Reactive Systems: Modelling, Specification and Verification

Luca Aceto[1 2] Anna Ingolfsdottir[1 2]
Kim G. Larsen[1] Jiří Srba[1]

[1]BRICS, Department of Computer Science, Aalborg University, 9220 Aalborg Ø, Denmark.
[2]Department of Computer Science, School of Science and Engineering, Reykjavík University, Iceland

Contents

List of Figures

List of Tables

Preface

This book is based on courses that have been held at Aalborg University and Reykjavík 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 to which this textbook aims at contributing.

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, twentieth 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, 1996).

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 develop models of the design to be built and subject them 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 twenty-first 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 its subject matter 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 much of the material presented in this textbook in several one semester courses at Aalborg University and at Reykjavík University, amongst others. 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 URL

`http://www.cs.aau.dk/rsbook/`.

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, 1998) 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 (Cleaveland, Parrow and Steffen, 1993) for the part of the course devoted to classic reactive systems, and, not surprisingly, UPPAAL (Behrmann, David and Larsen, 2004) 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 shall maintain a page with all of the supporting material, and other useful resources for students and instructors alike, at the URL

`http://www.cs.aau.dk/rsbook/`.

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 for 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 address

`rsbook@cs.aau.dk`

with subject line 'RS Book'.

Historical remarks and acknowledgments As already stated in this preface, we have used the material covered in this textbook in the present form for courses given at several institutions during the last five-six years. However, the story of its developments is much older, and goes back at least to 1986. During that year, the third author (Kim G. Larsen, then a freshly minted PhD graduate from Edinburgh University) took up an academic position at Aalborg University. He immediately began designing a course on the theory of concurrency—the branch of theoretical computer science that he had worked on during his doctoral studies under the supervision of Robin Milner. His aim was to use the course, and the accompanying set of notes and slides, to attract students to his research area by conveying his enthusiasm for it, as well as his belief that the theory of concurrency is important in applications. That material has stood the 'lecture room test' well, and still forms the basis for the first part of the book you now hold in your hands.

The development of those early courses was strongly influenced by Robin Milner's teaching and supervision that Kim G. Larsen enjoyed during his doctoral

studies in Edinburgh, and would not have been possible without them. Even though the other three authors were not students of Milner's themselves, the strong intellectual influence of his work and writings on their view of concurrency theory will be evident to the readers of this book. Indeed, the 'Edinburgh concurrency theory school' features prominently in the academic genealogy of each of the authors. For example, Rocco De Nicola and Matthew Hennessy had a strong influence on the view of concurrency theory and the work of Luca Aceto and/or Anna Ingolfsdottir, and Jiri Srba enjoyed the liberal supervision of Mogens Nielsen.

The material upon which the courses we have held at Aalborg University and elsewhere since the late 1980s were based has undergone gradual changes before reaching the present form. Over the years, the part of the course devoted to Milner's Calculus of Communicating Systems and its underlying theory has decreased, and so has the emphasis on some topics of mostly theoretical interest. At the same time, the course material has grown to include models and specification languages for real-time systems. The present material aims at offering a good balance between classic and real-time systems, and between the theory and its applications.

Overall, as already stated above, the students' appreciation of the theoretical material covered here has been greatly increased by the availability of software tools based on it. We thank all of the developers of the tools we use in our teaching; their work has made our subject matter come alive for our students, and has been instrumental in achieving whatever level of success we might have in our teaching based on this textbook.

This book was partly written while Luca Aceto was on leave from Aalborg University at Reykjavík University, Anna Ingolfsdottir 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 institutions for their hospitality and excellent working conditions. Luca Aceto and Anna Ingolfsdottir were partly supported by the project 'The Equational Logic of Parallel Processes' (nr. 060013021) of The Icelandic Research Fund. Jiří Srba received partial support from a grant of the Ministry of Education of the Czech Republic, project No. 1M0545.

We thank Silvio Capobianco, Pierre-Louis Curien, Gudmundur Hreidarson, Rocco De Nicola, Ralph Leibmann, MohammadReza Mousavi, Guy Vidal-Naquet 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 and 2006) at Aarhus University. The suggestions from Claus and his students helped us improve the text further. Moreover, Claus and Martin Mosegaard designed and implemented an excellent simulator for Milner's Calculus of Communicating Systems and the 'bisimulation-game game' that our students can use to experiment with the behaviour of processes written in this language, and to play the bisimulation game described in the textbook.

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 Ingolfsdottir dedicate this book to their son Róbert, to Anna's sons Logi and Kári, and to Luca's mother, Imelde Diomede Aceto. Kim G. Larsen dedicates the book to his wife Merete and to his two daughters Mia and Trine. Finally, Jiří Srba dedicates the book to his parents Jaroslava and Jiří, and to his wife Vanda.

Luca Aceto and Anna Ingolfsdottir, Reykjav´ık, Iceland
Kim G. Larsen and Jiř´ı 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
- Hennessy-Milner Logic (HML) (Hennessy and Milner, 1985) and its extension with recursive definitions of formulae (Larsen, 1990).

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 (Yi, 1990; Yi, 1991a; Yi, 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).

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 et al., 1993) and the model checker for real-time systems UPPAAL (Behrmann et al., 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

$$U = \textbf{while}\ \text{true}\ \textbf{do skip} \ ,$$

where we use **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

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, 2005). (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, or communication channels, namely coffee, $\overline{\text{coin}}$ and $\overline{\text{pub}}$. The port coffee is used for input, whereas the ports $\overline{\text{coin}}$ and $\overline{\text{pub}}$ 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 $\overline{\text{coin}}$ 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

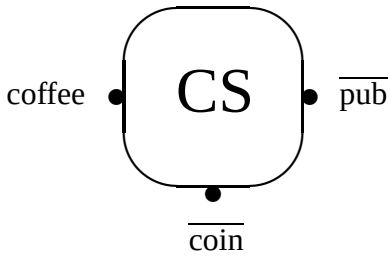


Figure 2.1: The interface for process CS

all is the process $\mathbf{0}$ (read 'nil'). This is the most boring process imaginable, as it performs no action whatsoever. The process $\mathbf{0}$ 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 $\mathbf{0}$ and action prefixing are a match and a complex match, described by the expressions

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

respectively. Intuitively, a match is a process that dies when stricken (i.e., that becomes the process $\mathbf{0}$ 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 $\overline{\text{pub}}$, 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:

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

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

behaviour of an everlasting clock thus:

$$\text{Clock} \stackrel{\text{def}}{=} \text{tick.Clock} \ .$$

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

$$\begin{aligned} \text{Clock} &\stackrel{\text{def}}{=} \text{tick.Clock} \\ &= \text{tick.tick.Clock} \\ &= \text{tick.tick.tick.Clock} \\ &\vdots \\ &= \underbrace{\text{tick}.\ldots.\text{tick}}_{n\text{-times}}.\text{Clock} \ , \end{aligned}$$

for each positive integer n.

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

$$\text{CM} \stackrel{\text{def}}{=} \text{coin}.\overline{\text{coffee}}.\text{CM} \ . \tag{2.1}$$

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:

$$\text{CTM} \stackrel{\text{def}}{=} \text{coin}.(\overline{\text{coffee}}.\text{CTM} + \overline{\text{tea}}.\text{CTM}) \ . \tag{2.2}$$

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:

> If P and Q 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 $(\mathcal{Q}, A, \delta, q_0)$, where*

- *$\mathcal{Q}$ is a finite set of states,*
- *A is a finite set of labels,*
- *$q_0 \in \mathcal{Q}$ is the start state and*
- *$\delta : \mathcal{Q} \times A \to 2^{\mathcal{Q}}$ 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

$$\mathrm{CS} \stackrel{\text{def}}{=} \overline{\mathrm{pub}}.\overline{\mathrm{coin}}.\mathrm{coffee}.\mathrm{CS} \ . \tag{2.3}$$

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 $\mathrm{CM} \mid \mathrm{CS}$ 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

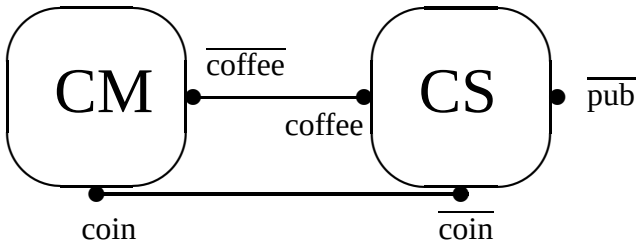


Figure 2.2: The interface for process CM | CS

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 $P \mid Q$ 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 $\backslash$coin and $\backslash$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:

$$\text{SmUni} \stackrel{\text{def}}{=} (\text{CM} \mid \text{CS}) \setminus \text{coin} \setminus \text{coffee} \ . \tag{2.4}$$

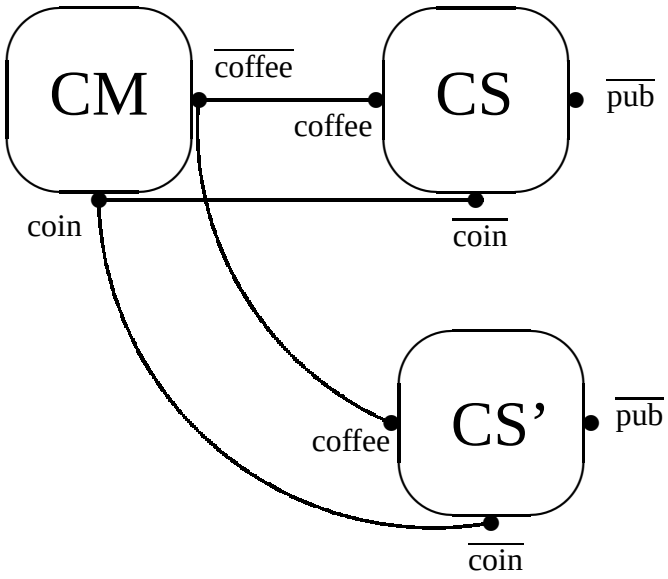


Figure 2.3: The interface for process CM | CS | CS′

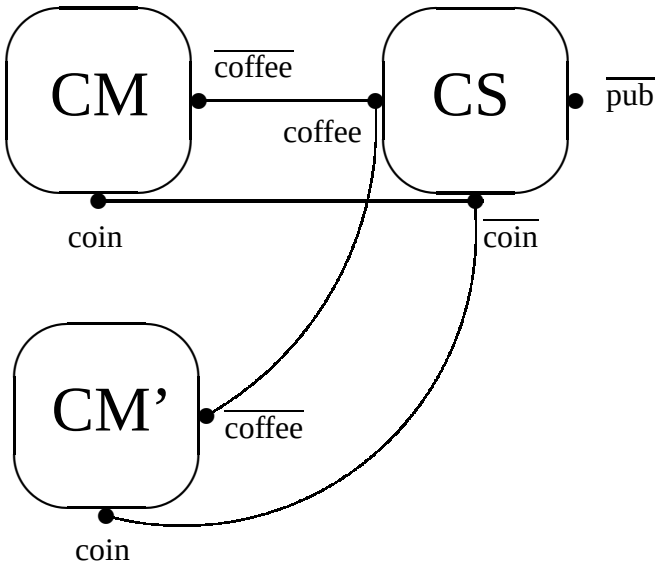


Figure 2.4: The interface for process CM | CS | CM′

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

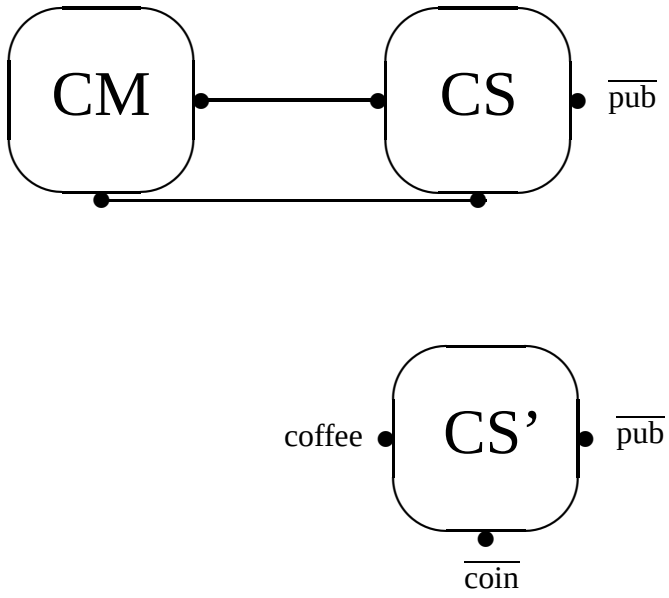


Figure 2.5: The interface for process SmUni | CS′

processes

$$\begin{aligned} \mathrm{CHM} &\stackrel{\mathrm{def}}{=} \mathrm{coin}.\overline{\mathrm{choc}}.\mathrm{CHM} \\ \mathrm{DFM} &\stackrel{\mathrm{def}}{=} \mathrm{coin}.\overline{\mathrm{figs}}.\mathrm{DFM} \\ \mathrm{CRM} &\stackrel{\mathrm{def}}{=} \mathrm{coin}.\overline{\mathrm{crisps}}.\mathrm{CRM} \ . \end{aligned}$$

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

$$\mathrm{VM} \stackrel{\mathrm{def}}{=} \mathrm{coin}.\overline{\mathrm{item}}.\mathrm{VM} \ .$$

All of the aforementioned specific vending machines may be obtained as appropriate 'renamings' of VM. For example,

$$\mathrm{CHM} \stackrel{\mathrm{def}}{=} \mathrm{VM}[\mathrm{choc}/\mathrm{item}] \ ,$$

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

> If P is a process and f is a function from labels to labels satisfying certain requirements that will be made precise in Section 2.2, then $P[f]$ 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).

