction.]

11.2 Timed Bisimilarity vs. Hennessy-Milner Logic with Time

In Exercise 11.2, you were asked, amongst other things, to find properties distinguishing the two timed automata in Example 10.9. Those automata are not timed bisimilar.

Consider, as another example, the two timed automata

\[
\begin{align*}
\bigcirc \xrightarrow{a} x &\leq 1 \\
\bigcirc \xrightarrow{a} x &\leq 1
\end{align*}
\]

whose behaviour we analyzed in Figure 10.2. Based on that analysis, you should be able to convince yourselves easily that these two timed automata are not timed
bisimilar. A formula in the language \( \mathcal{M}_t \) that distinguishes these two timed automata is \( y \in \mathbb{R} \exists (y > 1) \). In fact, the timed automaton on the left-hand side satisfies this formula, but the one on the right-hand side does not because it cannot delay for more than one time unit.

**Exercise 11.8** Prove the above claim formally using the semantics of formulae. ♦

Again, faced with two timed automata that are not timed bisimilar, we have been able to find a formula in the logic \( \mathcal{M}_t \) that distinguishes them, in the sense that one timed automaton satisfies it, but the other does not. Is this true in general? And what can we say about two timed automata that satisfy precisely the same formulae in \( \mathcal{M}_t \)? Are they guaranteed to be timed bisimilar?

In Section 5, we saw that classic HML characterizes bisimilarity over image-finite processes—see Definition 5.3 for the formal definition of this class of processes. This was the import of an elegant characterization theorem due to Hennessy and Milner (Theorem 5.1). The acid test for our development of the language \( \mathcal{M}_t \) is whether a similar characterization theorem holds for timed bisimilarity. We shall now proceed to show that this is indeed the case—at least if we use timed automata as our model for real-time systems.

We first show that, in the technical sense stated in the following theorem, two timed bisimilar states in an arbitrary timed labelled transition system satisfy the same formulae in \( \mathcal{M}_t \).

**Theorem 11.1** Let \((\text{Proc}, \text{Lab}, \{a \to | a \in \text{Lab}\})\) be a timed labelled transition system. Assume that \(p, q\) are timed bisimilar states in \(\text{Proc}\). Let \(u\) be a clock valuation for the formula clocks in \(D\). Then the extended states \((p, u)\) and \((q, u)\) satisfy exactly the same formulae (both closed and opened) in \(\mathcal{M}_t\).

**Proof:** Assume that \(p, q\) are timed bisimilar states in \(\text{Proc}\). Let \(u\) be a clock valuation for the formula clocks in \(D\). Assume that \((p, u) \models F\) for some formula \(F \in \mathcal{M}_t\). Using structural induction on \(F\), we prove that \((q, u) \models F\). By symmetry, this is enough to establish that \((p, u)\) and \((q, u)\) satisfy the same formulae in \(\mathcal{M}_t\).

The proof proceeds by a case analysis on the form of \(F\). We only present the details for the case \(F = \forall G\) for some formula \(G\). Our inductive hypothesis is that, for all states \(r\) and \(s\), if \(r\) and \(s\) are timed bisimilar and \((r, u') \models G\) for some valuation \(u'\) of the formula clocks, then \((s, u') \models G\). Using this hypothesis, we shall prove that \((q, u) \models \forall G\). To this end, assume that \(q \xrightarrow{d} q'\) for some state \(q'\) and \(d \in \mathbb{R}_{\geq 0}\). We wish to show that \((q', u + d) \models G\). Now, since \(p\) and \(q\) are timed bisimilar and \(q \xrightarrow{d} q'\), there is a process \(p'\) such that \(p \xrightarrow{d} p'\) and \(p'\) is timed bisimilar to \(q'\). (Why?) By our assumption that \((p, u) \models \forall G\), we have that \((p', u + d) \models G\).
CHAPTER 11.2. TIMED BISIMILARITY VS. LOGIC

The inductive hypothesis yields that \((q', u + d) \models G\). Since \(q'\) and \(d\) were arbitrary, and we may conclude that \((q, u) \models \forall G\), which was to be shown. \(\square\)

By instantiating the above result to the TLTSs that give semantics to timed automata, we get the following result.

**Corollary 11.1** Let \(A\) and \(A'\) be timed bisimilar timed automata. Then \(A\) and \(A'\) satisfy exactly the same formulae in \(\mathcal{M}_t\).

**Exercise 11.9** Assume that \(p, q\) are timed bisimilar states in a TLTS. Suppose, furthermore, that each formula in \(\mathcal{M}_t\) satisfied by \(p\) is also satisfied by \(q\). Prove that \(p\) and \(q\) satisfy the same formulae in \(\mathcal{M}_t\). ♦

In the setting of image-finite labelled transition systems, Theorem 5.1 tells us that two states that satisfy the same formulae in Hennessy-Milner logic are bisimilar. However, the converse of Theorem 11.1 does *not* hold over TLTSs, regardless of whether they are image-finite or not.

Intuitively, this lack of expressiveness of the logic \(\mathcal{M}_t\) is due to the assumptions we have made about the syntax of clock constraints. Recall that the syntax of clock constraints allows us to compare the values of clocks with integer values. The delay transitions that are possible in an TLTS are, however, labelled by arbitrary non-negative real numbers. This means, for instance, that there is nothing that prevents us from specifying a TLTS like this:

- the set of states of the TLTS is \(\{(A, d), (B, d) \mid d \in \mathbb{R}_{\geq 0}\} \cup \{\text{End}\}\),
- for each \(d < \sqrt{2}\), there are transitions \((A, d) \xrightarrow{a} \text{End}\) and \((B, d) \xrightarrow{a} \text{End}\),
- \((B, \sqrt{2}) \xrightarrow{a} \text{End}\) holds, and
- for each \(d, d' \in \mathbb{R}_{\geq 0}\), we have that
  \[
  (A, d) \xrightarrow{d'} (A, d + d') \\
  (B, d) \xrightarrow{d'} (B, d + d') \quad \text{and} \\
  \text{End} \xrightarrow{d'} \text{End}.
  \]

Observe, first of all, that the states \((A, 0)\) and \((B, 0)\) are not timed bisimilar. Indeed

\[
(B, 0) \xrightarrow{a} (B, \sqrt{2}) \xrightarrow{a} \text{End}
\]

whereas the only state that \((A, 0)\) can reach by delaying \(\sqrt{2}\) units of time is \((A, \sqrt{2})\), from which no \(a\)-labelled transition is possible. You should be able to convince
yourselves that this is the only difference in the behaviour of the two states \((A, 0)\) and \((B, 0)\). Therefore, we could logically distinguish the behaviour of these two states only if the logic \(M_t\) allowed us to express a property stating, informally, that

"by delaying exactly \(\sqrt{2}\) time units one can reach a state from which an a-labelled transition is possible."

Using the language \(M_t\), we are able to specify such properties for integer delays, and after appropriate “change of the time scale” for rational delays (and you are encouraged to try and do so!), but not for irrational delays. Therefore our intuition suggests that the states \((A, 0)\) and \((B, 0)\) satisfy the same properties expressible in \(M_t\). Indeed, the intuition is confirmed by the following result.

**Proposition 11.2** The states \((A, 0)\) and \((B, 0)\) satisfy the same properties expressible in \(M_t\).

**Proof:** (Sketch) The theorem follows from the following stronger claim:

For each \(d < \sqrt{2}\) and valuation \(u\) for the formula clocks, the extended states \(((A, d), u)\) and \(((B, d), u)\) satisfy the same formulae in \(M_t\).

The theorem follows immediately from the above claim by taking \(d = 0\) and \(u = u_0\). The proof of the claim is left as an exercise for the keenest amongst our readers. (See Exercise 11.10.)

**Exercise 11.10 (For the Keenest)** Show the claim made in the above proof. To this end, you might find it useful to begin by proving the claim by induction on the structure of formulae assuming the following auxiliary statements:

1. \((A, \sqrt{2})\) and \((B, d)\) are timed bisimilar for each \(d > \sqrt{2}\),
2. for each \(d, e > \sqrt{2}\), the states \((A, d)\) and \((B, e)\) are timed bisimilar, and
3. for each \(d < \sqrt{2}\), clock valuations \(u, u'\) and formula \(F\),

\[([(A, \sqrt{2}), u] |= F \text{ and } [(A, d), u'] |= F \text{ imply } [(B, \sqrt{2}), u] |= F).\]

Next you should proceed to establish each of the above auxiliary statements. For the last statement, use structural induction on \(F\).