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

$$\begin{aligned} \mathrm{CS} &\stackrel{\text{def}}{=} \overline{\mathrm{pub}}.\mathrm{CS}_1 \\ \mathrm{CS}_1 &\stackrel{\text{def}}{=} \overline{\mathrm{coin}}.\mathrm{CS}_2 \\ \mathrm{CS}_2 &\stackrel{\text{def}}{=} \mathrm{coffee}.\mathrm{CS} \end{aligned}$$

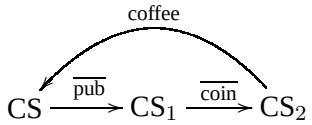
Table 2.1: An alternative formulation for process CS

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 $\overline{\mathrm{pub}}$ and evolve into a process whose behaviour is described by the CCS expression CS_1 in doing so. Process CS_1 can then output a coin, thereby evolving into a process whose behaviour is described by the CCS expression CS_2. Finally, this process can receive coffee as input, and behave like our good old CS all over again. Thus the processes CS, CS_1 and CS_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

$$\mathrm{CS} \stackrel{\overline{\mathrm{pub}}}{\rightarrow} \mathrm{CS}_1 \ ,$$

which says that CS can perform action $\overline{\mathrm{pub}}$, and become CS_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:

$$\begin{aligned} \mathrm{CM} &\stackrel{\text{def}}{=} \mathrm{coin}.\mathrm{CM}_1 \\ \mathrm{CM}_1 &\stackrel{\text{def}}{=} \overline{\mathrm{coffee}}.\mathrm{CM} \ . \end{aligned}$$

Note that the computer scientist is willing to output a coin in state CS_1, as witnessed by the transition

$$\mathrm{CS}_1 \stackrel{\overline{\mathrm{coin}}}{\rightarrow} \mathrm{CS}_2 \ ,$$

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

$$\mathrm{CM} \stackrel{\mathrm{coin}}{\rightarrow} \mathrm{CM}_1 \ .$$

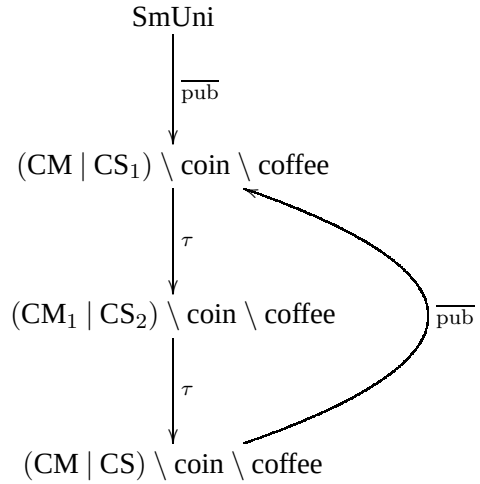
Therefore, when put in parallel with one another, these two processes may communicate and change state simultaneously. The result of the communication should be described as a state transition of the form

$$\mathrm{CM} \mid \mathrm{CS}_1 \stackrel{?}{\rightarrow} \mathrm{CM}_1 \mid \mathrm{CS}_2 \ .$$

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 τ. So the above transition is indicated by

$$\mathrm{CM} \mid \mathrm{CS}_1 \stackrel{\tau}{\rightarrow} \mathrm{CM}_1 \mid \mathrm{CS}_2 \ .$$

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 τ actions are supposed to be unobservable, the following process seems to be an appropriate high level specification of the behaviour exhibited by process SmUni:

$$\mathrm{Spec} \stackrel{\mathrm{def}}{=} \overline{\mathrm{pub}}.\mathrm{Spec} \ .$$

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 $p \stackrel{a}{\rightarrow} p'$. 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

$$p \stackrel{\text{tea}}{\longrightarrow} p_1 \text{ and } p \stackrel{\text{coffee}}{\longrightarrow} p_2 \ .$$

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

$$p_1 \stackrel{1€}{\longrightarrow} p_3 \text{ and } p_2 \stackrel{2€}{\longrightarrow} p_3 \ ,$$

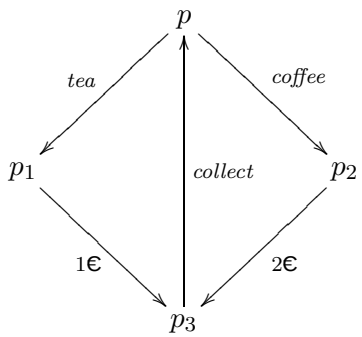
whose target state p_3 records that the machine has received payment for the chosen drink.

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

$$p_3 \stackrel{\text{collect}}{\longrightarrow} p \ .$$

♦

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.



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.

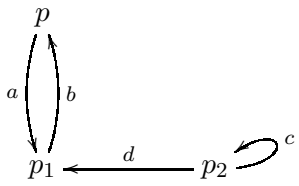
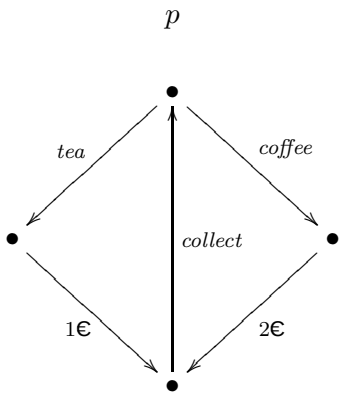


Figure 2.6: Labelled transition system with initial state p



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 $(\mathsf{Proc}, \mathsf{Act}, \{\stackrel{\alpha}{\rightarrow} \mid \alpha \in \mathsf{Act}\})$, where:

- Proc is a set of *states* (or *processes*);
- Act is a set of *actions* (or *labels*);
- $\stackrel{\alpha}{\rightarrow} \subseteq \mathsf{Proc} \times \mathsf{Proc}$ is a *transition relation*, for every $\alpha \in \mathsf{Act}$. As usual, we shall use the more suggestive notation $s \stackrel{\alpha}{\rightarrow} s'$ in lieu of $(s, s') \in \stackrel{\alpha}{\rightarrow}$, and write $s \stackrel{\alpha}{\not\rightarrow}$ (read 's refuses a') iff $s \stackrel{\alpha}{\rightarrow} s'$ 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:

$$\begin{aligned}
\mathsf{Proc} &= \{\mathrm{SmUni}, (\mathrm{CM} \mid \mathrm{CS}_1) \setminus \mathrm{coin} \setminus \mathrm{coffee}, (\mathrm{CM}_1 \mid \mathrm{CS}_2) \setminus \mathrm{coin} \setminus \mathrm{coffee}, \\
&\qquad (\mathrm{CM} \mid \mathrm{CS}) \setminus \mathrm{coin} \setminus \mathrm{coffee}\} \\
\mathsf{Act} &= \{\overline{\mathrm{pub}}, \tau\} \\
\xrightarrow{\overline{\mathrm{pub}}} &= \{\big(\mathrm{SmUni}, (\mathrm{CM} \mid \mathrm{CS}_1) \setminus \mathrm{coin} \setminus \mathrm{coffee}\big), \\
&\qquad \big((\mathrm{CM} \mid \mathrm{CS}) \setminus \mathrm{coin} \setminus \mathrm{coffee}, (\mathrm{CM} \mid \mathrm{CS}_1) \setminus \mathrm{coin} \setminus \mathrm{coffee}\big)\} \ , \text{ and} \\
\xrightarrow{\tau} &= \{\big((\mathrm{CM} \mid \mathrm{CS}_1) \setminus \mathrm{coin} \setminus \mathrm{coffee}, (\mathrm{CM}_1 \mid \mathrm{CS}_2) \setminus \mathrm{coin} \setminus \mathrm{coffee}\big), \\
&\qquad \big((\mathrm{CM}_1 \mid \mathrm{CS}_2) \setminus \mathrm{coin} \setminus \mathrm{coffee}, (\mathrm{CM} \mid \mathrm{CS}) \setminus \mathrm{coin} \setminus \mathrm{coffee}\big)\} \ .
\end{aligned}$$

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 $\xrightarrow{\alpha}$ are presented as a ternary relation $\rightarrow \subseteq \mathsf{Proc} \times \mathsf{Act} \times \mathsf{Proc}$ and we write $s \xrightarrow{\alpha} s'$ whenever $(s, \alpha, s') \in \rightarrow$. 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 Act^* (the set of all finite strings over Act including the empty string ε). The definition is as follows:
  - $s \xrightarrow{\varepsilon} s$ for every $s \in \mathsf{Proc}$, and
  - $s \xrightarrow{\alpha w} s'$ iff there is a state $t \in \mathsf{Proc}$ such that $s \xrightarrow{\alpha} t$ and $t \xrightarrow{w} s'$, for every $s, s' \in \mathsf{Proc}$, $\alpha \in \mathsf{Act}$ and $w \in \mathsf{Act}^*$.

  In other words, if $w = \alpha_1\alpha_2 \cdots \alpha_n$ for $\alpha_1, \alpha_2 \ldots, \alpha_n \in \mathsf{Act}$ then we write $s \xrightarrow{w} s'$ whenever there exist states $s_0, s_1, \ldots, s_{n-1}, s_n \in \mathsf{Proc}$ such that

  $$s = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} s_3 \xrightarrow{\alpha_4} \cdots \xrightarrow{\alpha_{n-1}} s_{n-1} \xrightarrow{\alpha_n} s_n = s' \ .$$

  For the transition system in Figure 2.6 we have, for example, that $p \xrightarrow{\varepsilon} p$, $p \xrightarrow{ab} p$ and $p_1 \xrightarrow{bab} p$.

- We write $s \rightarrow s'$ whenever there is an action $\alpha \in \mathsf{Act}$ such that $s \stackrel{\alpha}{\rightarrow} s'$.

  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 $s \stackrel{\alpha}{\rightarrow}$ meaning that there is some $s' \in \mathsf{Proc}$ such that $s \stackrel{\alpha}{\rightarrow} s'$.

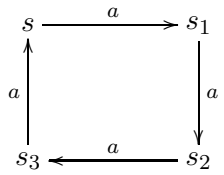
  For the transition system in Figure 2.6 we have, for instance, that $p \stackrel{a}{\rightarrow}$ and $p_1 \stackrel{b}{\rightarrow}$.

- We write $s \rightarrow^* s'$ iff $s \stackrel{w}{\rightarrow} s'$ for some $w \in \mathsf{Act}^*$. 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.*



- *Define the labelled transition system as a triple* $(\mathsf{Proc}, \mathsf{Act}, \{\stackrel{\alpha}{\rightarrow} | \ \alpha \in \mathsf{Act}\})$.
- *What is the reflexive closure of the binary relation* $\stackrel{a}{\rightarrow}$*? (A drawing is fine.)*
- *What is the symmetric closure of the binary relation* $\stackrel{a}{\rightarrow}$*? (A drawing is fine.)*
- *What is the transitive closure of the binary relation* $\stackrel{a}{\rightarrow}$*? (A drawing is fine.)*

♦

Definition 2.2 [Reachable states] Let $T = (\mathsf{Proc}, \mathsf{Act}, \{\stackrel{\alpha}{\rightarrow} | \ \alpha \in \mathsf{Act}\})$ be a labelled transition system, and let $s \in \mathsf{Proc}$ be its initial state. We say that $s' \in \mathsf{Proc}$ 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 $\{p, p_1\}$.

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 τ, 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

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

be the set of *actions*. In our formal developments, we shall use a, b to range over $\mathcal{L}$ and α as a typical member of 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:

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

where

- K is a process name in $\mathcal{K}$;
- α is an action in Act;
- I is a possibly infinite index set;
- $f : \mathsf{Act} \to \mathsf{Act}$ is a *relabelling function* satisfying the following constraints:

$$\begin{array}{rcl} f(\tau) & = & \tau \text{ and} \\ f(\bar{a}) & = & \overline{f(a)} \text{ for each label } a \ ; \end{array}$$

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

We write $\mathbf{0}$ for an empty sum of processes, i.e.,

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

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

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

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 $\backslash \{a\}$.

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 $a.\mathbf{0} \mid b.P \setminus L + c.\mathbf{0}$ stands for

$$((a.\mathbf{0}) \mid (b.(P \setminus L))) + (c.\mathbf{0}) \ .$$

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.*

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

♦

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:

$$\text{Counter}_0 \stackrel{\text{def}}{=} \text{up.Counter}_1 \tag{2.5}$$

$$\text{Counter}_n \stackrel{\text{def}}{=} \text{up.Counter}_{n+1} + \text{down.Counter}_{n-1} \quad (n > 0) \ . \tag{2.6}$$

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 $P \stackrel{\alpha}{\rightarrow} Q$ holds for CCS expressions P, Q and action α if, and only if, it can be proven using the rules in Table 2.2.

A rule like

$$\frac{}{\alpha.P \stackrel{\alpha}{\rightarrow} P}$$

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 $\alpha.P \stackrel{\alpha}{\rightarrow} P$ (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 $\alpha.P \stackrel{\alpha}{\rightarrow} P$. As an example, we have that the following transition

$$\overline{\text{pub}}.\text{CS}_1 \stackrel{\overline{\text{pub}}}{\rightarrow} \text{CS}_1 \tag{2.7}$$

is provable using the above rule for action prefixing.

On the other hand, a rule like

$$\frac{P \stackrel{\alpha}{\rightarrow} P'}{K \stackrel{\alpha}{\rightarrow} P'} \quad K \stackrel{\text{def}}{=} P$$

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 $P \stackrel{\alpha}{\rightarrow} P'$. Using this rule, pattern matching and transition (2.7), we can prove the transition

$$\text{CS} \stackrel{\overline{\text{pub}}}{\rightarrow} \text{CS}_1 \ ,$$

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 $K \stackrel{\text{def}}{=} P$, 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.

$$\frac{P \stackrel{\alpha}{\rightarrow} P'}{P \setminus L \stackrel{\alpha}{\rightarrow} P' \setminus L} \quad \alpha, \bar{\alpha} \notin L$$

$$\text{ACT} \quad \frac{}{\alpha.P \xrightarrow{\alpha} P}$$

$$\text{SUM}_j \quad \frac{P_j \xrightarrow{\alpha} P'_j}{\sum_{i \in I} P_i \xrightarrow{\alpha} P'_j} \quad \text{where } j \in I$$

$$\text{COM1} \quad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

$$\text{COM2} \quad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\text{COM3} \quad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\text{RES} \quad \frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L} \quad \text{where } \alpha, \bar{\alpha} \notin L$$

$$\text{REL} \quad \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]}$$

$$\text{CON} \quad \frac{P \xrightarrow{\alpha} P'}{K \xrightarrow{\alpha} P'} \quad \text{where } K \stackrel{\text{def}}{=} P$$

Table 2.2: SOS rules for CCS ($\alpha \in \mathsf{Act}$, $a \in \mathcal{L}$)

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

$$(\text{coffee.CS}) \setminus \text{coffee} \stackrel{\text{coffee}}{\rightarrow} \text{CS} \setminus \text{coffee} \ .$$

Finally, note that, when considering the binary version of the summation operator, the family of rules SUM_j reduces to the following two rules.

$$\text{SUM}_1 \quad \frac{P_1 \stackrel{\alpha}{\rightarrow} P_1'}{P_1 + P_2 \stackrel{\alpha}{\rightarrow} P_1'} \qquad \text{SUM}_2 \quad \frac{P_2 \stackrel{\alpha}{\rightarrow} P_2'}{P_1 + P_2 \stackrel{\alpha}{\rightarrow} P_2'}$$

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

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

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

$$\text{C} \stackrel{\text{up}}{\rightarrow} \text{C} \mid \text{down}.\mathbf{0} \ . \tag{2.9}$$

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

$$\text{C} \mid \text{down}.\mathbf{0} \stackrel{\text{down}}{\rightarrow} \text{C} \mid \mathbf{0} \ .$$

Since it is reasonable to expect that the process $\text{C} \mid \mathbf{0}$ 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

$$\text{C} \mid \text{down}.\mathbf{0} \stackrel{\text{up}}{\rightarrow} (\text{C} \mid \text{down}.\mathbf{0}) \mid \text{down}.\mathbf{0}$$

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 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 SmUni defined by equation 2.4 on page 13. (Use the definition of CS in Table 2.1.)* ♦

Exercise 2.8 *Assume that* $A \stackrel{def}{=} b.a.B$. *By using the SOS rules for CCS prove the existence of the following transitions:*

- $(A \mid \overline{b}.\mathbf{0}) \setminus \{b\} \stackrel{\tau}{\rightarrow} (a.B \mid \mathbf{0}) \setminus \{b\}$,
- $(A \mid \overline{b}.a.B) + (\overline{b}.A)[a/b] \stackrel{\overline{b}}{\rightarrow} (A \mid a.B)$, *and*
- $(A \mid \overline{b}.a.B) + (\overline{b}.A)[a/b] \stackrel{\overline{a}}{\rightarrow} A[a/b]$.