To sum up what we have learned from the above discussion, we have no hope of achieving a characterization theorem for timed bisimilarity in terms of the logic \(M_t\) over arbitrary timed labelled transition systems. However, this is not as bad as it sounds! Indeed, timed labelled transition systems are a very expressive formalism for real-time systems, and cannot in general be finitely described—for instance,
by means of timed automata. Since timed automata provide a good formalism for the finite description of timed labelled transition systems, and the syntactic restrictions that we have imposed on the clock constraints are exactly the same as those present in the syntax of the logic $\mathcal{M}_t$, we might expect that a converse of Theorem 11.1 holds over (states of) timed automata. The following result states that this is indeed the case.

**Theorem 11.2** Let $A$ and $B$ be two timed automata that satisfy the same formulae in the logic $\mathcal{M}_t$. Then $A$ and $B$ are timed bisimilar.

**Proof:** Kim: What would be a good reference for this result? □

As an immediate consequence of the above theorem and of Corollary 11.1, we can now obtain the following result—offering a counterpart of the characterization theorem of Hennessy and Milner for timed automata.

**Corollary 11.2** Two timed automata are timed bisimilar if, and only if, they satisfy the same formulae in the language $\mathcal{M}_t$.

An interesting, and useful, consequence of Theorem 11.2 above is that whenever two timed automata are not timed bisimilar, then we can always find a formula in the language $\mathcal{M}_t$ that one satisfies, but the other does not. This formula, which is often referred to as a distinguishing formula, gives a reason why the two timed automata are not timed bisimilar, and can be algorithmically constructed. (The details of the algorithmic construction for the synthesis of distinguishing formulae are beyond the scope of this introductory textbook. We refer the interested readers to the paper (Godskesen and Larsen, 1995) for more information.)

We have already seen examples of such distinguishing formulae in this section. Such formulae play an important role in implementation verification. Indeed, if we use timed bisimilarity as our notion of equivalence between real-time systems, and the timed automaton describing an implementation of a system is not equivalent to the specification automaton, then a distinguishing formula offers a reason why the implementation is not correct with respect to the given specification. That formula can be used as debugging information to locate the source of the error in the implementation, and correct it.

**Exercise 11.11** Would Theorem 11.1 hold if all we knew about the states $p$ and $q$ were that they are untimed bisimilar? ♦

**Exercise 11.12** Find a sub-language of $\mathcal{M}_t$ that characterizes untimed bisimilarity over TLTSs. ♦
Exercise 11.13  In Example 9.1, we saw how to view the set of non-negative real numbers as a TLTS. Using the same ideas, we can view the intervals \([0, \sqrt{2})\) and \([0, \sqrt{2}]\) as two TLTSs with the number 0 as their distinguished initial state.

1. Are these two TLTSs timed bisimilar?

2. If your answer is no, can you find a formula in \( \mathcal{M}_t \) that distinguishes them?

Motivate your answers!

11.3  Adding Recursively Defined Formulae to HML with Time

In the previous text on Hennessy-Milner logic we successfully extended the language to handle recursively defined formulae. In what follows we aim at introducing formulae with one recursively defined variable also for Hennessy-Milner logic with time.

Consider the timed automaton below (this automaton will be our running example in this section).

Using the logic \( \mathcal{M}_t \) we are able to express that, in its initial state, no matter how this automaton performs two \( a \)-actions in a row, the time delay between those action occurrences will be of at most one time unit. A formula in the language \( \mathcal{M}_t \) stating this property is

\[
\text{TwoAs} \equiv [a](\forall \mathcal{A}[\mathcal{A}[y \leq 1]]).
\]

(11.2)

(We encourage you to show that the initial state of the above timed automaton does satisfy this formula using the formal definition of the satisfaction relation for formulae in \( \mathcal{M}_t \) over states of timed automata.) However, a little reflection should convince you that the above property does not just hold for the initial state of the timed automaton we are considering. Rather, this property holds for all of the states of the timed labelled transition system that gives semantics to that timed automaton. In other words, we expect that the automaton under consideration has the following property:

“\text{It is always the case that whenever two } a \text{-actions occur in a row, then the time delay between them is of at most one time unit.}”
This natural property, however, cannot be expressed in the language \( M_t \). In fact, even though the modal operators \( \exists \) (“there is a delay”) and \( \forall \) (“for each delay”) allow us to examine the behaviour of a state of a timed automaton for arbitrarily long delays, an \( M_t \) formula can only describe a finite part of the overall behaviour of a process that is due to the performance of actions.

You might recall that we discussed a similar shortcoming for HML in Section 6. As was the case for HML, a single formula in the language \( M_t \) can only describe properties of a fixed finite fragment of the computations of a real-time system that are due to action occurrences. As we discussed in Section 6, how much of the behaviour of a real-time system we can explore using a single formula is entirely determined by its so-called modal depth—that is, by the maximum nesting of action modalities in it.

The formula we stated informally above is an example of a safety or invariant property. In Section 6, we saw that a natural specification language in which one can express properties like the aforementioned one is HML extended with a facility for the recursive definition of properties. Following the developments in that section, we shall now extend the language \( M_t \) with recursion in similar fashion. In order to keep our presentation as simple and intuitive as possible, we shall consider the language obtained by extending \( M_t \) with a single recursively defined formula, specified by the identifier \( X \). As in the setting of HML with recursion, this identifier will denote a set of states (namely those that satisfy the property it expresses), and can be used in the definition of formulae using the abstract syntax of the language \( M_t \).

How can we specify a timing property recursively? Let us consider, by way of example, the formal description of the aforementioned property

“It is always the case that whenever two \( a \)-actions occur in a row, then the time delay between them is of at most one time unit.”

A state of a timed automaton satisfies this property if

\[
\bullet \quad \text{whenever it does two } a \text{-actions in a row, the time delay between them is of at most one time unit,}
\]

\[
\bullet \quad \text{if each state that it can reach by delaying some amount of time has the property, and}
\]

\[
\bullet \quad \text{if each state that it can reach by performing an action also has that property.}
\]

Assuming, for the sake of simplicity, that \( a \) is the only action the timed automaton performs, this means that the property above should satisfy the following recursive equation:

\[
X \equiv \text{TwoAs} \land \forall X \land [a]X \quad (11.3)
\]
where the property TwoAs is the one given in (11.2). As claimed in our previous
discussion, the first conjunct on the right-hand side of the recursive equation above
states that if the present state performs two \(a\)-actions in a row, then the time delay
between them is of at most one time unit. The second conjunct states the require-
ment that the property is preserved by arbitrary delays, and the third that it still
holds true after the performance of the action \(a\).

As you might recall from our discussion in Section 6, the above recursive equa-
tion is meant to specify a set of states in a timed automaton—namely, the set of
states that satisfy the property \(X\) is supposed to express. By analogy with our de-
velopments of the theory of HML with recursion, the formula \(X\) is supposed to
stand for a set \(S\) of states of a timed automaton such that

\[
S = \bigl[ \text{TwoAs} \bigr] \cap \bigl[ \cdot \cdot \bigr] S \cap \bigl[ \cdot a \cdot \bigr] S \tag{11.4}
\]

where \([\text{TwoAs}]\) stands for the set of states that satisfy the formula TwoAs in (11.2).
Note that the empty set of states satisfies the above “set equation”. This is due to
the fact that \(\bigl[ \cdot \cdot \bigr] \emptyset = \emptyset\). However, this is certainly not the meaning we have in mind
for the formula \(X\)!

Observe that the empty set is the least solution of the above set equation. By
analogy with our developments in the setting of HML with recursion, we expect
instead that the solution we have in mind for the set equation corresponding to the
recursive equation defining the property \(X\) is the largest one. This is because the
property that \(X\) is intended to formalize is a safety property. (See the discu-
sion in Section 6.1.) In the timed labelled transition system that gives semantics to
our running example, it turns out that the largest set of states that satisfies the set
equation (11.4) is the set \(\text{Proc}\) of all of states of that automaton. This is because
\([\text{TwoAs}]\) is equal to \(\mathcal{E}S(\text{Proc})\) since, for each \(d \in \mathbb{R}_{\geq 0},\)

\[
(\ell, [x = d]) \xrightarrow{a} \text{ implies } d \leq 1 .
\]

As we shall see in what follows, the same techniques from standard fixed point
theory, that turned out to be very useful in making sure that the set equations asso-
ciated with recursively defined formulae in HML have least and largest solutions,
can be applied here.

Formally, the syntax of Hennessy-Milner logic with time and one variable \(X\)
is given by the following grammar:

\[
F ::= X \mid \# \mid ff \mid F \wedge G \mid F \vee G \mid \langle a \rangle F \mid [a] F \mid \exists F \mid \forall F \mid x \downarrow F \mid g
\]

where \(a \in \text{Act}, x \in D\) and \(g \in \mathcal{B}(D)\).
In what follows, we shall interpret formulae in the above language over the collection of extended states associated with a given timed automaton \( A \)—for instance, the one in our running example. Let \( \text{Proc} \) denote the set of states of the timed automaton under consideration. Semantically a formula \( F \) (that may contain the variable \( X \)) is interpreted as a function \( O_F : \mathcal{ES}(\text{Proc}) \to \mathcal{ES}(\text{Proc}) \) that, given a set of extended states that are assumed to satisfy \( X \), gives us the set of extended states that satisfy \( F \). The function \( O_F \) may be defined as follows along the lines of Definition 6.1.

**Definition 11.5** For each \( S \subseteq \mathcal{ES}(\text{Proc}) \) and formula \( F \), we define \( O_F(S) \) inductively by:

- \( O_X(S) = S \)
- \( O_\mathbb{E}(S) = \mathcal{ES}(\text{Proc}) \)
- \( O_\mathbb{F}(S) = \emptyset \)
- \( O_{F_1 \land F_2}(S) = O_{F_1}(S) \cap O_{F_2}(S) \)
- \( O_{F_1 \lor F_2}(S) = O_{F_1}(S) \cup O_{F_2}(S) \)
- \( O_{\langle a \rangle F}(S) = \langle \cdot \rangle O_F(S) \)
- \( O_{\langle a \rangle F}(S) = [\cdot \langle a \rangle \rangle O_F(S) \)
- \( O_{\exists x \ F}(S) = \langle \cdot \rangle O_F(S) \)
- \( O_{\forall x \ F}(S) = [\cdot \langle a \rangle \rangle O_F(S) \)
- \( O_{\exists \ F}(S) = \{(p, u) \mid (p, u[x \mapsto 0]) \in O_F(S)\} \)
- \( O_{\forall \ F}(S) = \{(p, u) \mid u \models g\} \).

**Exercise 11.14** Use the above definition to calculate \( O_{\langle a \rangle F}(\{(p, u[x = 0]), [y = 0]\}) \), where \( F \) is the formula on the right-hand side of the defining equation for \( X \) in (11.3).

**Exercise 11.15** Assume that \( S_1 \) and \( S_2 \) are subsets of \( \mathcal{ES}(\text{Proc}) \) with \( S_1 \) included in \( S_2 \). Argue that

- \( \langle \cdot \rangle S_1 \subseteq \langle \cdot \rangle S_2 \)
- \( [\cdot \rangle S_1 \subseteq [\cdot \rangle S_2 \) and
- \( \{(p, u) \mid (p, u[x \mapsto 0]) \in S_1\} \subseteq \{(p, u) \mid (p, u[x \mapsto 0]) \in S_2\} \).

Use these observations, together with your answers to Exercise 6.5, to show that \( O_F \) is monotonic for each \( F \). (In other words, for all subsets \( S_1, S_2 \) of \( \mathcal{ES}(\text{Proc}) \), if \( S_1 \subseteq S_2 \) then \( O_F(S_1) \subseteq O_F(S_2) \).)
Exercise 11.16  Show that \( \mathcal{P}(\mathcal{ES}(\text{Proc})), \subseteq \) is a complete lattice.

We now know that, for each \( F \), the function \( \mathcal{O}_F \) is monotonic over the complete lattice \( \mathcal{P}(\mathcal{ES}(\text{Proc})), \subseteq \). As mentioned above, and by analogy with our developments in Section 6.2, the idea underlying the definition of the function \( \mathcal{O}_F \) is that if \( [X] \subseteq \mathcal{ES}(\text{Proc}) \) gives the set of extended states that satisfy \( X \), then \( \mathcal{O}_F([X]) \) will be the set of extended states that satisfy \( F \). As we did in the setting of standard HML with recursion, syntactically we shall assume that \( [X] \) is implicitly given by a recursive equation for \( X \) of the form

\[
X^{\min} = F_X \quad \text{or} \quad X^{\max} = F_X .
\]

As argued by example above, such an equation can semantically be interpreted as the following set equation.

\[
[X] = \mathcal{O}_{F_X}([X]) \quad \text{(11.5)}
\]

As \( \mathcal{O}_{F_X} \) is a monotonic function over a complete lattice we know that (11.5) has solutions—that is, that \( \mathcal{O}_{F_X} \) has fixed points. In particular Tarski’s Fixed Point Theorem gives us that there are the largest fixed point and the least fixed-point given respectively by

\[
\bigcup \{ S \subseteq \mathcal{ES}(\text{Proc}) \mid S \subseteq \mathcal{O}_{F_X}(S) \} \quad \text{and} \quad \bigcap \{ S \subseteq \mathcal{ES}(\text{Proc}) \mid \mathcal{O}_{F_X}(S) \subseteq S \} .
\]

Let us use the former expression above to argue formally that the largest fixed point of the function \( \mathcal{O}_F \), where \( F \) is the formula on the right-hand side of the defining equation for \( X \) in (11.3), over the set of extended states for our running example is indeed the whole collection of extended states, as claimed. To this end, it suffices only to show that

\[
\mathcal{ES}(\text{Proc}) \subseteq \mathcal{O}_{F}(\mathcal{ES}(\text{Proc})) .
\]

(Why?) We have already argued that \([\text{TwoAs}]\) is equal to \( \mathcal{ES}(\text{Proc}) \), so using Definition 11.5, you should be able to convince yourselves that

\[
\mathcal{O}_{F}(\mathcal{ES}(\text{Proc})) \quad = \quad [\varepsilon:]\mathcal{ES}(\text{Proc}) \cap [a:]\mathcal{ES}(\text{Proc}) .
\]

Observe now that both \([\varepsilon:]\mathcal{ES}(\text{Proc})\) and \([a:]\mathcal{ES}(\text{Proc})\) are equal to \( \mathcal{ES}(\text{Proc}) \) (Exercise 11.3). Therefore \( \mathcal{O}_{F}(\mathcal{ES}(\text{Proc})) \) is just \( \mathcal{ES}(\text{Proc}) \), as claimed.

Characteristic Properties for Timed Bisimilarity

In Section 6.6, we saw how to characterize the equivalence classes for strong bisimulation with a single formula in HML with recursive definition. The formula that
characterizes the bisimulation equivalence class for a state in a finite labelled transition system was called its characteristic formula.

It is natural to ask whether the language $\mathcal{M}_t$ extended with recursively defined formulae is expressive enough to allow us to obtain a similar result for timed bisimilarity over timed automata. Indeed, achieving such a result would give us yet another indication that our design choices for the logic $\mathcal{M}_t$ are “good”, at least in that this language affords properties that are akin to those of classic Hennessy-Milner logic—with or without recursively defined formulae.

For the sake of simplicity, we shall focus in this section on timed automata without invariants—that is, on time automata whose location invariants are all tautologies—, and over a single action $a$.

Consider, by way of example, the timed automaton used as our running example in this section.

A formula characterizing node $\ell$ in this timed automaton up to timed bisimilarity should offer a description of:

1. all the actions that are enabled in the node,

2. which node is entered by taking a given edge, together with the clock resets associated with it, and

3. the fact that arbitrary delays are allowed in the node.