♦

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

$$A \stackrel{def}{=} (a.A) \setminus b \ .$$

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*

$$A \stackrel{def}{=} (a_0.A)[f]$$

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 Mutex*$_1$ *whose behaviour is given by the defining equation*

$$\begin{aligned} \mathit{Mutex}_1 &\stackrel{def}{=} (\mathit{User} \mid \mathit{Sem}) \setminus \{p, v\} \\ \mathit{User} &\stackrel{def}{=} \bar{p}.\mathit{enter}.\mathit{exit}.\bar{v}.\mathit{User} \\ \mathit{Sem} &\stackrel{def}{=} p.v.\mathit{Sem} \ . \end{aligned}$$

2. *Draw the transition graph for the process name Mutex$_2$ whose behaviour is given by the defining equation*

$$\mathit{Mutex}_2 \stackrel{\mathit{def}}{=} ((\mathit{User} \mid \mathit{Sem}) \mid \mathit{User}) \setminus \{p, v\} \ ,$$

where User and Sem are defined as before.

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

$$\mathit{User} \stackrel{\mathit{def}}{=} \bar{p}.\mathit{enter}.\bar{v}.\mathit{exit}.\mathit{User} \ ?$$

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

$$\mathit{FMutex} \stackrel{\mathit{def}}{=} ((\mathit{User} \mid \mathit{Sem}) \mid \mathit{FUser}) \setminus \{p, v\} \ ,$$

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

$$\mathit{FUser} \stackrel{\mathit{def}}{=} \bar{p}.\mathit{enter}.(\mathit{exit}.\bar{v}.\mathit{FUser} + \mathit{exit}.\bar{v}.\mathbf{0}) \ .$$

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.

- 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:

$$\begin{aligned} \mathrm{B} &\stackrel{\text{def}}{=} \mathrm{in}(x).\mathrm{B}(x) \\ \mathrm{B}(x) &\stackrel{\text{def}}{=} \overline{\mathrm{out}}(x+1).\mathrm{B} \ . \end{aligned}$$

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, $\mathrm{B}(x)$. 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 $\mathrm{B}(n)$—that is, like a full one-place buffer storing the value n. The behaviour of the process $\mathrm{B}(n)$ 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 '$\overline{\mathrm{out}}(x+1)$' 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

$$\frac{}{a(x).P \stackrel{a(n)}{\rightarrow} P[n/x]} \quad n \geq 0$$

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.

$$\frac{}{\bar{a}(e).P \stackrel{\bar{a}(n)}{\rightarrow} P} \quad n \text{ is the result of evaluating } e$$

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 $\mathrm{A}(x_1, \ldots, x_n)$, 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.

$$\frac{P[v_1/x_1, \ldots, v_n/x_n] \stackrel{\alpha}{\rightarrow} P'}{\mathrm{A}(e_1, \ldots, e_n) \stackrel{\alpha}{\rightarrow} P'} \quad \mathrm{A}(x_1, \ldots, x_n) \stackrel{\text{def}}{=} P \text{ and each } e_i \text{ has value } v_i$$

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 $\mathrm{A}(x_1, \ldots, x_n)$ with $y = x_i$ for some $1 \leq i \leq n$. For instance, the expression

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

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:

$$\begin{aligned} \mathrm{Pred} &\stackrel{\text{def}}{=} \mathrm{in}(x).\mathrm{Pred}(x) \\ \mathrm{Pred}(x) &\stackrel{\text{def}}{=} \textbf{if } x = 0 \textbf{ then } \overline{\mathrm{out}}(0).\mathrm{Pred} \textbf{ else } \overline{\mathrm{out}}(x-1).\mathrm{Pred} \ . \end{aligned}$$

We expect $\mathrm{Pred}(0)$ to output the value 0 on channel 'out', and $\mathrm{Pred}(n+1)$ 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

$$\frac{P \stackrel{\alpha}{\rightarrow} P'}{\textbf{if } \mathrm{bexp} \textbf{ then } P \textbf{ else } Q \stackrel{\alpha}{\rightarrow} P'} \quad \mathrm{bexp\ is\ true}$$

and

$$\frac{Q \stackrel{\alpha}{\rightarrow} Q'}{\textbf{if } \mathrm{bexp} \textbf{ then } P \textbf{ else } Q \stackrel{\alpha}{\rightarrow} Q'} \quad \mathrm{bexp\ is\ false} \ .$$

Exercise 2.12 *Consider a one place buffer defined by*

$$\begin{aligned} \mathit{Cell} &\stackrel{\mathit{def}}{=} \mathit{in}(x).\mathit{Cell}(x) \\ \mathit{Cell}(x) &\stackrel{\mathit{def}}{=} \overline{\mathit{out}}(x).\mathit{Cell} \ . \end{aligned}$$

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. ♦

Exercise 2.13 *Consider the process B defined thus:*

$$\begin{aligned} B &\stackrel{def}{=} \mathit{push}(x).(C(x)^\frown B) + \mathit{empty}.B \\ C(x) &\stackrel{def}{=} \mathit{push}(y).(C(y)^\frown C(x)) + \overline{\mathit{pop}}(x).D \\ D &\stackrel{def}{=} o(x).C(x) + \bar{e}.B \ , \end{aligned}$$

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

$$P^\frown Q = (P[p'/p, e'/e, o'/o] \mid Q[p'/\mathit{push}, e'/\mathit{empty}, o'/\mathit{pop}]) \setminus \{p', o', e'\} \ .$$

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 equivalences

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

$$\mathrm{Spec} \stackrel{\mathrm{def}}{=} \overline{\mathrm{pub}}.\mathrm{Spec} \ ,$$

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*. ⧫

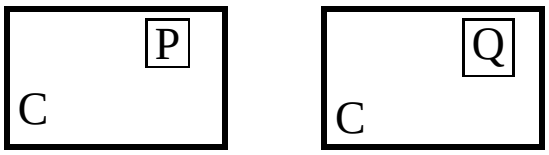
An equivalence relation is therefore a more abstract version of the notion of equality that we are familiar with since elementary school.

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 $M_2 = \{(n, m) \mid n, m \in \mathbb{N},\ n \mod 2 = m \mod 2\}$.*

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



$C(P)$ $C(Q)$

Figure 3.1: $P\ R\ Q$ implies that $C[P]\ R\ C[Q]$

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 $\alpha_1 \cdots \alpha_k \in \mathsf{Act}^*$ ($k \geq 0$) such that there exists a sequence of transitions

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

for some $P_1, \ldots, P_k$. We write $Traces(P)$ for the collection of all traces of P. Since $Traces(P)$ 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

$$\text{if } P \text{ and } Q \text{ are behaviourally equivalent, then } Traces(P) = Traces(Q) \ . \quad (3.1)$$

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

$$\mathrm{CTM}' \stackrel{\mathrm{def}}{=} \mathrm{coin}.\overline{\mathrm{coffee}}.\mathrm{CTM}' + \mathrm{coin}.\overline{\mathrm{tea}}.\mathrm{CTM}' \ . \tag{3.2}$$

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

$$\mathrm{CA} \stackrel{\mathrm{def}}{=} \overline{\mathrm{coin}}.\mathrm{coffee}.\mathrm{CA} \ .$$

Consider now the terms

$$(\mathrm{CA} \mid \mathrm{CTM}) \setminus \{\mathrm{coin}, \mathrm{coffee}, \mathrm{tea}\}$$

and

$$(\mathrm{CA} \mid \mathrm{CTM}') \setminus \{\mathrm{coin}, \mathrm{coffee}, \mathrm{tea}\}$$

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 τ-labelled transitions, whereas the second term can deadlock thus:

$$(\mathrm{CA} \mid \mathrm{CTM}') \setminus \{\mathrm{coin}, \mathrm{coffee}, \mathrm{tea}\} \stackrel{\tau}{\rightarrow} (\mathrm{coffee}.\mathrm{CA} \mid \overline{\mathrm{tea}}.\mathrm{CTM}') \setminus \{\mathrm{coin}, \mathrm{coffee}, \mathrm{tea}\} \ .$$

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 $\alpha_1 \cdots \alpha_k \in \mathsf{Act}^*$ ($k \geq 0$) such that there exists a sequence of transitions*

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

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*

   $$(\mathit{CA} \mid \mathit{CTM}) \setminus \{\mathit{coin}, \mathit{coffee}, \mathit{tea}\}$$

   and

   $$(\mathit{CA} \mid \mathit{CTM}') \setminus \{\mathit{coin}, \mathit{coffee}, \mathit{tea}\}$$

   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

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 α is an action:

- if $s_1 \stackrel{\alpha}{\rightarrow} s_1'$, then there is a transition $s_2 \stackrel{\alpha}{\rightarrow} s_2'$ such that $s_1' \,\mathcal{R}\, s_2'$;

- if $s_2 \stackrel{\alpha}{\rightarrow} s_2'$, then there is a transition $s_1 \stackrel{\alpha}{\rightarrow} s_1'$ 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.

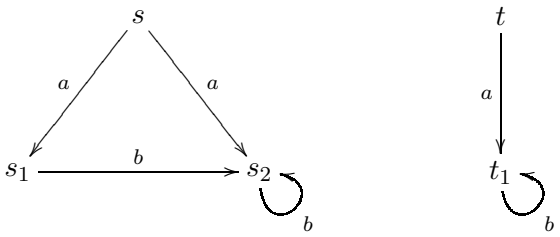
Example 3.1 Consider the labelled transition system

$$(\mathsf{Proc}, \mathsf{Act}, \{\stackrel{\alpha}{\rightarrow} | \ \alpha \in \mathsf{Act}\}) \ ,$$

where

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

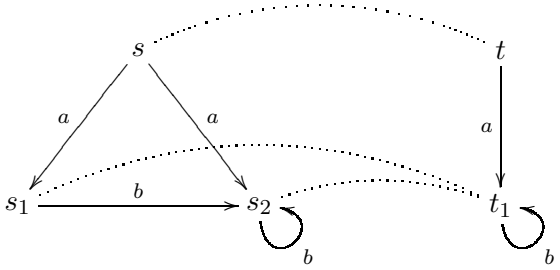
Here is a graphical representation of this labelled transition system.



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.



Obviously, $(s,t) \in \mathcal{R}$. We have to show that $\mathcal{R}$ 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 $\mathcal{R}$, 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 $\mathcal{R}$ 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 $(s_1,t_1) \in \mathcal{R}$,
    - $s \xrightarrow{a} s_2$ can be matched by $t \xrightarrow{a} t_1$ and $(s_2,t_1) \in \mathcal{R}$, 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 $(s_2,t_1) \in \mathcal{R}$ (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 $(s_2,t_1) \in \mathcal{R}$, 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 $(s_2,t_1) \in \mathcal{R}$, 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 $(s_2,t_1) \in \mathcal{R}$, 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 $(s_2,t_1) \in \mathcal{R}$, 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

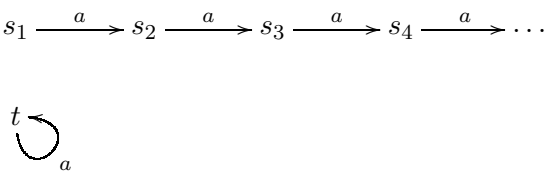
$$\mathcal{R} = \{(s_1, s_2), (s_2, s_2)\} \ .$$

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 $(\mathsf{Proc}, \mathsf{Act}, \{\stackrel{\alpha}{\rightarrow} \mid \alpha \in \mathsf{Act}\})$ where

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

Here is a graphical representation of this labelled transition system.



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 CTM and CTM′ are *not* strongly bisimilar thus. Assume, towards a contradiction, that CTM and CTM′ are strongly bisimilar. This means that there is a strong bisimulation $\mathcal{R}$ such that

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

Recall that

$$\mathrm{CTM}' \stackrel{\mathrm{coin}}{\rightarrow} \overline{\mathrm{tea}}.\mathrm{CTM}' \ .$$

So, by the second requirement in Definition 3.2, there must be a transition

$$\text{CTM} \stackrel{\text{coin}}{\rightarrow} P$$

for some process P such that $P \mathcal{R} \overline{\text{tea}}.\text{CTM}'$. 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 $\overline{\text{coffee}}.\text{CTM} + \overline{\text{tea}}.\text{CTM}$. So we are requiring that

$$(\overline{\text{coffee}}.\text{CTM} + \overline{\text{tea}}.\text{CTM}) \mathcal{R} \overline{\text{tea}}.\text{CTM}' \ .$$

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

$$\overline{\text{coffee}}.\text{CTM} + \overline{\text{tea}}.\text{CTM} \stackrel{\overline{\text{coffee}}}{\rightarrow} \text{CTM} \ ,$$

but $\overline{\text{tea}}.\text{CTM}'$ 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:

$$\begin{aligned} P &\stackrel{\text{def}}{=} a.P_1 + b.P_2 \\ P_1 &\stackrel{\text{def}}{=} c.P \\ P_2 &\stackrel{\text{def}}{=} c.P \end{aligned}$$

and

$$\begin{aligned} Q &\stackrel{\text{def}}{=} a.Q_1 + b.Q_2 \\ Q_1 &\stackrel{\text{def}}{=} c.Q_3 \\ Q_2 &\stackrel{\text{def}}{=} c.Q_3 \\ Q_3 &\stackrel{\text{def}}{=} a.Q_1 + b.Q_2 \ . \end{aligned}$$

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:*

$$\begin{aligned} P &\stackrel{def}{=} a.P_1 \\ P_1 &\stackrel{def}{=} b.P + c.P \end{aligned}$$

and

$$\begin{aligned} Q &\stackrel{def}{=} a.Q_1 \\ Q_1 &\stackrel{def}{=} b.Q_2 + c.Q \\ Q_2 &\stackrel{def}{=} a.Q_3 \\ Q_3 &\stackrel{def}{=} b.Q + c.Q_2 \ . \end{aligned}$$

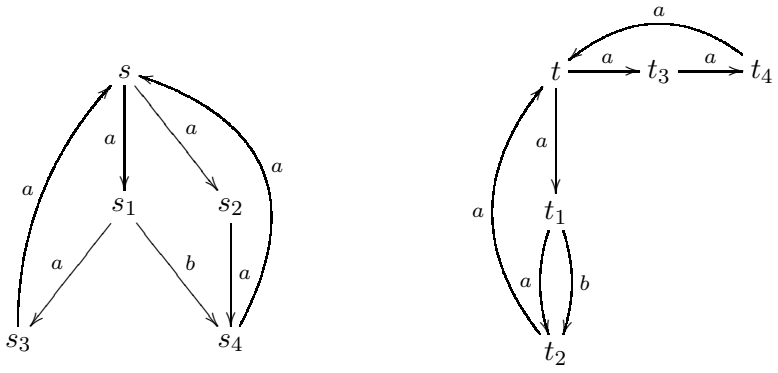
Show that $P \sim Q$ holds by exhibiting an appropriate strong bisimulation. ♦

Exercise 3.4 *Consider the processes*

$$\begin{aligned} P &\stackrel{def}{=} a.(b.\mathbf{0} + c.\mathbf{0}) \quad \textit{and} \\ Q &\stackrel{def}{=} a.b.\mathbf{0} + a.c.\mathbf{0} \ . \end{aligned}$$

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 α,

   - if $s_1 \stackrel{\alpha}{\rightarrow} s_1'$, then there is a transition $s_2 \stackrel{\alpha}{\rightarrow} s_2'$ such that $s_1' \sim s_2'$;
   - if $s_2 \stackrel{\alpha}{\rightarrow} s_2'$, then there is a transition $s_1 \stackrel{\alpha}{\rightarrow} s_1'$ such that $s_1' \sim s_2'$.