The resulting characteristic formula is presented below, where we consider each $X_\ell$ to be our recursively defined variable. The formula consists of three conjuncts, each associated to one of the above properties:

$$X_\ell^{\max} = (x \leq 1 \Rightarrow ((a)x \mathcal{M}_\ell X_\ell)) \land$$

$$[a](x \leq 1 \land (x \mathcal{M}_\ell X_\ell)) \land$$

$$\forall X_\ell.$$

The above formula encodes the behaviour of the timed automaton as follows. The first conjunct in the formula on the right-hand side of the recursive equation states that if the value of the clock $x$ is no larger than 1, then the timed automaton can perform an $a$-labelled transition, and reach a state that satisfies the characteristic
formula for node $\ell$ after resetting the clock $x$. (Note that this conjunct encodes the self loop in the timed automaton.) The second conjunct is intended to express that the self loop edge is the only one in this timed automaton. This is done by saying that no matter how an $a$-action is performed, this is done within one time unit from the last time the clock $x$ was reset, and it will lead to a state that satisfies the characteristic formula for node $\ell$ after resetting clock $x$. The third and last conjunct expresses the fact that, no matter how long we delay, we should still satisfy the characteristic formula for node $\ell$.

The following theorem states that the above recursively defined formula is characteristic for our running example modulo timed bisimilarity.

**Theorem 11.3** Let $A$ be a timed automaton whose set of clocks does not include $x$. Let $\ell'$ be a node of $A$. Assume that $d \in \mathbb{R}_{\geq 0}$ and $u$ is a valuation for the clocks of $A$. Then $((\ell', [x = d]), x = d')$ is timed bisimilar to $(\ell', u)$ if, and only if, the extended state $((\ell', u), [x = d])$ satisfies $X_\ell$.

**Proof:** We first show that the extended state $((\ell', u), [x = d])$ satisfies $X_\ell$ whenever $(\ell, [x = d])$ is timed bisimilar to $(\ell', u)$. To this end, it suffices only to argue that

$$S \subseteq O_F(S),$$

where $F$ is the formula on the right-hand side of the recursive definition for $X_\ell$, and the set $S$ is defined as:

$$S = \{(\ell', u'), [x = d'] | (\ell, [x = d]) \text{ is timed bisimilar to } (\ell', u')\}.$$

(Convince yourselves of this claim!)

Assume that $((\ell', u'), [x = d']) \in S$. We shall prove that $((\ell', u'), [x = d'])$ is also contained in $O_F(S)$. By the definition of the function $O_F$, this amounts to arguing that

1. $((\ell', u'), [x = d']) \in O_{x \leq 1 \Rightarrow (a \times \mathbb{N} \times X_\ell)}(S)$,
2. $((\ell', u'), [x = d']) \in O_{[a](x \leq 1 \land (x \mathbb{N} X_\ell))}(S)$ and
3. $((\ell', u'), [x = d']) \in O_{\forall X_\ell}(S)$.

We shall limit ourselves to presenting the details of the verification of the first of these claims. (You are invited to fill in the details of the proof of the latter two claims yourselves.)

Observe, first of all, that $((\ell', u'), [x = d']) \in O_{x \leq 1 \Rightarrow (a \times \mathbb{N} \times X_\ell)}(S)$ holds trivially if $d' > 1$. (Why?) Assume therefore that $d' \leq 1$. In this case, we wish to argue that

$$((\ell', u'), [x = d']) \in O_{(a \times \mathbb{N} \times X_\ell)}(S).$$
By the definition of the function $O$, this holds precisely when, for some state $(\ell'', u'')$ of the timed automaton $A$,
\[
(\ell', u') \overset{a}{\rightarrow} (\ell'', u'') \text{ and } ((\ell'', u''), [x = 0]) \in S.
\]
(Check this claim!) To see that the above criterion is met by the state $(\ell', u')$, we argue as follows. Since $d' \leq 1$, we have that
\[
(\ell, [x = d']) \overset{a}{\rightarrow} (\ell, [x = 0]).
\]
By the definition of $S$, the states $(\ell', u')$ and $(\ell, [x = d'])$ are timed bisimilar. Therefore, there exists a state $(\ell'', u'')$ of the timed automaton $A$ such that
\[
(\ell', u') \overset{a}{\rightarrow} (\ell'', u'') \text{ and } (\ell'', u'') \text{ is timed bisimilar to } (\ell, [x = 0]).
\]
Again by the definition of $S$, we may conclude that
\[
((\ell'', u''), [x = 0]) \in S,
\]
which was to be shown.

Our order of business will now be to show that if the extended state $((\ell', u), [x = d])$ satisfies $X_\ell$, then $(\ell, [x = d])$ is timed bisimilar to $(\ell', u)$. To this end, it suffices only to prove that the relation
\[
R = \{((\ell, [x = d]), (\ell', u)) \mid ((\ell', u), [x = d]) \models X_\ell\}
\]
is a timed bisimulation. The proof is left as a strongly recommended exercise for the reader. □

**Exercise 11.17 (Strongly Recommended)** Complete the proof of the above theorem.

This theorem and the construction of the characteristic formula for our running example are specific instances of the general construction of characteristic formulae, and of the theorem showing its correctness, presented in (Aceto, Ingólfsdóttir, Pedersen and Poulsen, 2000). Related results may be found in (Aceto, Bouyer, Burgueño and Larsen, 2003; Laroussinie et al., 1995)—the latter reference offers, to the best of our knowledge, the first construction of characteristic formulae for timed automata, modulo timed bisimilarity presented in the literature.

**Exercise 11.18** Give characteristic formulae for the timed automata in Example 10.9, and prove a version of Theorem 11.3 for them. (You might find it useful to define a characteristic formula for each location in the timed automata. You won’t need recursion!)
Exercise 11.19 (Characteristic Formulae for Timed Simulation) A timed simulation over the states of some timed labelled transition system is a relation $R$ such that whenever $s_1 R s_2$ and $a \in \text{Lab}$, then:

- if $s_1 \xrightarrow{a} s'_1$ then $s_2 \xrightarrow{a} s'_2$ for some $s'_2$ such that $s'_1 R s'_2$.

State $s_1$ is simulated by $s_2$ iff the pair $(s_1, s_2)$ is contained in a timed simulation.

For timed automata $A_1$ and $A_2$, we say that $A_1$ is simulated by $A_2$ iff the initial state of $A_1$ is simulated by that of $A_2$.

Give a characteristic formula for our running example modulo timed simulation, and prove a version of Theorem 11.3 for it.

♦

Examples of Real-time Temporal Properties

The basic constructs of the logic $\mathcal{M}_t$ extended with recursive definitions can be used to define high level temporal operators, which may be used to simplify the writing of logical specifications. Here we confine ourselves to showing how to define the temporal operators until, before and Inv: (in the following formulae, $t$ is a non-negative integer)

$$
F \text{ until } G \overset{\text{max}}{=} G \lor (F \land [\text{Act}](F \text{ until } G) \land \Box(F \text{ until } G))
$$

$$
F \text{ until}_{\leq t} G = \mu n ((F \land x \leq t) \text{ until } G)
$$

$$
\text{before}_t F = \mu n \mu n (F \text{ until } \leq t F)
$$

$$
\text{Inv}(F) \overset{\text{max}}{=} F \land [\text{Act}]\text{Inv}(F) \land \Box\text{Inv}(F)
$$

The intuitive meaning of the above temporal operators is as follows.

- $F \text{ until } G$ is true iff, no matter how long the systems delays or what action transitions it takes, $F$ is satisfied at least until $G$ becomes true. Since we are specifying this property using a greatest fixed point, the formula $F$ may be satisfied forever and $G$ might never become true. The above formula is therefore an example of a so-called weak until. (See the discussion on page 121.)

- $F \text{ until}_{\leq t} G$ is the time bounded version of the above property. There we are stating that $F$ is satisfied at least until $G$ holds, and moreover $G$ is guaranteed to hold within $t$ time units.