Proof: Consider an LTS $(\mathsf{Proc}, \mathsf{Act}, \{\stackrel{\alpha}{\rightarrow} \mid \alpha \in \mathsf{Act}\})$. We prove each of the statements in turn.

1. In order to show that $\sim$ is an equivalence relation over the set of states 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 $s \in \mathsf{Proc}$. It is not hard to see that the *identity relation*

   $$\mathcal{I} = \{(s, s) \mid s \in \mathsf{Proc}\}$$

   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 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 $\mathcal{R}$ that contains the pair of states (s_1, s_2). Consider now the relation

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

   You should now be able to convince yourselves that the pair (s_2, s_1) is contained in $\mathcal{R}^{-1}$, 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 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 $\mathcal{R}$ and $\mathcal{R}'$ that contain the pairs of states (s_1, s_2) and (s_2, s_3), respectively. Consider now the relation

   $$\mathcal{S} = \{(s_1', s_3') \mid (s_1', s_2') \in \mathcal{R} \text{ and } (s_2', s_3') \in \mathcal{R}', \text{ for some } s_2'\} \ .$$

The pair (s_1, s_3) is contained in $\mathcal{S}$. (Why?) Moreover, using that $\mathcal{R}$ and $\mathcal{R}'$ are bisimulations, you should be able to show that so is $\mathcal{S}$. 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 Proc. To this end, observe, first of all, that the definition of $\sim$ states that

$$\sim \;=\; \bigcup\{\mathcal{R} \mid \mathcal{R} \text{ is a bisimulation}\} \ .$$

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

$$(s_1, s_2) \in \bigcup\{\mathcal{R} \mid \mathcal{R} \text{ is a bisimulation}\} \text{ and } s_1 \stackrel{\alpha}{\to} s_1' \ , \qquad (3.3)$$

then there is a state s_2' such that $s_2 \stackrel{\alpha}{\to} s_2'$ and

$$(s_1', s_2') \in \bigcup\{\mathcal{R} \mid \mathcal{R} \text{ is a bisimulation}\} \ .$$

Assume, therefore, that (3.3) holds. Since

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

there is a bisimulation $\mathcal{R}$ that contains the pair (s_1, s_2). As $\mathcal{R}$ is a bisimulation and $s_1 \stackrel{\alpha}{\to} s_1'$, we have that there is a state s_2' such that $s_2 \stackrel{\alpha}{\to} s_2'$ and $(s_1', s_2') \in \mathcal{R}$. Observe now that the pair (s_1', s_2') is also contained in

$$\bigcup\{\mathcal{R} \mid \mathcal{R} \text{ is a bisimulation}\} \ .$$

Hence, we have argued that there is a state s_2' such that $s_2 \stackrel{\alpha}{\to} s_2'$ and

$$(s_1', s_2') \in \bigcup\{\mathcal{R} \mid \mathcal{R} \text{ is a bisimulation}\} \ ,$$

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 α,

- if $s_1 \stackrel{\alpha}{\to} s_1'$, then there is a transition $s_2 \stackrel{\alpha}{\to} s_2'$ such that $s_1' \sim s_2'$;
- if $s_2 \stackrel{\alpha}{\to} s_2'$, then there is a transition $s_1 \stackrel{\alpha}{\to} s_1'$ 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 Proc having the following property:

($*$) for each action α,
- if $s_1 \stackrel{\alpha}{\rightarrow} s_1'$, then there is a transition $s_2 \stackrel{\alpha}{\rightarrow} s_2'$ such that $s_1' \sim s_2'$;
- if $s_2 \stackrel{\alpha}{\rightarrow} s_2'$, then there is a transition $s_1 \stackrel{\alpha}{\rightarrow} s_1'$ such that $s_1' \sim s_2'$.

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 $\mathcal{R}$? First of all, we must add the pair (s_1, s_2) to $\mathcal{R}$ because we wish to use that relation to prove $s_1 \sim s_2$. Since $\mathcal{R}$ should be a bisimulation, each transition $s_1 \stackrel{\alpha}{\rightarrow} s_1'$ from s_1 should be matched by a transition $s_2 \stackrel{\alpha}{\rightarrow} s_2'$ from s_2, for some state s_2' such that $(s_1', s_2') \in \mathcal{R}$. In light of the aforementioned property, this can be easily achieved by adding to the relation $\mathcal{R}$ 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 $\mathcal{R}$.

The above discussion suggests that we consider the relation

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

Indeed, by construction, the pair (s_1, s_2) is contained in $\mathcal{R}$. Moreover, using property ($*$) and statement 2 of the theorem, it is not hard to prove that $\mathcal{R}$ 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 $\mathcal{R}$ over the set of states of an LTS is a string bisimulation iff whenever $s_1 \,\mathcal{R}\, s_2$ and σ is a sequence of actions in* Act*:*

- *if $s_1 \stackrel{\sigma}{\rightarrow} s_1'$, then there is a transition $s_2 \stackrel{\sigma}{\rightarrow} s_2'$ such that $s_1' \,\mathcal{R}\, s_2'$;*
- *if $s_2 \stackrel{\sigma}{\rightarrow} s_2'$, then there is a transition $s_1 \stackrel{\sigma}{\rightarrow} s_1'$ such that $s_1' \,\mathcal{R}\, s_2'$.*

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 $K \stackrel{def}{=} P$. 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 $\alpha_1 \cdots \alpha_k$ ($k \geq 0$),*

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

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:*

$$\{(P \mid Q, Q \mid P) \mid \text{where } P, Q \text{ are CCS processes}\}$$
$$\{(P \mid \mathbf{0}, P) \mid \text{where } P \text{ is a CCS process}\}$$
$$\{((P \mid Q) \mid R, P \mid (Q \mid R)) \mid \text{where } P, Q, R \text{ are CCS processes}\} \ .$$

Conclude that, for all P, Q, R,

$$P \mid Q \sim Q \mid P \ , \tag{3.4}$$

$$P \mid \mathbf{0} \sim P \ , \text{ and} \tag{3.5}$$

$$(P \mid Q) \mid R \sim P \mid (Q \mid R) \ . \tag{3.6}$$

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

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

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

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

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

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

$$P_1 \mid P_2 \mid \cdots \mid P_k \ .$$

If $k = 0$, then, by convention, the above term is just $\mathbf{0}$.

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 α;
- $P + R \sim Q + R$ and $R + P \sim R + Q$, for each process R;
- $P \mid R \sim Q \mid R$ and $R \mid P \sim R \mid Q$, 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 $P \mid R \sim Q \mid R$. To this end, we shall build a bisimulation $\mathcal{R}$ that contains the pair of processes $(P \mid R, Q \mid R)$.

  Consider the relation

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

  You should readily be able to convince yourselves that the pair of processes $(P \mid R, Q \mid R)$ 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 $(P' \mid R', Q' \mid R')$ is contained in $\mathcal{R}$ and $P' \mid R' \stackrel{\alpha}{\rightarrow} S$ for some action α and CCS process S, then $Q' \mid R' \stackrel{\alpha}{\rightarrow} T$ for some CCS process T such that $(S, T) \in \mathcal{R}$. This we now proceed to do.

  Assume that $(P' \mid R', Q' \mid R')$ is contained in $\mathcal{R}$ and $P' \mid R' \stackrel{\alpha}{\rightarrow} S$ for some action α and CCS process S. We now proceed with the proof by a case analysis on the possible origins of the transition $P' \mid R' \stackrel{\alpha}{\rightarrow} S$. 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 $P' \mid R' \stackrel{\alpha}{\rightarrow} S$ may take, namely:

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

     $$P' \mid R' \stackrel{\alpha}{\rightarrow} S$$

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

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

     $$P' \mid R' \stackrel{\alpha}{\rightarrow} S$$

     because, by rule COM2 for parallel composition in Table 2.2, $R' \stackrel{\alpha}{\rightarrow} R''$ and $S = P' \mid R''$, 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,

$$P' \mid R' \stackrel{\alpha}{\rightarrow} S$$

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 $P' \stackrel{a}{\rightarrow} P''$, $R' \stackrel{\bar{a}}{\rightarrow} R''$ and $S = P'' \mid R''$.

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

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

$$Q' \mid R' \stackrel{\alpha}{\rightarrow} Q'' \mid R' \ .$$

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

$$Q' \mid R' \stackrel{\alpha}{\rightarrow} Q' \mid R'' \ .$$

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

$$Q' \mid R' \stackrel{\tau}{\rightarrow} Q'' \mid R'' \ .$$

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

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

  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 $(P' \setminus L, Q' \setminus L)$ is contained in $\mathcal{R}$ and $P' \setminus L \stackrel{\alpha}{\rightarrow} S$ for some action α and CCS process S, then $Q' \setminus L \stackrel{\alpha}{\rightarrow} T$ 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. □

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 τ each action α 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.*

$$\frac{P \stackrel{\alpha}{\rightarrow} P'}{\tau_L(P) \stackrel{\tau}{\rightarrow} \tau_L(P')} \quad \textit{if } \alpha \in L \textit{ or } \bar{\alpha} \in L$$

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

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 place holder when another process can be plugged) such that, for each process P,

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

Argue for your answer. ♦

Recall that we defined the specification of a counter thus:

$$\begin{aligned} \text{Counter}_0 &\stackrel{\text{def}}{=} \text{up.Counter}_1 \\ \text{Counter}_n &\stackrel{\text{def}}{=} \text{up.Counter}_{n+1} + \text{down.Counter}_{n-1} \quad (n > 0) \ . \end{aligned}$$

Moreover, we stated that we expect that process to be 'behaviourally equivalent' to the process C defined by

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

We can now show that, in fact, C and Counter_0 are strongly bisimilar. To this end, note that this follows if we can show that the relation $\mathcal{R}$ below

$$\begin{aligned} \{(C \mid \Pi_{i=1}^{k} P_i, \text{Counter}_n) \mid \quad & (1)\ k \geq 0 \ , \\ & (2)\ P_i = \mathbf{0} \text{ or } P_i = \text{down.}\mathbf{0}, \text{ for each } i \ , \\ & (3) \text{ the number of } i\text{s with } P_i = \text{down.}\mathbf{0} \text{ is } n\} \end{aligned}$$

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

Proposition 3.1 The relation $\mathcal{R}$ defined above is a strong bisimulation.

Proof: Assume that

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

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

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

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

We establish these two claims separately.

1. Assume that $C \mid \Pi_{i=1}^{k} P_i \stackrel{\alpha}{\rightarrow} P$ for some some action α and process P. Then

   - either $\alpha = \text{up}$ and $P = C \mid \text{down.}\mathbf{0} \mid \Pi_{i=1}^{k} P_i$
   - 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 ℓ, and at that position $P_\ell = \text{down.}\mathbf{0}$ and $P'_\ell = \mathbf{0}$.

In the former case, argue that the matching transition is

$$\text{Counter}_n \stackrel{\text{up}}{\rightarrow} \text{Counter}_{n+1} \ .$$

In the latter, argue that the matching transition is

$$\text{Counter}_n \stackrel{\text{down}}{\rightarrow} \text{Counter}_{n-1} \ .$$

2. Assume that $\text{Counter}_n \stackrel{\alpha}{\rightarrow} Q$ for some some action α 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 $C \mid \Pi_{i=1}^{k} P_i$ is left as an exercise for the reader.

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

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:

$$\begin{aligned} B_0^1 &\stackrel{\text{def}}{=} \text{in}.B_1^1 \\ B_1^1 &\stackrel{\text{def}}{=} \overline{\text{out}}.B_0^1 \ . \end{aligned}$$

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:

$$\begin{aligned} B_0^n &\stackrel{\text{def}}{=} \text{in}.B_1^n \\ B_i^n &\stackrel{\text{def}}{=} \text{in}.B_{i+1}^n + \overline{\text{out}}.B_{i-1}^n \quad \text{for } 0 < i < n \\ B_n^n &\stackrel{\text{def}}{=} \overline{\text{out}}.B_{n-1}^n \ . \end{aligned}$$

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

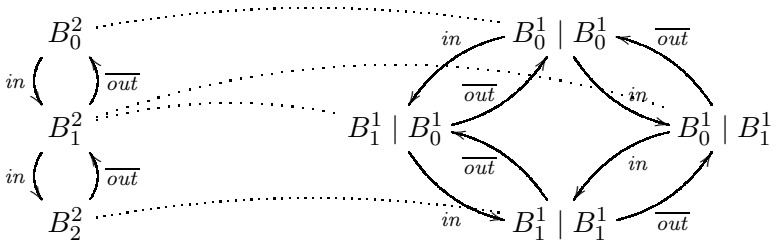


Figure 3.2: A bisimulation showing $B_0^2 \sim B_0^1 \mid B_0^1$

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 $i_1, i_2, \ldots, i_n \in \{0, 1\}$:

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

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

- $\big(B_0^n,\ B_0^1 \mid B_0^1 \mid \cdots \mid B_0^1\big) \in \mathcal{R}$, 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. □

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 α is an action:*

- *if $s_1 \stackrel{\alpha}{\rightarrow} s_1'$, then there is a transition $s_2 \stackrel{\alpha}{\rightarrow} s_2'$ such that $s_1' \mathcal{R} s_2'$.*

We say that s' simulates s, written $s \sqsubseteq_{\simeq} s'$, 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 $s \sqsubseteq_{\simeq} s'$ and $s' \sqsubseteq_{\simeq} s$ both hold.

1. *Prove that $\sqsubseteq_{\simeq}$ is a preorder and $\simeq$ is an equivalence relation.*
2. *Build simulations showing that*

$$\begin{array}{rcl} a.\mathbf{0} & \sqsubseteq_{\simeq} & a.a.\mathbf{0} \quad \textit{and} \\ a.b.\mathbf{0} + a.c.\mathbf{0} & \sqsubseteq_{\simeq} & a.(b.\mathbf{0} + c.\mathbf{0}) \ . \end{array}$$

   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 α is an action:*

- *if $s_1 \stackrel{\alpha}{\rightarrow} s_1'$, then there is a transition $s_2 \stackrel{\alpha}{\rightarrow} s_2'$ such that $s_1' \mathcal{R} s_2'$; and*
- *if $s_2 \stackrel{\alpha}{\rightarrow}$, then $s_1 \stackrel{\alpha}{\rightarrow}$.*

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

1. *Prove that $\sqsubseteq_{\mathrm{RS}\simeq}$ is a preorder and $\simeq_{\mathrm{RS}}$ is an equivalence relation.*
2. *Do the following relations hold?*

$$\begin{array}{rcl} a.\mathbf{0} & \sqsubseteq_{\mathrm{RS}\simeq} & a.a.\mathbf{0} \quad \textit{and} \\ a.b.\mathbf{0} + a.c.\mathbf{0} & \sqsubseteq_{\mathrm{RS}\simeq} & a.(b.\mathbf{0} + c.\mathbf{0}) \ . \end{array}$$

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?*

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

Exercise 3.19 (For the theoretically minded) *Consider the processes*

$$\begin{array}{rcl} P & \stackrel{def}{=} & a.b.c.\mathbf{0} + a.b.d.\mathbf{0} \quad \textit{and} \\ Q & \stackrel{def}{=} & a.(b.c.\mathbf{0} + b.d.\mathbf{0}) \ . \end{array}$$

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*

$$(P \mid R) \setminus L \textit{ and } (Q \mid R) \setminus L$$

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

$$\begin{array}{rcl} P \mid Q & \sim & Q \mid P \ , \\ P \mid \mathbf{0} & \sim & P \ , \text{ and} \\ (P \mid Q) \mid R & \sim & P \mid (Q \mid R) \ . \end{array}$$

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).)