- The formula $\text{before}_t F$ states that $F$ will hold within $t$ time units. This is an example of a time bounded eventuality property.

- Finally, $\text{Inv}(F)$ states that the formula $F$ holds invariantly. (This is just a real-time version of the invariance property that we already met in Section 6.)
11.4 Overview of the Main Results

The second and the third property introduced above are examples of so-called *bounded liveness* properties. This kind of properties arises quite often when we try to describe expected behaviours of real-time systems. Recall that a typical liveness properties states intuitively that “something good will eventually happen.” For instance, a liveness property could specify that each request to access some resource is eventually granted. In real-time setting, however, we are often not just interested in knowing that our requests will be granted at some unspecified time in the future. Rather, we expect that we are granted access to whatever service we need within a specified time bound! As a concrete example, consider the behaviour of an airbag system in a car described by the property

If the car crushes, the airbag must be inflated within 50 milliseconds.

This property is an example of a bounded liveness property, and can be described using the formula

\[ \text{Inv}\left([\text{crush}]\left(\text{before}_{50}\langle\text{inflate}\rangle\right)\right). \]

Indeed, the above formula states that no matter how the system evolves—that is, in all reachable states—, then each crush action can be followed by an inflate action within 50 time units.

11.4 Overview of the Main Results

We can add something here if you think that it is appropriate, and that it fits with the rest of the chapter. What do you say?
Chapter 12

Modelling and Analysis of Fischer’s Algorithm

Mutual exclusion algorithms, like those we discussed in Chapter 7, have the schematic behaviour described by the following pseudocode:

```
while true do
begin
  remainder region
  trying region
  critical section
  exit region
end
```

Such algorithms are supposed to satisfy the following two properties:

- **Mutual exclusion**: No two processes are in their critical section at the same time.

- **Deadlock freedom**: If some process is in its trying region, then eventually some process is in its critical section. (Note that the process in the critical section might be different from the one initially in its trying region.) Moreover, if a process is in its exit region, then that process will eventually enter its remainder region.

Known asynchronous mutual exclusion algorithms for \( n \) processes, require \( O(n) \) read/write registers and \( O(n) \) operations to access the critical section. These bounds make them rather impractical for large scale applications, where the number of processes could be very large. This raises the question of whether it is possible
to achieve mutual exclusion in asynchronous systems consisting of \( n \) processes by using a smaller number of shared registers and/or fewer than \( O(n) \) operations to access the critical section. Unfortunately, this is impossible for “classic reactive systems” in an asynchronous setting. In fact, Burns and Lynch showed the following theorem in (Burns and Lynch, 1989; Burns and Lynch, 1993):

**Theorem 12.1** [Burns and Lynch] There is no asynchronous algorithm providing mutual exclusion with deadlock-freedom for \( n \geq 2 \) processes, using fewer than \( n \) shared read/write registers.

This theorem is a classic example of an *impossibility result*—a type of results that Nancy Lynch has contributed in abundance to the literature on distributed computation. (See, for instance, the impossibility results mentioned in her encyclopedic book (Lynch, 1996).) Despite their apparently negative nature, such results play a fundamental role in the theory and practice of computing science because they set precise limits to what is possible to achieve using some computational paradigm—thus preventing futile efforts to overcome computational barriers that cannot, in fact, be broken within a given computational model. For example, Theorem 12.1 above tells us that there is no point in trying to come up with asynchronous, deadlock-free mutual exclusion algorithms that use fewer than \( n \) shared read/write registers because such algorithms do not exist!

However, as repeatedly stated by Richard Hamming in his general writings on science and engineering—see, for instance, (Hamming, 1997, page 305)—, all impossibility proofs must rest on a number of assumptions, and these assumptions may or may not apply in the particular situation under analysis or in the chosen computational model. For instance, the above result by Burns and Lynch applies to “deadlock free” algorithms. If we remove this, admittedly very reasonable, assumption, then there is a mutual exclusion algorithm that uses no shared register at all. It suffices only to make the execution of all of the processes enter a livelock at the start of the protocol, and none will access its critical section! This “solution” to the mutual exclusion problem is, however, completely unacceptable—so much so that nobody would actually consider it a proper solution anyway. We can, however, ask the following natural question:

**Can the lower bound in Theorem 12.1 for deadlock free mutual exclusion be overcome by considering other computational models than the one underlying the aforementioned result by Burns and Lynch?**

This is a typical question that arises from impossibility and lower bound theorems, and is another example of how apparently negative results can help stimulate the search for new computational paradigms and the exploration of their computing power.
12.1 Mutual Exclusion Using Timing

According to Lynch and Shavit in (Lynch and Shavit, 1992), Michael Fischer seems to have been the first researcher who overcame the lower bound of \( n \) registers for deadlock free mutual exclusion by assuming timing constraints. His, by now classic, algorithm uses just one shared multi-writer register \( \text{id} \), whose initial value is 0. In order to ensure mutual exclusion, each process \( P_i \) \( (i \in \{1, \ldots, n\}) \) executes the following algorithm, where we use delay to stand for a positive integer constant:

```plaintext
while true do
    "noncritical section";
    L: if id \( \neq 0 \) then goto L;
    1: id := i;
    2: pause(delay);
    3: if id \( \neq i \) then goto L;
    "critical section";
    id := 0;
end
```

In the above pseudocode algorithm the statement \texttt{pause}(delay) makes the process wait for the amount of time specified by the constant delay. But, what should be the value of such constant? Since Fischer’s algorithm is a real-time one, we might expect that its behaviour depends crucially on an appropriate choice for this timing parameter.

In order to find a suitable value for the constant delay, we assume an upper bound \( c \), where \( c \) is a positive integer, for the time between successive steps of the execution of a process while it is trying to access its critical section. Intuitively, a process that takes steps every \( c \) time units is executing slowly. In Fischer’s algorithm, we choose the value of the constant delay to be larger than \( c \), the longest time that a process may take to perform a step while trying to enter its critical section. The key idea behind this choice for the parameter delay is that by the time process \( i \) has reached Line 3 in the pseudocode algorithm, each process \( j \) that has passed the test in Line L and might write \( j \) in variable \( \text{id} \) has already done so since delay \( > c \), and \( c \) is the longest time that such a step may take. Therefore, whenever process \( i \) finds that \( \text{id} = i \) in Line 3, then it can safely enter its critical section because all of the other processes are either before Line L, or after Line 1 with their index overwritten by process \( i \)—so they will fail the test on Line 3.

The algorithm we have just presented is conceptually very simple. However, as you will see later on, Fischer’s algorithm has the drawback that it fails to guarantee
It is well-known that Fischer’s algorithm is deadlock-free, and ensures mutual exclusion provided its timing assumptions are met. Moreover, as shown by Lynch and Shavit in (Lynch and Shavit, 1992, Theorem 4.6), its timing behaviour is nearly optimal. Our order of business here is to model Fischer’s algorithm using networks of timed automata, as supported by the verification tool UPPAAL, and to hint at the automatic verification of its behaviour using that tool.

12.2 Modelling Fischer’s Algorithm

Fischer’s mutual exclusion algorithm for $n$ processes can be modelled as a network of timed automata—see Chapter 10.7. Each of the $n$ timed automata in the network will describe the behaviour of one of the processes running Fischer’s algorithm described above in pseudocode. The timed automaton $A_i$ running the code for process $i$ in Fischer’s algorithm will use a local clock $x$ to guarantee that the upper bound between successive steps of the process while it is trying to enter its critical section is $c$, and will have access to the shared integer variable id. The timed automaton $A_i$ is depicted in Figure 12.1.

The label of all of the edges is immaterial for this algorithm, and is therefore omitted from the picture in Figure 12.1. For consistency with previous notation, you can assume that all of the edges have label $\tau$. In that picture, we use the short-hand “$\text{not} (id = i), x > c$” as an abbreviation for the boolean condition

$$(id < i \lor id > i) \land x > c .$$

The edge from location 2 to location L should therefore be read as standing for two
edges: one that applies when id < i and x > c, and the other that is enabled when id > i and x > c. For pictorial convenience, we have also written the invariant x ≤ c, in node 1 within the node itself.

As you might already noticed, the timed automaton in Figure 12.1 is based on a slight extension of the model we introduced in Chapter 10. In fact, the timed automaton A_i uses an integer variable id as well as one clock x. The integer variable id can be updated when the automaton follows an edge, and its current value can be tested to determine whether an edge from the present location is enabled or not. This slightly extended model of timed automata is supported by the verification tool UPPAAL, and makes it easier to model algorithms like Fischer’s mutual exclusion algorithm that rely on the use of shared variables.

Nodes L, 1 and 2 in the timed automaton in Figure 12.1 model the similarly numbered steps in the pseudocode for Fischer’s algorithm, and the timed automaton A_i begins its execution in location L. (As you might have already noticed, location L has no outgoing edge that is enabled when id has value different from 0. The “busy waiting” loop in the behaviour of the pseudocode algorithm is modelled by delaying in location L of automaton A_i.) The invariant x ≤ c in node 1 is used to model the upper bound on the time that a step of the process can take while it is trying to enter the critical section. Such an invariant ensures that process i can be in location 2 for at most c time units. Location 2 in the automaton A_i describes steps 2 and 3 in the pseudocode for Fischer’s algorithm. In fact, since the private clock x of automaton A_i is reset upon entering location 2, the guard x > c on the outgoing edges from that location ensures that the process is delayed more than c units of time before testing the value of the shared variable id. If after that amount of time, the value of id is i, then the process can safely enter its critical section—abstractly modelled here by the location CS—by following its edge from location 2 to location CS. The edge from location CS back to location L implements the exit from the critical section and the resetting of the shared variable id.

Fischer’s algorithm for n processes is modelled as the network of timed automata

\[ A_1 | A_2 | \cdots | A_n. \]

We recall that states of this network consist of an n-tuple of locations (\( \ell_1, \ldots, \ell_n \)), where each \( \ell_i \) is a location of automaton A_i (\( i \in \{1, \ldots, n\} \)), and a valuation for the set of clocks \( \{x_1, \ldots, x_n\} \), where we use \( x_i \) to stand for the local clock x of automaton A_i. However, unlike for the networks of classic timed automata introduced in Section 10.7, this is not enough to give a complete snapshot of the behaviour of this system. In fact, since the value of the shared variable id determines whether certain edges are enabled or not in the component automata, a state of the network must record, in addition, the current value of the shared variable id.
In what follows, we shall write a state of the network \( A_1 | A_2 | \cdots | A_n \) thus:

\[
(\ell_1, \ldots, \ell_n, x_1 = c_1, \ldots, x_n = c_n, \text{id} = i),
\]

where \( c_1, \ldots, c_n \) are non-negative real numbers, and \( i \in \{1, \ldots, n\} \). The initial state of the network is

\[
(L, \ldots, L, x_1 = 0, \ldots, x_n = 0, \text{id} = 0),
\]

because the initial value of each clock and of the variable id is 0.

### 12.2.1 Proving Mutual Exclusion Using UPPAAL

Now that we have a model of Fischer's algorithm as a UPPAAL network of timed automata, our order of business will be to analyze the behaviour of this model to verify that it indeed affords the mutual exclusion property. Before doing so, however, we need to specify precisely what it means for our network of timed automata to guarantee mutual exclusion.

As we saw already in Chapter 7, temporal logics like Hennessy-Milner logic with recursive definitions provide a natural language in which one can specify properties of reactive systems like mutual exclusion. In fact, everything we said there applies *mutatis mutandis* to the setting of real time systems described as networks of timed automata. It would therefore be tempting, and most natural, to describe the mutual exclusion property for our model of Fischer’s algorithm using the real-time version of Hennessy-Milner logic with greatest fixed points that we presented in Chapter 11.

Note, however, that the network of timed automata describing Fischer’s algorithm for \( n \) processes is a *closed system*. This means that that network is not willing to communicate with its environment. Moreover, interaction between the automata in the network takes place via the shared variable id.

On the other hand, Hennessy-Milner logic and its variants are *action based* temporal logics—at least in our textbook presentation. This means that formulae in variants of Hennessy-Milner logic describe properties pertaining to the communication potential of processes via the labelled modalities \( \langle a \rangle \) and \( [a] \). At first sight, this makes these logics unsuitable for describing properties of systems that, like our model of Fischer’s algorithm, exhibit no observable communication behaviour.

What we should like to express for Fischer’s algorithm is an invariant property that states that

“No matter how the network evolves, at no point of its computation two different component automata will be in their location CS at the same time.”
We have already seen how to express invariance properties in Hennessy-Milner logic with time (see Section 11.3), but how do we express the requirement that “two different component automata cannot be in their location CS at the same time”?

One possibility would be to modify our model by adding self-loop edges to location CS in the timed automaton $A_i$. These edges could be labelled with some observable synchronization action, say $\text{in}_i!$, used to signal to the environment that automaton $A_i$ is in its critical section. One could then express mutual exclusion using the property

$$\text{Inv}(\bigwedge_{1 \leq i < j \leq n} ([\text{in}_i!]f \lor [\text{in}_j!]f)) .$$

(You should try to convince yourself that the above property would indeed state that at most one automaton is in its critical section in each state of the computation of Fischer’s algorithm.)

However, if we aim at verifying the correctness of Fischer’s mutual exclusion algorithm using an automatic verification tool like UPPAAL, we are forced to use a specification language for the properties to be model checked that is accepted by the tool itself. Unfortunately, UPPAAL models can only be closed systems, and the language supported by that tool for the writing of specifications does not allow us to write formulae like the invariant property above.

The specification language of the tool UPPAAL, however, permits the use of atomic predicates whose truth value over states can be determined locally. An example such predicate is $A_i.CS$ ($i \in \{1, \ldots, n\}$), stating that the $i$th automaton in the network is presently in location CS. Formally, a state

$$(\ell_1, \ldots, \ell_n, x_1 = c_1, \ldots, x_n = c_n, \text{id} = i) ,$$

of our network satisfies the atomic predicate $A_i.CS$ if, and only if, $\ell_i = \text{CS}$.

Using boolean combinations of these atomic predicates, we can therefore state that at most one of the component automata is currently in its critical section by means of the formula

$$\text{MutexNow} = \bigwedge_{1 \leq i < j \leq n} (\neg A_i.CS \lor \neg A_j.CS) ,$$

where the symbol $\neg$ stands for logical negation.

One can therefore express mutual exclusion using the property

$$\text{Inv}(\text{MutexNow}) .$$
Since the specification language for queries supported by the tool UPPAAL follows the syntax of Timed Computation Tree Logic (or TCTL)—see the reference (Alur, Courcoubetis and Dill, 1993)—, the above property is actually written thus:

$$A\, □ \text{MutexNow}.$$  

In the above formula, the initial $A$ states that the formula $□\text{MutexNow}$ must be satisfied in all of the computation paths of the network. The truth value of the property $□\text{MutexNow}$ is therefore evaluated over a single path. The $□$ modality here indicates that the property MutexNow should be true in all of the states along the path. Following this informal explanation, you should be able to convince yourself that the above formula states the following property:

“In all computation paths, and in each state along each path, at most one process is in its critical section.”

This is precisely what “ensuring mutual exclusion” means. Further information on the actual syntax of the specification language for queries used by UPPAAL may be found in the tutorial paper (Behrmann et al., 2004).

**Exercise 12.1** Read the tutorial paper (Behrmann et al., 2004) carefully. Install the UPPAAL tool, and experiment with the demo examples that come with the tool.

**Exercise 12.2**

1. Create a UPPAAL model for Fischer’s algorithm based on the one proposed in the textbook. In your model you may assume that the network consists of 4 processes and that the value of the constant c is 2.

2. Upload the predefined collection for queries for Fischer’s algorithm that comes with the tool. Check whether your model of Fischer’s algorithm affords all of the stated properties.