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 τ actions in process behaviours are supposed to be *internal*, and thus *unobservable*. This is a natural consequence of Milner's design decision to let τ 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 τ 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 $a.\tau.\mathbf{0}$ and $a.\mathbf{0}$. Since τ actions should be unobservable, we intuitively expect these to be observationally equivalent. Unfortunately, however, the processes $a.\tau.\mathbf{0}$ and $a.\mathbf{0}$ 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 τ, and $a.\tau.\mathbf{0}$ affords the trace $a\tau$, whereas $a.\mathbf{0}$ does not.

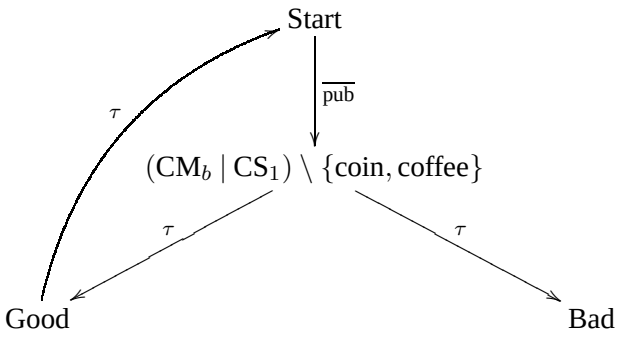
In hindsight, this failure of strong bisimilarity to account for the unobservable nature of τ actions is expected because the definition of strong bisimulation treats internal actions as if they were ordinary observable actions. What we should like to have is a notion of bisimulation equivalence that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

$$\mathrm{CM}_b \stackrel{\text{def}}{=} \mathrm{coin}.\overline{\mathrm{coffee}}.\mathrm{CM}_b + \mathrm{coin}.\mathrm{CM}_b \ . \tag{3.7}$$

(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



where

Start	$\equiv$	$(\text{CM}_b \mid \text{CS}) \setminus \{\text{coin}, \text{coffee}\}$		CS	$\stackrel{\text{def}}{=}$	$\overline{\text{pub}}.\text{CS}_1$
Good	$\equiv$	$(\overline{\text{coffee}}.\text{CM}_b \mid \text{CS}_2) \setminus \{\text{coin}, \text{coffee}\}$		CS_1	$\stackrel{\text{def}}{=}$	$\overline{\text{coin}}.\text{CS}_2$
Bad	$\equiv$	$(\text{CM}_b \mid \text{CS}_2) \setminus \{\text{coin}, \text{coffee}\}$		CS_2	$\stackrel{\text{def}}{=}$	$\text{coffee}.\text{CS}$.

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

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

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

The short-hands Bad and Good are also introduced in the picture using the 'declarations'

$$\begin{array}{lcl} \text{Good} & \equiv & (\overline{\text{coffee}}.\text{CM}_b \mid \text{CS}_2) \setminus \{\text{coin}, \text{coffee}\} \quad \text{and} \\ \text{Bad} & \equiv & (\text{CM}_b \mid \text{CS}_2) \setminus \{\text{coin}, \text{coffee}\} \ . \end{array}$$

Note that, there are two possible τ-transitions that stem from the process

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

and that one of them, namely

$$(\text{CM}_b \mid \text{CS}_1) \setminus \{\text{coin}, \text{coffee}\} \stackrel{\tau}{\rightarrow} (\text{CM}_b \mid \text{CS}_2) \setminus \{\text{coin}, \text{coffee}\} \ ,$$

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 ε-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 SmUni is *not* equivalent to that of the process $(\mathrm{CM}_b \mid \mathrm{CS}) \setminus \{\mathrm{coin}, \mathrm{coffee}\}$ 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 SmUni from $(\mathrm{CM}_b \mid \mathrm{CS}) \setminus \{\mathrm{coin}, \mathrm{coffee}\}$, 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 α, we shall write $P \stackrel{\alpha}{\Rightarrow} Q$ iff either

- $\alpha \neq \tau$ and there are processes P' and Q' such that
$$P(\stackrel{\tau}{\rightarrow})^* P' \stackrel{\alpha}{\rightarrow} Q'(\stackrel{\tau}{\rightarrow})^* Q$$
- or $\alpha = \tau$ and $P(\stackrel{\tau}{\rightarrow})^* Q$,

where we write $(\stackrel{\tau}{\rightarrow})^*$ for the reflexive and transitive closure of the relation $\stackrel{\tau}{\rightarrow}$. ♦

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

In the LTS depicted in Table 3.1, apart from the obvious one step $\overline{\mathrm{pub}}$-labelled transition, we have that

$$\mathrm{Start} \stackrel{\overline{\mathrm{pub}}}{\Rightarrow} \mathrm{Good}\ ,$$
$$\mathrm{Start} \stackrel{\overline{\mathrm{pub}}}{\Rightarrow} \mathrm{Bad}\ ,\ \text{and}$$
$$\mathrm{Start} \stackrel{\overline{\mathrm{pub}}}{\Rightarrow} \mathrm{Start}\ .$$

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 $\mathcal{R}$ over the set of states of an LTS is a *weak bisimulation* iff whenever $s_1 \,\mathcal{R}\, s_2$ and α is an action (including τ):

- if $s_1 \stackrel{\alpha}{\rightarrow} s_1'$, then there is a transition $s_2 \stackrel{\alpha}{\Rightarrow} s_2'$ such that $s_1' \,\mathcal{R}\, s_2'$;
- if $s_2 \stackrel{\alpha}{\rightarrow} s_2'$, then there is a transition $s_1 \stackrel{\alpha}{\Rightarrow} s_1'$ such that $s_1' \,\mathcal{R}\, s_2'$.

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.

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

Obviously $s \not\sim t$. 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 $s \stackrel{\tau}{\rightarrow} s_1$ then $t \stackrel{\tau}{\Rightarrow} t$ and $(s_1,t) \in \mathcal{R}$. If $t \stackrel{a}{\rightarrow} t_1$ then $s \stackrel{a}{\Rightarrow} s_2$ and $(s_2,t_1) \in \mathcal{R}$.
- Let us examine all possible transitions from (s_1,t). If $s_1 \stackrel{a}{\rightarrow} s_2$ then $t \stackrel{a}{\Rightarrow} t_1$ and $(s_2,t_1) \in \mathcal{R}$. Similarly if $t \stackrel{a}{\rightarrow} t_1$ then $s_1 \stackrel{a}{\Rightarrow} s_2$ 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 $a.\mathbf{0} \approx a.\tau.\mathbf{0}$ 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

$$\text{Spec} \stackrel{\text{def}}{=} \overline{\text{pub}}.\text{Spec} \ ,$$

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

$$\begin{aligned} \text{A?} &\stackrel{\text{def}}{=} a.\mathbf{0} + \tau.\text{B?} \\ \text{B?} &\stackrel{\text{def}}{=} b.\mathbf{0} + \tau.\text{A?} \ . \end{aligned}$$

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

$$\text{A?} \approx \text{B?} \approx a.\mathbf{0} + b.\mathbf{0} \ .$$

(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 τ-loop

$$\text{A?} \stackrel{\tau}{\rightarrow} \text{B?} \stackrel{\tau}{\rightarrow} \text{A?} \ ,$$

but $a.\mathbf{0} + b.\mathbf{0}$ 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,

$$
\begin{array}{lcl@{\qquad}lcl}
\text{Send} & \stackrel{\text{def}}{=} & \text{acc.Sending} & \text{Rec} & \stackrel{\text{def}}{=} & \text{trans.Del} \\
\text{Sending} & \stackrel{\text{def}}{=} & \overline{\text{send}}\text{.Wait} & \text{Del} & \stackrel{\text{def}}{=} & \overline{\text{del}}\text{.Ack} \\
\text{Wait} & \stackrel{\text{def}}{=} & \text{ack.Send} + \text{error.Sending} & \text{Ack} & \stackrel{\text{def}}{=} & \overline{\text{ack}}\text{.Rec}
\end{array}
$$

$$
\begin{array}{rcl}
\text{Med} & \stackrel{\text{def}}{=} & \text{send.Med}' \\
\text{Med}' & \stackrel{\text{def}}{=} & \tau\text{.Err} + \overline{\text{trans}}\text{.Med} \\
\text{Err} & \stackrel{\text{def}}{=} & \overline{\overline{\text{error}}}\text{.Med}
\end{array}
$$

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 $\mathbf{0}$ is observationally equivalent to the process

$$\text{Div} \stackrel{\text{def}}{=} \tau\text{.Div} \ .$$

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 $\mathbf{0}$ 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

$$(\text{Send} \mid \text{Med} \mid \text{Rec}) \setminus L \qquad (L = \{\text{send}, \text{error}, \text{trans}, \text{ack}\}) \tag{3.8}$$

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:

$$\text{ProtocolSpec} \stackrel{\text{def}}{=} \text{acc}.\overline{\text{del}}.\text{ProtocolSpec} \ .$$

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:

   $s_1 \approx s_2$ iff for each action α,

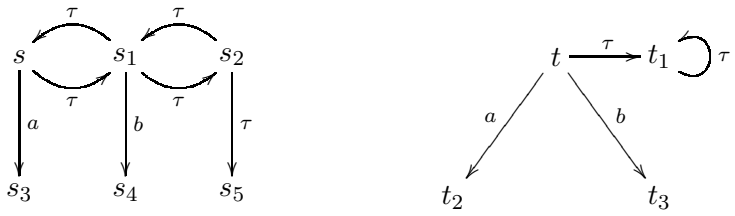
   - if $s_1 \stackrel{\alpha}{\rightarrow} s_1'$, then there is a transition $s_2 \stackrel{\alpha}{\Rightarrow} s_2'$ such that $s_1' \approx s_2'$;
   - if $s_2 \stackrel{\alpha}{\rightarrow} s_2'$, then there is a transition $s_1 \stackrel{\alpha}{\Rightarrow} s_1'$ such that $s_1' \approx s_2'$.

Proof: The proof follows the lines of that of Theorem 3.1, and is therefore omitted. □

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.*



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 τ-laws, hold:*

$$\alpha.\tau.P \approx \alpha.P \tag{3.9}$$

$$P + \tau.P \approx \tau.P \tag{3.10}$$

$$\alpha.(P + \tau.Q) \approx \alpha.(P + \tau.Q) + \alpha.Q \ . \tag{3.11}$$

Hint: Build appropriate weak bisimulations. ♦

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

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

Prove that no τ-free CCS process is observationally equivalent to $a.\mathbf{0} + \tau.\mathbf{0}$. ♦

Exercise 3.29 *Prove that, for each CCS process P, the process $P \setminus (\mathsf{Act} - \{\tau\})$ is observationally equivalent to $\mathbf{0}$. 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 α (including τ), if $s_1 \stackrel{\alpha}{\Rightarrow} s_1'$, then there is a transition $s_2 \stackrel{\alpha}{\Rightarrow} s_2'$ 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 $\stackrel{\alpha}{\Rightarrow}$, with α ranging over the set of actions including τ. ♦

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

- *$s \stackrel{\varepsilon}{\Rightarrow} t$ iff $s \stackrel{\tau}{\Rightarrow} t$, and*
- *$s \stackrel{a\sigma'}{\Rightarrow} t$ iff $s \stackrel{a}{\Rightarrow} s' \stackrel{\sigma'}{\Rightarrow} t$, 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 σ is a (possibly empty) sequence of observable actions in $\mathcal{L}$:

- *if $s_1 \stackrel{\sigma}{\Rightarrow} s_1'$, then there is a transition $s_2 \stackrel{\sigma}{\Rightarrow} s_2'$ such that $s_1' \mathcal{R} s_2'$;*
- *if $s_2 \stackrel{\sigma}{\Rightarrow} s_2'$, then there is a transition $s_1 \stackrel{\sigma}{\Rightarrow} s_1'$ 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 $\mathbf{0}$ is observationally equivalent to $\tau.\mathbf{0}$. However, it is not hard to see that

$$a.\mathbf{0} + \mathbf{0} \not\approx a.\mathbf{0} + \tau.\mathbf{0} \ .$$

In fact, the transition $a.\mathbf{0}+\tau.\mathbf{0} \stackrel{\tau}{\rightarrow} \mathbf{0}$ from the process $a.\mathbf{0}+\tau.\mathbf{0}$ can only be matched by $a.\mathbf{0} + \mathbf{0} \stackrel{\tau}{\Rightarrow} a.\mathbf{0} + \mathbf{0}$, and the processes $\mathbf{0}$ and $a.\mathbf{0} + \mathbf{0}$ 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 α,
- $P \mid R \approx Q \mid R$ 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 $Q \stackrel{a}{\Rightarrow} Q'$ and $R \stackrel{\bar{a}}{\rightarrow} R'$, then $Q|R \stackrel{\tau}{\Rightarrow} Q'|R'$.

Show this fact by induction on the number of τ-steps in the transition $Q \stackrel{a}{\Rightarrow} Q'$. ♦

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

$$\mathit{ThinkFactory} \stackrel{def}{=} \mathit{think}.\mathit{ThinkFactory} \ .$$

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 (Glabbeek and Weijland, 1996) iff it is symmetric, and whenever $s \,\mathcal{R}\, t$ and α is an action (including τ):*

if $s \stackrel{\alpha}{\rightarrow} s'$, then

- *either $\alpha = \tau$ and $s' \,\mathcal{R}\, t$*
- *or there is a $k \geq 0$ and a sequence of transitions*

$$t = t_0 \stackrel{\tau}{\rightarrow} t_1 \stackrel{\tau}{\rightarrow} t_2 \cdots t_k \stackrel{\alpha}{\rightarrow} t'$$

such that $s \,\mathcal{R}\, t_i$ for each $i \in \{0, \ldots, k\}$, and $s' \,\mathcal{R}\, t'$.

Two states s and t are branching bisimulation equivalent (or branching bisimilar) iff there is a branching bisimulation that relates them. The largest branching bisimulation is called branching bisimilarity.

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 τ-laws from Exercise 3.26 holds with respect to branching bisimilarity?*
4. *Is branching bisimilarity a congruence over the language CCS?*

In answering the last question, you may assume that branching bisimilarity is an equivalence relation. In fact, showing that branching bisimilarity is transitive is non-trivial; see, for instance, (Basten, 1996) for a proof. ♦

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

$s_1 \approx^c s_2$ iff for each action α (including τ):

- *if* $s_1 \stackrel{\alpha}{\rightarrow} s_1'$, *then there is a sequence of transitions* $s_2 \stackrel{\tau}{\Rightarrow} s_2'' \stackrel{\alpha}{\rightarrow} s_2''' \stackrel{\tau}{\Rightarrow} s_2'$ *such that* $s_1' \approx s_2'$;
- *if* $s_2 \stackrel{\alpha}{\rightarrow} s_2'$, *then there is a sequence of transitions* $s_1 \stackrel{\tau}{\Rightarrow} s_1'' \stackrel{\alpha}{\rightarrow} s_1''' \stackrel{\tau}{\Rightarrow} s_1'$ *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* α;
   - $P + R \approx^c Q + R$ *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 τ*-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.

Definition 3.5 [Strong bisimulation game] Let $(\mathsf{Proc}, \mathsf{Act}, \{\stackrel{\alpha}{\rightarrow} | \ \alpha \in \mathsf{Act}\})$ be a labelled transition system. A *strong bisimulation game* starting from the pair of states $(s_1, t_1) \in \mathsf{Proc} \times \mathsf{Proc}$ 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 $\mathsf{Proc} \times \mathsf{Proc}$. 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 α from Act.
   - If the attacker chose left then he has to perform a transition $s \stackrel{\alpha}{\rightarrow} s'$ for some state $s' \in \mathsf{Proc}$.
   - If the attacker chose right then he has to perform a transition $t \stackrel{\alpha}{\rightarrow} t'$ for some state $t' \in \mathsf{Proc}$.
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 $t \stackrel{\alpha}{\rightarrow} t'$ for some $t' \in \mathsf{Proc}$.
   - If the attacker chose right then the defender plays on the left-hand side and has to respond by making a transitions $s \stackrel{\alpha}{\rightarrow} s'$ for some $s' \in \mathsf{Proc}$.
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.

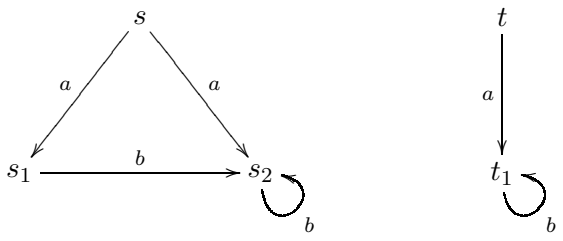
The following proposition relates strong bisimilarity with the corresponding game characterization (see, e.g., (Stirling, 1995; Thomas, 1993)).

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).

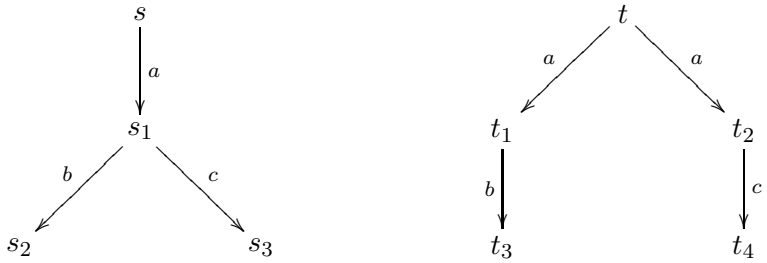
From (s_2, t_1) it is easy to see that any continuation of the game will always go through the same current configuration (s_2, t_1) and hence the game will be necessarily infinite. According to the definition of a winning play, the defender is the winner in this case.

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).



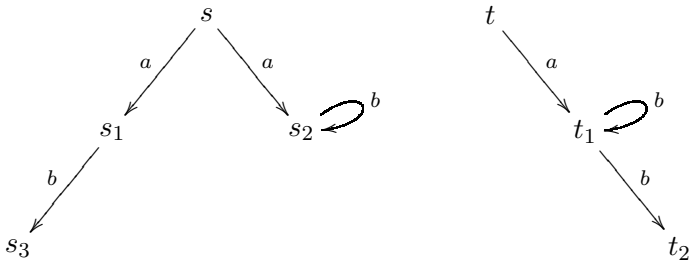
We will show that $s \not\sim t$ by describing a universal winning strategy for the attacker in the bisimulation game starting from (s, t). We will in fact show two different strategies (but of course finding one is sufficient for proving non-bisimilarity).

- 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 $t_1 \xrightarrow{c}\!\!\!\!\!\!\rightarrow$.

♦

Example 3.7 Let us consider a slightly more complex transition system.

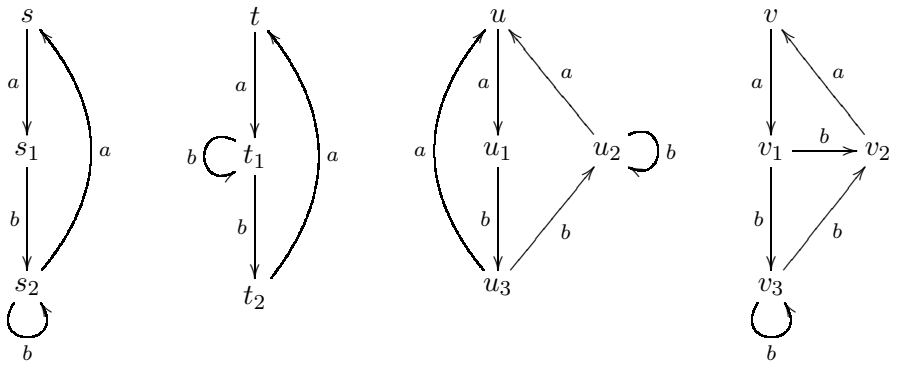


We will define attacker's universal winning strategy from (s, t) and hence show that $s \not\sim t$.

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

♦

Exercise 3.37 *Consider the following labelled transition system.*



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 $\mathcal{R}$ over the set of states of an LTS is a simulation iff whenever $s_1 \mathcal{R} s_2$ and a is an action then*

- *if $s_1 \stackrel{a}{\rightarrow} s_1'$, then there is a transition $s_2 \stackrel{a}{\rightarrow} s_2'$ such that $s_1' \mathcal{R} s_2'$.*

A binary relation $\mathcal{R}$ over the set of states of an LTS is a 2-nested simulation iff $\mathcal{R}$ is a simulation and moreover $\mathcal{R}^{-1} \subseteq \mathcal{R}$.

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 τ, 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 $\stackrel{\alpha}{\Rightarrow}$ instead of only $\stackrel{\alpha}{\rightarrow}$ as in the strong bisimulation game. The attacker is still allowed to use only the $\stackrel{\alpha}{\rightarrow}$ moves. ⧫

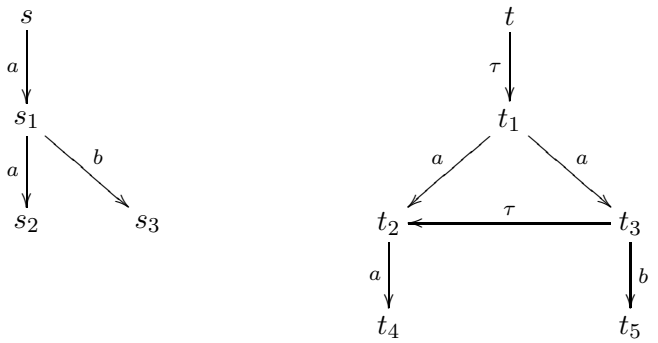
The definitions of a play and winning strategy are exactly as before and we have a similar proposition as for the strong bisimulation game.

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 τ (let us say $s \stackrel{\tau}{\rightarrow} s'$) then the defender can (as one possibility) simply answer by doing 'nothing', i.e., by idling in the state t (as we always have $t \stackrel{\tau}{\Rightarrow} t$). 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 $s \stackrel{a}{\rightarrow} s_1$. The defender has three possible moves to answer: (i) $t \stackrel{a}{\Rightarrow} t_2$ via t_1, (ii) $t \stackrel{a}{\Rightarrow} t_2$ via t_1 and t_3, and (iii) $t \stackrel{a}{\Rightarrow} t_3$ 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 $s_1 \stackrel{b}{\rightarrow} s_3$, and the defender loses because $t_2 \stackrel{b}{\not\Rightarrow}$.

From the configuration (s_1,t_3) the attacker plays the τ-move from the right-hand side: $t_3 \stackrel{\tau}{\rightarrow} t_2$. Defender's only answer from s_1 is $s_1 \stackrel{\tau}{\Rightarrow} s_1$ because no τ 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 $\stackrel{a}{\rightarrow}$ moves for the attacks, and the defender can use $\stackrel{a}{\Rightarrow}$ moves in response. Argue that if we modify the rules of the game so that the attacker can also use moves of the form $\stackrel{a}{\Rightarrow}$ then this does not provide any additional power for the attacker. Conclude that both versions of the game provide the same answer about bisimilarity/non-bisimilarity 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

$$a_1.\mathbf{0} \mid a_2.\mathbf{0} \mid \cdots \mid a_n.\mathbf{0}$$

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* (Petri, 1962; 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* (often abbreviated to

poset) is a pair $(D, \sqsubseteq)$, 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. ♦

Example 4.1 The following are examples of posets.

- $(\mathbb{N}, \leq)$, where $\mathbb{N}$ denotes the set of natural numbers, and $\leq$ stands for the standard ordering over $\mathbb{N}$.
- $(\mathbb{R}, \leq)$, where $\mathbb{R}$ denotes the set of real numbers, and $\leq$ stands for the standard ordering over $\mathbb{R}$.
- $(A^*, \leq)$, 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!)
- 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:
  1. the length of s is smaller than that of t;
  2. 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 $a \leq b$.
- 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:

$$f \sqsubseteq g \text{ iff } f(s) \sqsubseteq g(s), \text{ for each } s \in S \ .$$

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

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?* ♦

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 $(2^{\mathsf{Proc}}, \subseteq)$ is a poset for each set of states 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 $\bigsqcup X$, 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 $\bigsqcap X$, 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 $(\{a, b\}^*, \leq)$, where $\leq$ is the prefix ordering over strings defined in the third bullet of Example 4.1. Find examples of subsets of $\{a, b\}^*$ 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 $\bigsqcup X$ and $\bigsqcap X$ exist for every subset X of D. ♦

Note that a complete lattice $(D, \sqsubseteq)$ has a least element $\bot = \bigsqcap D$, often called *bottom*, and a top element $\top = \bigsqcup D$. 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 $\bigsqcup \emptyset$ and $\bigsqcap \emptyset$? 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 least upper bounds for its infinite subsets.
- The poset $(\mathbb{N} \cup \{\infty\}, \sqsubseteq)$, obtained by adding a largest element ∞ to $(\mathbb{N}, \leq)$, is instead a complete lattice. This complete lattice can be pictured as follows:

$$\begin{array}{c} \infty \\ \vdots \\ \uparrow \\ 2 \\ \uparrow \\ 1 \\ \uparrow \\ 0 \end{array}$$

  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

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

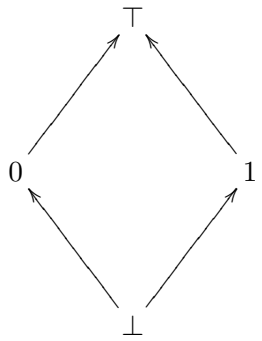
is monotonic. The set $\{1, 2\}$ is a fixed point of f because

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

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 over $\{\bot, 0, 1, \top\}$ 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 $f : D \to D$ be monotonic. Then f has a largest fixed point $z_{\max}$ and a least fixed point $z_{\min}$ given by

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

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

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

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 $z_{\max}$ is a fixed point of f, it is sufficient to show that

$$z_{\max} \sqsubseteq f(z_{\max}) \quad \text{and} \tag{4.1}$$
$$f(z_{\max}) \sqsubseteq z_{\max} \ . \tag{4.2}$$

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

$$z_{\max} = \bigsqcup A \ .$$

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

$$x \sqsubseteq f(x) \sqsubseteq f(z_{\max}) \ .$$

Thus $f(z_{\max})$ is an upper bound for the set A. By definition, $z_{\max}$ is the *least upper bound* of A. Thus $z_{\max} \sqsubseteq f(z_{\max})$, 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 $f(z_{\max}) \sqsubseteq f(f(z_{\max}))$. This implies that $f(z_{\max}) \in A$. Therefore $f(z_{\max}) \sqsubseteq z_{\max}$, as $z_{\max}$ is an upper bound for A.

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

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

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

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

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

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

$$f(z_{\min}) \sqsubseteq z_{\min} \quad \text{and} \tag{4.3}$$

$$z_{\min} \sqsubseteq f(z_{\min}) \ . \tag{4.4}$$

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 $z_{\min}$ 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 $f : S \to S$. If we instantiate the statement of the above theorem to this setting, the largest and least fixed points for f can be characterized thus:

$$z_{\max} = \bigcup \{X \subseteq S \mid X \subseteq f(X)\} \quad \text{and}$$

$$z_{\min} = \bigcap \{X \subseteq S \mid f(X) \subseteq X\} \ .$$

For instance, the largest fixed point of the function $f : 2^{\mathbb{N}} \rightarrow 2^{\mathbb{N}}$ 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 $f : D \rightarrow D$. For each natural number n, we define $f^n(d)$ as follows:

$$\begin{array}{rcl} f^0(d) & = & d \text{ and} \\ f^{n+1}(d) & = & f(f^n(d)) \ . \end{array}$$

♦

Theorem 4.2 Let $(D, \sqsubseteq)$ be a *finite* complete lattice and let $f : D \rightarrow D$ 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

$$\bot \sqsubseteq f(\bot) \sqsubseteq f^2(\bot) \sqsubseteq \ldots \sqsubseteq f^i(\bot) \sqsubseteq f^{i+1}(\bot) \sqsubseteq \ldots$$

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 $f(\bot) \sqsubseteq f(d) = d$. 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 is 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* ω-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 ω-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*

   $$f(\bigsqcup_{i \geq 0} d_i) = \bigsqcup_{i \geq 0} f(d_i) \ ,$$

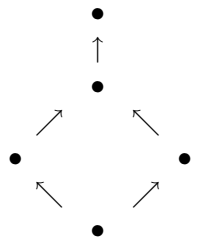
   for each ω-chain d_i ($i \geq 0$).

   Prove that if $(D, \sqsubseteq)$ is a cpo and $f : D \to D$ is continuous, then the poset

   $$(\{x \in D \mid f(x) = x\}, \sqsubseteq) \ ,$$

   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*

   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 $\bigsqcup X \in \{x \in D \mid x \sqsubseteq f(x)\}$.

(b) Give an example showing that, in general,

$$\bigsqcap X \notin \{x \in D \mid x \sqsubseteq f(x)\} \ .$$

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 $\{x \in D \mid f(x) \sqsubseteq x\}$.*

   (a) Prove that $\bigsqcap X \in \{x \in D \mid f(x) \sqsubseteq x\}$.

   (b) Give an example showing that, in general, $\bigsqcup X \notin \{x \in D \mid f(x) \sqsubseteq x\}$. Hint: Use your solution to Exercise 3b 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

   $$f \preceq g \text{ iff } f(d) \sqsubseteq g(d), \text{ for each } d \in D \ .$$

   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 $d \in D$,*

   $$(\bigvee \mathcal{F})(d) = \bigsqcup \{f(d) | f \in \mathcal{F}\} \ .$$

   - *If $\mathcal{F} \subseteq D \to_{mon} D$ then, for each $d \in D$,*

   $$(\bigwedge \mathcal{F})(d) = \bigsqcap \{f(d) | f \in \mathcal{F}\} \ .$$

   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 $(\mathsf{Proc}, \mathsf{Act}, \{\stackrel{\alpha}{\rightarrow} \mid \alpha \in \mathsf{Act}\})$ be a labelled transition system. We recall that a relation $\mathcal{R} \subseteq \mathsf{Proc} \times \mathsf{Proc}$ is a *strong bisimulation*—see Definition 3.2 on page 43—if the following holds:

> If $(p, q) \in \mathcal{R}$ then, for every $\alpha \in \mathsf{Act}$:
>
> 1. $p \stackrel{\alpha}{\rightarrow} p'$ implies $q \stackrel{\alpha}{\rightarrow} q'$ for some q' such that $(p', q') \in \mathcal{R}$;
> 2. $q \stackrel{\alpha}{\rightarrow} q'$ implies $p \stackrel{\alpha}{\rightarrow} p'$ for some p' such that $(p', q') \in \mathcal{R}$.