**TO DO:** PERHAPS ADD SOME REFERENCE TO HENRIK EJERSBO’S PAPER ON FISCHER’S ALGORITHM.

### 12.2.2 An Erroneous Version of Fischer’s Algorithm

We have already mentioned that timing plays a crucial role in the workings of Fischer’s mutual exclusion algorithm. Indeed, this algorithm fails to ensure mutual
12.2. MODELLING FISCHER’S ALGORITHM

exclusion if its timing assumptions are not satisfied. In particular, it is crucial that, on line 2 of the pseudocode algorithm, process $i$ be delayed by some amount of time that is strictly larger than the constant $c$—namely, the longest time that it takes for a process to take a step while it is trying to enter its critical section. To see this, consider an erroneous version of Fischer’s protocol modelled by the timed automaton $A^w_i$ in Figure 12.2. Note that the only difference between that timed automaton and the one in Figure 12.1 is in the guards labelling the edges that stem from node 2. In particular, the process may now check whether the shared variable $id$ has value $i$ after a delay of exactly $c$ units of time.

We shall now exhibit a sequence of transitions for our model of this erroneous version of Fischer’s algorithm that does not preserve mutual exclusion.

Assume, for the sake of simplicity, that there are only two processes running this version of Fischer’s algorithm. The overall network of timed automata is therefore $A^w_1 | A^w_2$, and its initial state is given by

$$(L, L, x_1 = 0, x_2 = 0, id = 0)$$

where, for $i \in \{1, 2\}$, we write $x_i$ for the local clock $x$ of automaton $A^w_i$. Since the value of id is 0, and the edge from location L to location 1 does not change it, the network $A^w_1 | A^w_2$ can perform the following two transitions:

$$\begin{align*}
(L, L, x_1 = 0, x_2 = 0, id = 0) & \rightarrow (1, L, x_1 = 0, x_2 = 0, id = 0) \\
& \rightarrow (1, 1, x_1 = 0, x_2 = 0, id = 0).
\end{align*}$$

In state $(1, 1, x_1 = 0, x_2 = 0, id = 0)$, automaton $A^w_1$ may follow the edge from location 1 to location 2, and thereafter the system may delay for $c$ time units. Thus
the network $A_1^w | A_2^w$ can perform the following two transitions:

$$(1, 1, x_1 = 0, x_2 = 0, \text{id} = 0) \xrightarrow{(1, x_1 = 0, x_2 = 0, \text{id} = 1)} (2, 1, x_1 = c, x_2 = c, \text{id} = 1).$$

Since the value of $x_1$ is $c$ and that of $\text{id}$ is 1, automaton $A_1^w$ may now enter its critical section:

$$(2, 1, x_1 = c, x_2 = c, \text{id} = 1) \xrightarrow{c} (CS, 1, x_1 = c, x_2 = c, \text{id} = 1).$$

At this point of the computation, automaton $A_2^w$ may decide to follow the edge from location 1 to location 2, and thereafter the system may delay for $c$ time units. Thus the network $A_1^w | A_2^w$ can perform the following two transitions:

$$(CS, 1, x_1 = c, x_2 = c, \text{id} = 1) \xrightarrow{(CS, 2, x_1 = c, x_2 = 0, \text{id} = 2)} (CS, 2, x_1 = 2c, x_2 = c, \text{id} = 2).$$

Note now that automaton $A_2^w$ can also enter the critical section because the value of $x_2$ is $c$ and that of $\text{id}$ is 2:

$$(CS, 2, x_1 = 2c, x_2 = c, \text{id} = 2) \xrightarrow{c} (CS, CS, x_1 = 2c, x_2 = c, \text{id} = 2).$$

In the target state of the above transition, both $A_1^w$ and $A_2^w$ are in their critical section, leading to the claimed failure of mutual exclusion. (You should now be able to argue that a similar behaviour is possible for an arbitrary network of $n$ processes. Do so!)

We may therefore conclude that, as claimed previously, the correctness of Fischer’s mutual exclusion algorithm depends crucially on its timing assumptions. In the following section, you will be working with, amongst others, a modification of this algorithm proposed by Lynch and Shavit that guarantees mutual exclusion regardless of whether the timing assumptions on the speed of the processes are met or not.

Exercise 12.3 Implement the erroneous version of Fischer’s algorithm for $n = 4$ and $c = 2$ in UPPAAL. Use the tool to determine that the system does not afford mutual exclusion, and to find a (shortest/fastest) trace leading to a state where at least two processes are in their critical section at the same time.

12.3 Further Exercises on Timing Based Mutual Exclusion Algorithms

In the previous sections, we have seen how to use timing information to ensure mutual exclusion in an asynchronous setting by means a truly simple and beautiful
while true do
begin
    start: $x := i$;
    await ($y = 0$);
    $y := i$;
    if $x \neq i$ then delay ($2 \cdot \Delta$);
    if $y \neq i$ then goto start;
    await ($\neg z$)
else $z := true$;
    “critical section”;
    $z := false$;
    if $y = i$ then $y := 0$;
end

Table 12.1: Program for Process $i$ (Alur and Taubenfeld)

algorithm due to Michael Fischer. In Fischer’s solution, before it can enter the critical section, a process accesses the single shared variable $id$ thrice, and delays itself some amount of time that is larger than the upper bound on the time that processes need to execute a step. Note that, in Fischer’s algorithm, a process delays itself even if it is the only one that is currently trying to enter the critical section.

Another elegant, timing based mutual exclusion algorithm has been proposed by Alur and Taubenfeld in (Alur and Taubenfeld, 1992; Alur and Taubenfeld, 1996). The solution to the mutual exclusion problem proposed by those authors assumes that there is an upper bound $\Delta$ on the time required for reading or writing a variable in the shared memory. Furthermore, Alur and Taubenfeld suppose that this bound $\Delta$ is known to all of the processes in the system. Access to the shared memory takes a non-zero time, and there is no lower bound on the time needed to execute a step. As in the pseudocode for Fischer’s algorithm, processes can delay themselves by performing an explicit delay ($d$) statement, where $d$ is a positive integer.

Alur and Taubenfeld’s mutual exclusion algorithm uses three shared registers; the registers $x, y$ hold integers—with $y$ having initially the value 0—, and register $z$ holds a boolean value that is initially the value false. In order to ensure mutual exclusion, each process $P_i$ ($i \in \{1, \ldots, n\}$) executes the pseudocode algorithm in Table 12.1.

Note that, unlike in Fischer’s algorithm, in the absence of competing processes that want to enter their critical section, in the algorithm by Alur and Taubenfeld a process can always enter and exit its critical section without having to delay itself.
while true do
begin
start: $x := i$;
await ($y = 0$);
$y := i$;
if $x \neq i$ then delay($\Delta$);
  if $y \neq i$ then goto start;
  delay($\Delta$);
await ($\neg z$)
else $z := true$;
“critical section”;
$z := false$;
if $y = i$ then $y := 0$;
end

Table 12.2: Program for Process $i$ (Revised Algorithm)

In fact, in that case, the process writes $x$, reads $y$, writes $y$, reads $x$ (finding it equal to $i$ because no other process started the protocol to enter the critical section), writes $z$ and enters its critical section.

Alur and Taubenfeld proved in (Alur and Taubenfeld, 1992; Alur and Taubenfeld, 1996) that the algorithm in Table 12.1 and is deadlock free. Moreover, they report a mechanical verification of their proof for $n = 3$ processes using the verification tool COSPAN (Alur, Itai, Kurshan and Yannakakis, 1995).

Exercise 12.4 Model the algorithm in Table 12.1 in UPPAAL for $n = 3$ processes and $\Delta = 2$. Verify that it preserves mutual exclusion. Increase the number of processes to four and five, and repeat the verification.

Exercise 12.5 Take the model you produced in your solution to the previous exercise, and modify it so that at least one of the memory accesses of the processes takes at most one time unit. Does the resulting algorithm still preserve mutual exclusion? Experiment with different choices of “fast steps”. Do your conclusions depend on the steps that are chosen to be “fast”?

Exercise 12.6 Consider the variation of the algorithm by Alur and Taubenfeld offered in Table 12.2. Model it using UPPAAL for $n = 3$ processes and $\Delta = 2$. Verify that it preserves mutual exclusion. Increase the number of processes to four and five, and repeat the verification.
Despite being very elegant and fast, the algorithms by Fischer and Alur/Taubenfeld suffer, however, from an important drawback: they fail to guarantee mutual exclusion if the timing constraints upon which their workings are predicated are not satisfied. (Indeed, the solutions you gave to Exercises 12.3 and 12.5 should have already convinced you of this!)

According to Lynch and Shavit, a timing based mutual exclusion algorithm should guarantee mutual exclusion regardless of the timing constraints. In (Lynch and Shavit, 1992), they offered a simple and efficient timing based mutual exclusion algorithm that guarantees mutual exclusion regardless of the timing constraints. Their algorithm uses two shared integer registers $x, y$, whose value is initially 0. Moreover, as in Fischer’s algorithm, delay stands for a positive integer constant. The pseudocode for process $i$ in their algorithm is as follows:

```plaintext
while true do
begin
    “noncritical section”; 
    L: if $x \neq 0$ then goto L; 
    1: $x := i$; 
    2: pause(delay); 
    3: if $x \neq i$ then goto L; 
    4: if $y \neq 0$ then goto L; 
    5: $y := 1$; 
    6: if $x \neq i$ then goto L; 
    7: “critical section”; 
    8: $y := 0$; 
    9: $x := 0$; 
end
```

**Exercise 12.7** Model the algorithm by Lynch and Shavit in UPPAAL for $n = 3$ processes and delay = 2. Verify that it preserves mutual exclusion, for different upper bounds on the time that it takes for processes to execute steps in their entry to the critical section. Increase the number of processes to four and five, and repeat the verification.
Chapter 13

Suggestions for student projects

This appendix will contain descriptions of the student mini-projects that are presently available from the following web pages:

- http://www.cs.aau.dk/~luca/SV/miniproject.html,
- http://www.cs.aau.dk/~srba/courses/SV-05/mini.html,

The text below offers a preliminary description of a couple of these proposals for mini-projects. All of the mini-projects involve the use of software tools for verification and validation. In our courses, we usually introduce the students to the use of the Concurrency Workbench (CWB)\(^1\) and to UPPAAL\(^2\), but other tools could be used just as well.

13.1 Alternating bit protocol

In this mini-project you are asked to model the alternating bit protocol in the CCS language and verify it using the CWB. The alternating bit protocol is a simple yet effective protocol for managing the retransmission of lost messages. Consider a sender \(S\) and a receiver \(R\), and assume that the communication medium from \(S\) to \(R\) is initialized so that there are no messages in transit. The alternating bit protocol works as follows:

- Each message sent by \(S\) contains an additional protocol bit, 0 or 1.

\(^1\)http://homepages.inf.ed.ac.uk/perdita/cwb/
\(^2\)http://www.uppaal.com/
• When $S$ sends a message, it sends it repeatedly (with its corresponding bit) until it receives an acknowledgment (ACK) from $R$ that contains the same protocol bit as the message being sent.

• When $R$ receives a message, it sends an acknowledgment ACK to $S$ and includes the protocol bit of the received message. When a message is received for the first time, the receiver delivers it for processing, while subsequent messages with the same bit are simply acknowledged.

• When $S$ receives an acknowledgment containing the same bit as the message it is currently transmitting, it stops transmitting that message, flips the protocol bit, and repeats the protocol for the next message.

There is no direct communication between the sender and the receiver; all messages must travel through the medium.

Your tasks are as follows:

1. Implement the alternating bit protocol in the CWB. You can abstract away from the content of the messages and focus only on the additional control bit. To model the decision when the sender retransmits the message, use either nondeterminism or, ever better, a special process called Timer. The process Timer will communicate with the sender on a channel called timeout, and signal when a message should be retransmitted. You can also try to model the checksum verification using nondeterminism. Information on checksum verification is available from the URL http://www.answers.com/topic/alternating-bit-protocol.

2. Suggest a specification of the protocol and check whether it is equivalent to your implementation using a suitable equivalence notion available in the CWB. In particular, consider the following degrees of reliability of the communication medium and answer this question for all of these choices:

   (a) perfect channels (all received messages are delivered),

   (b) lossy channels (received messages can be lost without any warning) and

   (c) lossy and duplicating (in addition the received message can be delivered several times).
3. Check for possible deadlocks (stuck configurations) and livelocks (a possibility of an infinite sequence of $\tau$-labelled transitions) by formulating the properties as recursive formulae in Hennessy-Milner logic, and by verifying whether the implementation satisfies these formulae.

**Useful web pages** A brief, but clear, description of the protocol including checksum may be found at the URL


A graphical simulation of the protocol is available at the URL


(Note that the control bits in the acknowledgment of the messages are switched.)

### 13.2 Gossiping girls

In this mini-project you are asked to model and analyze the following gossiping girls problem in UPPAAL.

**Problem description** A number of girls, say $G_1, G_2, \ldots, G_n$ ($n \geq 2$), initially know one distinct secret each. You can assume that the secrets are subsets of $\{1, \ldots, n\}$, and that initially girl $G_i$ knows $\{i\}$, for each $i \in \{1, \ldots, n\}$. Each girl has access to a phone that can be used to call another girl to share their secrets. Every time two girls talk to each other they always exchange all of the secrets they know. Thus, after the phone call, they both know all secrets they knew together before the phone call. The girls can communicate only in pairs (no conference calls are allowed), but it is possible that different pairs of girls talk concurrently.

Your tasks are as follows:

- Model the problem as a network of timed automata in UPPAAL, and use UPPAAL to find the smallest number of phone calls needed for four girls to know all secrets.

- Refine your model so that each phone call lasts exactly 60 seconds (for simplicity this time duration is independent of the number of exchanged secrets). Find the minimum time needed to solve the gossiping girls problem for four girls.
• Experiment with the UPPAAL search options breath-first and depth-first search and with the diagnostic trace settings fastest and shortest. Try to solve the problem for five girls.

Hints

• Design a single template for all girls.

• For each girl, remember the currently known secrets in a local integer variable. (Use a binary encoding such that if a girl knows the secrets of, for instance, girls 1 and 3 but does not know the secrets of girls 2 and 4, the value in the integer variable will be 0101 in binary—that is, 5 in decimal representation. You might find the operation |, for a bitwise OR, useful.)

• In order to model value passing when two girls make a phone call, you might want to read section 6.2 in (Behrmann et al., 2004).
Bibliography


BIBLIOGRAPHY


Index

A. Tarski, 90
  action, 25
  action prefixing operator, 10
  Amir Pnueli, 6
B. Knaster, 90
  branching
    bisimilarity, 72
    bisimulation, 72
CCS, 9–35
  semantics, 28
  syntax, 26
  value passing, 32–35
channel name, 25
characteristic formula, 143
choice operator, 11
complementary name, 25
complete lattice, 88
David Harel, 6
Donald Knuth, 4
Edinburgh Concurrency Workbench, 4
  equational system
    mutually recursive, 134
function
  fixed point of, 89
  monotonic, 89
  partial, 5
  relabelling, 26
greatest lower bound (glb), 87
Hennessy-Milner logic (HML)
  negation in, 109
  recursive
    game characterization, 129–131
    semantics, 123–126
    syntax, 122
    semantics, 103, 108
    syntax, 102
image finite process, 110
labelled transition system (LTS), 22
least upper bound (lub), 87
logics
  branching-time, 153
  linear-time, 153
modal depth, 115
observational equivalence, see weak bisimilarity
partially ordered set (poset), 85
process algebra, 7
process constant, see process name
process name, 25
reactive system, 6
relation
  binary, 38
  congruence, 39
  equivalence, 38
  weak trace, 164
  preorder, 38
ready simulation, 60
  reflexive, 38
  simulation, 60
    weak, 165
  symmetric, 38
  transitive, 38
restriction operator, 13
Robin Milner, 8

semantics, 4
state of an LTS, 22
  initial, 23
  reachable, 24
  start, 23
strong bisimilarity, see strong bisimulation equivalence
strong bisimulation, 43
equivalence, 43
game, 74
syntax, 4

test, 168
testable property, 169
Tony Hoare, 8
trace, 40
  completed, 42
  weak, 164
transition graph, see labelled transition system (LTS)
Uppaal, 4

weak bisimilarity, 65
weak bisimulation, 65
game, 79
weak transition relation, 64