Then *strong bisimulation equivalence* (or *strong bisimilarity*) is defined as

$$\sim \;=\; \bigcup \{\mathcal{R} \in 2^{(\mathsf{Proc} \times \mathsf{Proc})} \mid \mathcal{R} \text{ is a strong bisimulation}\} \;\; .$$

In what follows we shall describe the relation $\sim$ as a fixed point to a suitable monotonic function. First we note that $(2^{(\mathsf{Proc} \times \mathsf{Proc})}, \subseteq)$ (i.e., the set of binary relations over 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 $\mathcal{R}$ over Proc—that is, an element of the set $2^{(\mathsf{Proc} \times \mathsf{Proc})}$. We define the set $\mathcal{F}(\mathcal{R})$ as follows:

> $(p, q) \in \mathcal{F}(\mathcal{R})$, for all $p, q \in \mathsf{Proc}$, if and only if
>
> 1. $p \stackrel{\alpha}{\rightarrow} p'$ implies $q \stackrel{\alpha}{\rightarrow} q'$ for some q' such that $(p', q') \in \mathcal{R}$;
> 2. $q \stackrel{\alpha}{\rightarrow} q'$ implies $p \stackrel{\alpha}{\rightarrow} p'$ for some p' such that $(p', q') \in \mathcal{R}$.

In other words, $\mathcal{F}(\mathcal{R})$ 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 $\mathcal{R}$.

You should now convince yourselves that a relation $\mathcal{R}$ is a bisimulation if and only if $\mathcal{R} \subseteq \mathcal{F}(\mathcal{R})$, and consequently that

$$\sim \;=\; \bigcup\{\mathcal{R} \in 2^{(\mathsf{Proc}\times\mathsf{Proc})} \mid \mathcal{R} \subseteq \mathcal{F}(\mathcal{R})\} \;\; .$$

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 $\mathcal{F}$ are precisely the binary relations $\mathcal{R}$ over Proc such that $\mathcal{R} \subseteq \mathcal{F}(\mathcal{R})$. This means that the definition of $\sim$ matches the one for the largest fixed point for $\mathcal{F}$ perfectly!

We note that if $\mathcal{R}, \mathcal{S} \in 2^{(\mathsf{Proc}\times\mathsf{Proc})}$ and $\mathcal{R} \subseteq \mathcal{S}$ then $\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}(\mathcal{S})$—that is, the function $\mathcal{F}$ 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 $\mathcal{F}$. In particular, by Theorem 4.2, if Proc is finite then $\sim$ is equal to $\mathcal{F}^M(\mathsf{Proc} \times \mathsf{Proc})$ 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
>
> $$\mathcal{F}^0(\mathsf{Proc} \times \mathsf{Proc}) \supseteq \mathcal{F}^1(\mathsf{Proc} \times \mathsf{Proc}) \supseteq \mathcal{F}^2(\mathsf{Proc} \times \mathsf{Proc}) \supseteq \cdots$$
>
> until the sequence stabilizes. (Recall, that $\mathcal{F}^0(\mathsf{Proc} \times \mathsf{Proc})$ 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:

$$\begin{aligned} Q_1 &= b.Q_2 + a.Q_3 \\ Q_2 &= c.Q_4 \\ Q_3 &= c.Q_4 \\ Q_4 &= b.Q_2 + a.Q_3 + a.Q_1 \;\; . \end{aligned}$$

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 $\mathcal{F}^i(\mathsf{Proc} \times \mathsf{Proc})$ for $i \geq 1$ thus:

$$\begin{aligned}
\mathcal{F}^1(\mathsf{Proc} \times \mathsf{Proc}) &= \{(Q_1, Q_4), (Q_4, Q_1), (Q_2, Q_3), (Q_3, Q_2)\} \cup I \\
\mathcal{F}^2(\mathsf{Proc} \times \mathsf{Proc}) &= \{(Q_2, Q_3), (Q_3, Q_2)\} \cup I \quad \text{and finally} \\
\mathcal{F}^3(\mathsf{Proc} \times \mathsf{Proc}) &= \mathcal{F}^2(\mathsf{Proc} \times \mathsf{Proc}) \ .
\end{aligned}$$

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:*

$$\begin{aligned}
P_1 &= a.P_2 \\
P_2 &= a.P_1 \\
P_3 &= a.P_2 + a.P_4 \\
P_4 &= a.P_3 + a.P_5 \\
P_5 &= \mathbf{0} \ .
\end{aligned}$$

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 $(\mathsf{Proc}, \mathsf{Act}, \{ \stackrel{\alpha}{\rightarrow} \mid \alpha \in \mathsf{Act}\})$ be a labelled transition system. For each $i \geq 0$, define the relation $\sim_i$ over* Proc *as follows:*

- *$s_1 \sim_0 s_2$ holds always;*
- *$s_1 \sim_{i+1} s_2$ holds iff for each action α:*
  - *if $s_1 \stackrel{\alpha}{\rightarrow} s_1'$, then there is a transition $s_2 \stackrel{\alpha}{\rightarrow} s_2'$ such that $s_1' \sim_i s_2'$;*
  - *if $s_2 \stackrel{\alpha}{\rightarrow} s_2'$, then there is a transition $s_1 \stackrel{\alpha}{\rightarrow} s_1'$ such that $s_1' \sim_i s_2'$.*

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. $\sim_i = \mathcal{F}^i(\mathsf{Proc} \times \mathsf{Proc})$.

⧫

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 Act is given by the following abstract syntax:

$$F, G ::= tt \mid ff \mid F \wedge G \mid F \vee G \mid \langle a \rangle F \mid [a]F \ ,$$

where $a \in \mathsf{Act}$, and we use tt and ff to denote 'true' and 'false', respectively. If $A = \{a_1, \ldots, a_n\} \subseteq \mathsf{Act}$ ($n \geq 0$), we use the abbreviation $\langle A \rangle F$ for the formula $\langle a_1 \rangle F \vee \ldots \vee \langle a_n \rangle F$ and $[A]F$ for the formula $[a_1]F \wedge \ldots \wedge [a_n]F$. (If $A = \emptyset$, then $\langle A \rangle F = ff$ and $[A]F = tt$.) ♦

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 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 tt.
- No process satisfies ff.
- A process satisfies $F \wedge G$ (respectively, $F \vee G$) iff it satisfies both F and G (respectively, either F or G).
- A process satisfies $\langle a \rangle F$ for some $a \in \mathsf{Act}$ iff it affords an a-labelled transition leading to a state satisfying F.
- A process satisfies $[a]F$ for some $a \in \mathsf{Act}$ 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

$$(\mathsf{Proc}, \mathsf{Act}, \{\xrightarrow{a} \mid a \in \mathsf{Act}\}) \ .$$

We shall use $[\![F]\!]$ to denote the set of processes in Proc that satisfy F. This we now proceed to define formally.

Definition 5.2 [Denotational semantics] We define $[\![F]\!] \subseteq \mathsf{Proc}$ for $F \in \mathcal{M}$ by

1. $[\![tt]\!] = \mathsf{Proc}$
2. $[\![ff]\!] = \emptyset$
3. $[\![F \wedge G]\!] = [\![F]\!] \cap [\![G]\!]$
4. $[\![F \vee G]\!] = [\![F]\!] \cup [\![G]\!]$
5. $[\![\langle a \rangle F]\!] = \langle \cdot a \cdot \rangle [\![F]\!]$
6. $[\![[a]F]\!] = [\cdot a \cdot][\![F]\!]$,

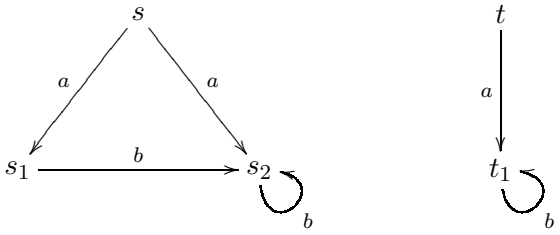
where we use the set operators $\langle \cdot a \cdot \rangle, [\cdot a \cdot] : 2^{\mathsf{Proc}} \to 2^{\mathsf{Proc}}$ defined by

$$\begin{aligned} \langle \cdot a \cdot \rangle S &= \{p \in \mathsf{Proc} \mid p \xrightarrow{a} p' \text{ and } p' \in S, \text{ for some } p'\} \quad \text{and} \\ [\cdot a \cdot] S &= \{p \in \mathsf{Proc} \mid p \xrightarrow{a} p' \text{ implies } p' \in S, \text{ for each } p'\} \ . \end{aligned}$$

We write $p \models F$, read 'p satisfies 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.



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 $s \stackrel{a}{\rightarrow} s_2$, 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

$$[\cdot a \cdot] \{s_1, t_1\} = \{p \in \mathsf{Proc} \mid p \stackrel{a}{\rightarrow} p' \text{ implies } p' \in \{s_1, t_1\}, \text{ for each } p'\} \ .$$

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 $\stackrel{a}{\rightarrow}$. This means, in particular, that we are not considering τ 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 $P \stackrel{a}{\rightarrow} Q$ 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 $F = tt$. So, it looks as if we can express our natural language requirement in terms of the formula $\langle \text{coffee} \rangle tt$. 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:

$$\begin{aligned}
[\![\langle \text{coffee} \rangle tt]\!] &= \langle \cdot \text{coffee} \cdot \rangle [\![tt]\!] \\
&= \langle \cdot \text{coffee} \cdot \rangle \mathsf{Proc} \\
&= \{P \mid P \stackrel{\text{coffee}}{\rightarrow} P' \text{ for some } P' \in \mathsf{Proc}\} \\
&= \{P \mid P \stackrel{\text{coffee}}{\rightarrow}\} \ .
\end{aligned}$$

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 $[\text{tea}]ff$. Intuitively this formula says that all the states that a process can reach by doing a tea-labelled transition must satisfy the formula ff, i.e., false. Since no state has the property 'false', the only way that a process can satisfy the property $[\text{tea}]ff$ 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:

$$\begin{aligned}
[\![[\text{tea}]ff]\!] &= [\cdot \text{tea} \cdot][\![ff]\!] \\
&= [\cdot \text{tea} \cdot]\emptyset \\
&= \{P \mid P \stackrel{\text{tea}}{\rightarrow} P' \text{ implies } P' \in \emptyset, \text{ for each } P'\} \\
&= \{P \mid P \stackrel{\text{tea}}{\not\rightarrow}\} \ .
\end{aligned}$$

The last equality above follows from the fact that, for each process P,

$$P \stackrel{\text{tea}}{\not\rightarrow} \text{ iff } (P \stackrel{\text{tea}}{\rightarrow} P' \text{ implies } P' \in \emptyset, \text{ for each } P') \ .$$

To see that this holds, observe first of all that if $P \stackrel{\text{tea}}{\rightarrow} Q$ for some Q, then it is not true that $P' \in \emptyset$ for all P' such that $P \stackrel{\text{tea}}{\rightarrow} P'$. 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 $P \stackrel{\text{tea}}{\not\rightarrow}$. Then it is vacuously true that $P' \in \emptyset$ for all P' such that $P \stackrel{\text{tea}}{\rightarrow} P'$—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 $a \in \mathsf{Act}$ with the formula $[a]ff$.

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

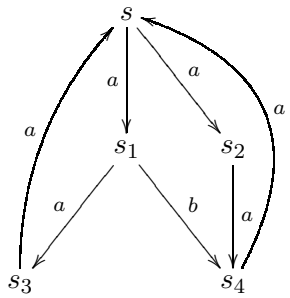
$$[\text{coffee}]\langle\text{biscuit}\rangle tt \quad .$$

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 $\langle a\rangle ff$ and $[a]tt$ express?*

♦

Exercise 5.3 *Consider the following labelled transition system.*



1. *Decide whether the following statements hold:*

   - $s \stackrel{?}{\models} \langle a \rangle \mathit{tt}$,
   - $s \stackrel{?}{\models} \langle b \rangle \mathit{tt}$,
   - $s \stackrel{?}{\models} [a] \mathit{ff}$,
   - $s \stackrel{?}{\models} [b] \mathit{ff}$,
   - $s \stackrel{?}{\models} [a]\langle b \rangle \mathit{tt}$,
   - $s \stackrel{?}{\models} \langle a \rangle\langle b \rangle \mathit{tt}$,
   - $s \stackrel{?}{\models} [a]\langle a \rangle[a][b] \mathit{ff}$,
   - $s \stackrel{?}{\models} \langle a \rangle\big(\langle a \rangle \mathit{tt} \wedge \langle b \rangle \mathit{tt}\big)$,
   - $s \stackrel{?}{\models} [a]\big(\langle a \rangle \mathit{tt} \vee \langle b \rangle \mathit{tt}\big)$,
   - $s \stackrel{?}{\models} \langle a \rangle\big([b][a] \mathit{ff} \wedge \langle b \rangle \mathit{tt}\big)$, *and*
   - $s \stackrel{?}{\models} \langle a \rangle\big([a](\langle a \rangle \mathit{tt} \wedge [b] \mathit{ff}) \wedge \langle b \rangle \mathit{ff}\big)$.

2. *Compute the following sets using the denotational semantics for Hennessy-Milner logic.*

   - $[\![[a][b] \mathit{ff}]\!] = ?$
   - $[\![\langle a \rangle\big(\langle a \rangle \mathit{tt} \wedge \langle b \rangle \mathit{tt}\big)]\!] = ?$
   - $[\![[a][a][b] \mathit{ff}]\!] = ?$
   - $[\![[a]\big(\langle a \rangle \mathit{tt} \vee \langle b \rangle \mathit{tt}\big)]\!] = ?$

♦

Exercise 5.4 *Consider an everlasting clock whose behaviour is defined thus:*

$$\mathit{Clock} \stackrel{\mathit{def}}{=} \mathit{tick}.\mathit{Clock} \ .$$

Prove that the process Clock satisfies the formula

$$[\mathit{tick}](\langle \mathit{tick} \rangle \mathit{tt} \wedge [\mathit{tock}] \mathit{ff}) \ .$$

Show also that, for each $n \geq 0$, the process Clock satisfies the formula

$$\underbrace{\langle \mathit{tick} \rangle \cdots \langle \mathit{tick} \rangle}_{n\text{-times}} \mathit{tt} \ .$$

♦

Exercise 5.5 (Mandatory) *Find a formula in $\mathcal{M}$ that is satisfied by $a.b.\mathbf{0} + a.c.\mathbf{0}$, but not by $a.(b.\mathbf{0} + c.\mathbf{0})$.*

Find a formula in $\mathcal{M}$ that is satisfied by $a.(b.c.\mathbf{0} + b.d.\mathbf{0})$, but not by $a.b.c.\mathbf{0} + a.b.d.\mathbf{0}$. ♦

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:

- $P \models \mathit{tt}$, for each P,
- $P \models \mathit{ff}$, for no P,
- $P \models F \wedge G$ iff $P \models F$ and $P \models G$,
- $P \models F \vee G$ iff $P \models F$ or $P \models G$,
- $P \models \langle a \rangle F$ iff $P \stackrel{a}{\rightarrow} P'$ for some P' such that $P' \models F$, and
- $P \models [a]F$ iff $P' \models F$, for each P' such that $P \stackrel{a}{\rightarrow} P'$.

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:*

- $\langle a \rangle (\langle b \rangle \langle c \rangle \mathit{tt} \wedge \langle c \rangle \mathit{tt})$,
- $\langle a \rangle \langle b \rangle ([a]\mathit{ff} \wedge [b]\mathit{ff} \wedge [c]\mathit{ff})$, *and*
- $[a]\langle b \rangle ([c]\mathit{ff} \wedge \langle a \rangle \mathit{tt})$.

♦

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. $tt^c = ff$
2. $ff^c = tt$
3. $(F \wedge G)^c = F^c \vee G^c$
4. $(F \vee G)^c = F^c \wedge G^c$
5. $(\langle a \rangle F)^c = [a]F^c$
6. $([a]F)^c = \langle a \rangle F^c$.

Note, for instance, that

$$\begin{array}{rcl} (\langle a \rangle tt)^c & = & [a]ff \quad \text{and} \\ ([a]ff)^c & = & \langle a \rangle tt. \end{array}$$

Proposition 5.1 Let $(\mathsf{Proc}, \mathsf{Act}, \{\stackrel{a}{\rightarrow} \mid a \in \mathsf{Act}\})$ be a labelled transition system. Then, for every formula $F \in \mathcal{M}$, it holds that $[\![F^c]\!] = \mathsf{Proc} \setminus [\![F]\!]$.

Proof: The proposition can be proven by structural induction on F. The details are left as an exercise to the reader. □

Exercise 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

$$\begin{array}{rcl} \mathrm{A} & \stackrel{\text{def}}{=} & a.\mathrm{A} + a.\mathbf{0} \quad \text{and} \\ \mathrm{B} & \stackrel{\text{def}}{=} & a.a.\mathrm{B} + a.\mathbf{0} \ . \end{array}$$

These two processes are *not* strongly bisimilar. In fact, A affords the transition

$$\mathrm{A} \stackrel{a}{\rightarrow} \mathrm{A} \ .$$

This transition can only be matched by either

$$\mathrm{B} \stackrel{a}{\rightarrow} \mathbf{0}$$

or

$$\mathrm{B} \stackrel{a}{\rightarrow} a.\mathrm{B} \ .$$

However, neither $\mathbf{0}$ nor $a.\mathrm{B}$ is strongly bisimilar to A, because this process can perform an a-labelled transition and become $\mathbf{0}$ in doing so. On the other hand,

$$a.\mathrm{B} \stackrel{a}{\rightarrow} \mathrm{B}$$

is the only transition that is possible from $a.\mathrm{B}$, and B is not strongly bisimilar to $\mathbf{0}$.

Based on this analysis, it seems that a property distinguishing the processes A and B is $\langle a\rangle\langle a\rangle[a]ff$—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 a satisfying, 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 $\{P' \mid P \stackrel{a}{\rightarrow} P'\}$ is finite for each action a.

An LTS is image finite if so is each of its states. ♦

For example, the process $\mathrm{A}_{\mathrm{rep}}$ (for 'A replicated') defined thus:

$$\mathrm{A}_{\mathrm{rep}} \stackrel{\mathrm{def}}{=} a.\mathbf{0} \mid \mathrm{A}_{\mathrm{rep}}$$

is *not* image finite. In fact, you should be able to prove by induction on n that, for each $n \geq 1$,

$$\mathrm{A}_{\mathrm{rep}} \stackrel{a}{\rightarrow} \underbrace{a.\mathbf{0} \mid \cdots \mid a.\mathbf{0}}_{n \text{ times}} \mid \mathbf{0} \mid \mathrm{A}_{\mathrm{rep}} \ .$$

Another example of a process that is not image finite is

$$\mathrm{A}^{<\omega} \stackrel{\text{def}}{=} \sum_{i \geq 0} a^i \ , \tag{5.1}$$

where $a^0 = \mathbf{0}$ 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

$$(\mathsf{Proc}, \mathsf{Act}, \{\stackrel{a}{\rightarrow} | \ a \in \mathsf{Act}\})$$

be an image finite LTS. Assume that P, Q are states in 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 $F \in \mathcal{M}$. 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 $\mathcal{M}$.

  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 $Q \stackrel{a}{\rightarrow} Q'$ for some Q'. We wish to show that $Q' \models G$. Now, since $P \sim Q$ and $Q \stackrel{a}{\rightarrow} Q'$, there is a process P' such that $P \stackrel{a}{\rightarrow} P'$ 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 $Q \stackrel{a}{\rightarrow} Q'$ 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 $\mathcal{M}$. We shall prove that P and Q are strongly bisimilar. To this end, note that it is sufficient to show that the relation

  $$\mathcal{R} = \{(R, S) \mid R, S \in \mathsf{Proc} \text{ satisfy the same formulae in } \mathcal{M}\}$$

  is a strong bisimulation.

  Assume that $R \,\mathcal{R}\, S$ and $R \stackrel{a}{\rightarrow} R'$. We shall now argue that there is a process S' such that $S \stackrel{a}{\rightarrow} S'$ 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 $S \stackrel{a}{\rightarrow} S'$ 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

$$R' \models F_i \text{ and } S_i \not\models F_i \ .$$

(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

$$\langle a \rangle (F_1 \wedge F_2 \wedge \cdots \wedge F_n)$$

is satisfied by R, but not by S. The easy verification is left to the reader.

The proof of the theorem is now complete. □

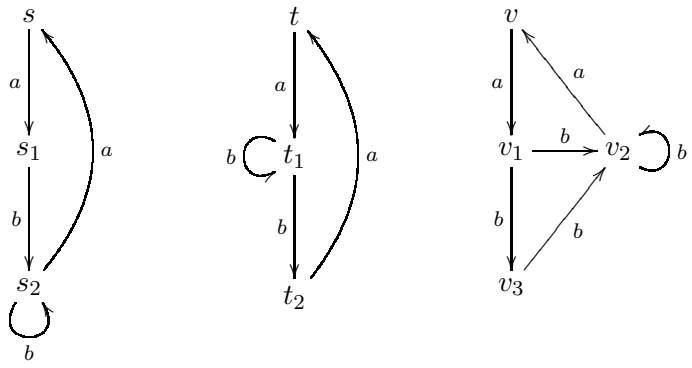
Exercise 5.9 (Mandatory) *Fill in the details that we have omitted in the above proof. What is the formula that we have constructed to distinguish R and S in the proof of the implication from right to left if $n = 0$?* ⧫

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 τ, and whose transitions are precisely the 'weak transitions' whose labels are either observable actions or τ.

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 `strongeq` *and* `checkprop` *commands) and check whether you found the shortest distinguishing formula (use the* `dfstrong` *command).* ⧫

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:*

- $b.a.\mathbf{0} + b.\mathbf{0}$ *and* $b.(a.\mathbf{0} + b.\mathbf{0})$,
- $a.(b.c.\mathbf{0} + b.d.\mathbf{0})$ *and* $a.b.c.\mathbf{0} + a.b.d.\mathbf{0}$,
- $a.\mathbf{0} \mid b.\mathbf{0}$ *and* $a.b.\mathbf{0} + b.a.\mathbf{0}$, *and*
- $(a.\mathbf{0} \mid b.\mathbf{0}) + c.a.\mathbf{0}$ *and* $a.\mathbf{0} \mid (b.\mathbf{0} + c.\mathbf{0})$.

Verify your claims in the Edinburgh Concurrency Workbench (use the `strongeq` *and* `checkprop` *commands) and check whether you found the shortest distinguishing formula (use the* `dfstrong` *command).* ⧫

Exercise 5.12 (For the theoretically minded) *Let* $(\mathsf{Proc}, \mathsf{Act}, \{\stackrel{a}{\rightarrow} \mid a \in \mathsf{Act}\})$ *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^ω given by:*

$$A^\omega \stackrel{def}{=} a.A^\omega \ .$$

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^ω satisfies a formula $F \in \mathcal{M}$ iff so does a^i, where i 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 $([a]\langle a\rangle ff) \vee \langle b\rangle tt$ 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 like, 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:

$$\begin{array}{lcl} p & \models & [a]\langle a\rangle tt \quad \text{but} \\ q & \not\models & [a]\langle a\rangle tt \ . \end{array}$$

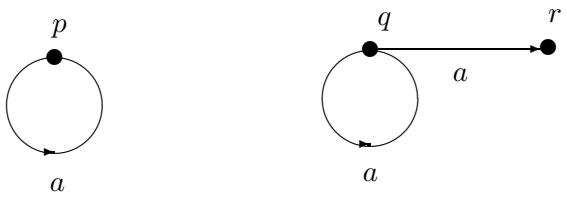


Figure 6.1: Two processes

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:

$$r = r_0 \stackrel{a}{\rightarrow} r_1 \stackrel{a}{\rightarrow} r_2 \stackrel{a}{\rightarrow} r_3 \cdots r_{n-1} \stackrel{a}{\rightarrow} r_n \quad (n \geq 0) \ .$$

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

$$\begin{array}{lcl} p & \models & [a]^{n+1}\langle a\rangle tt \quad \text{but} \\ q & \not\models & [a]^{n+1}\langle a\rangle tt \ , \end{array}$$

where $[a]^{n+1}$ stands for a sequence of modal operators $[a]$ 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, $\langle a\rangle tt$ is always true. Let us call this *invariance* property $Inv(\langle a\rangle tt)$. We could describe it in an extension of HML as an infinite conjunction thus:

$$Inv(\langle a\rangle tt) = \langle a\rangle tt \wedge [a]\langle a\rangle tt \wedge [a][a]\langle a\rangle tt \wedge \cdots = \bigwedge_{i\geq 0} [a]^i\langle a\rangle tt \ .$$

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 $\langle a\rangle tt$), 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 $[a]^i\langle a\rangle tt$ 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:

$$Pos([a]ff) = [a]ff \vee \langle a\rangle[a]ff \vee \langle a\rangle\langle a\rangle[a]ff \vee \cdots = \bigvee_{i\geq 0}\langle a\rangle^i[a]ff \ ,$$

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 $Inv(\langle a\rangle tt)$ by means of the following recursive equation:

$$X \quad \equiv \quad \langle a\rangle tt \wedge [a]X \ , \tag{6.1}$$

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

$$x = x + 1 \tag{6.2}$$

has no solution over the set of natural numbers, and there is no $X \subseteq \mathbb{N}$ such that

$$X = \mathbb{N} \setminus X \ . \tag{6.3}$$

On the other hand, there are uncountably many $X \subseteq \mathbb{N}$ such that

$$X = \{2\} \cup X \ , \tag{6.4}$$

namely all of the sets of natural numbers 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

$$X = \{10\} \cup \{n - 1 \mid n \in X,\ n \neq 0\} \ . \tag{6.5}$$

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 on page 88. 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:

$$S \;=\; \langle\cdot a\cdot\rangle \mathsf{Proc} \cap [\cdot a\cdot] S \;\; .$$

It is clear that $S = \emptyset$ is a solution to the equation (as no process can satisfy both $\langle a\rangle tt$ and $[a]ff$). However, the process p on Figure 6.1 can perform an a-transition invariantly and $p \notin \emptyset$, so this cannot be the solution we are looking for. Actually it turns out that it is the *largest* solution we need here, namely $S = \{p\}$. 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 $Pos([a]ff)$ by the following equation:

$$Y \;\equiv\; [a]ff \vee \langle a\rangle Y \;\; .$$

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 of the above equation 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 $Pos([a]ff)$.

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 $Inv(\langle a\rangle tt)$ we want the largest solution, and in this case we write

$$X \;\stackrel{\max}{=}\; \langle a\rangle tt \wedge [a] X \;\; .$$

For $Pos([a]ff)$ we will write

$$Y \;\stackrel{\min}{=}\; [a]ff \vee \langle a\rangle Y \;\; .$$

More generally we can express that the formula F holds for each reachable state in a labelled transition system having set of actions Act (written $Inv(F)$, and read

'invariantly F') by means of the equation

$$X \stackrel{\max}{=} F \wedge [\mathsf{Act}]X \ ,$$

and that F possibly holds at some point (written $Pos(F)$) by

$$Y \stackrel{\min}{=} F \vee \langle \mathsf{Act} \rangle Y \ .$$

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 $Inv(\langle a \rangle tt)$ 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 $Pos(\langle a \rangle tt)$ 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

$$p = p_0 \stackrel{a_1}{\rightarrow} p_1 \stackrel{a_2}{\rightarrow} p_2 \cdots \ ,$$

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:

$$X \stackrel{\max}{=} F \wedge ([\mathsf{Act}]ff \vee \langle \mathsf{Act} \rangle X) \ .$$

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 $[\mathsf{Act}]\mathit{ff}$) 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 $Even(F)$, 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:

$$Y \stackrel{\min}{=} F \vee (\langle \mathsf{Act} \rangle \mathit{tt} \wedge [\mathsf{Act}]Y) \ .$$

In this case we are interested in the least solution because $Even(F)$ should only be satisfied by those processes that are guaranteed to reach a state satisfying F in all of their computation paths.

Note that the definitions of $Safe(F)$ and $Even(F)$, respectively $Inv(F)$ and $Pos(F)$, are mutually *dual*, i.e., they can be obtained from one another by replacing $\vee$ by $\wedge$, $[\mathsf{Act}]$ by $\langle \mathsf{Act} \rangle$ and $\stackrel{\min}{=}$ by $\stackrel{\max}{=}$. One can show that $\neg Inv(F) \equiv Pos(\neg F)$ and $\neg Safe(F) \equiv Even(\neg F)$, 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:

- $F \,\mathcal{U}^s\, G$, 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 traverses before this happens F must hold;
- $F\mathcal{U}^w G$, the so-called *weak until*, that says that F must hold in all states p traverses until it gets into a state where G holds (but maybe this will never happen!).

We express these operators as follows:

$$F\,\mathcal{U}^s\,G \overset{\min}{=} G \vee (F \wedge \langle \mathsf{Act} \rangle tt \wedge [\mathsf{Act}](F\,\mathcal{U}^s\,G)) \ , \text{ and}$$
$$F\,\mathcal{U}^w\,G \overset{\max}{=} G \vee (F \wedge [\mathsf{Act}](F\,\mathcal{U}^w\,G)) \ .$$

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 $Even(F)$ and $Inv(F)$. In fact, $Even(G) \equiv tt\,\mathcal{U}^s\,G$ and $Inv(F) \equiv F\,\mathcal{U}^w\,ff$.

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, Emerson and A.P. Sistla, 1986; Manna and Pnueli, 1992; Pnueli, 1977).

The modal μ-calculus (Kozen, 1983) 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 μ-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 study 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:

$$F ::= X \mid tt \mid ff \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \langle a \rangle F \mid [a]F \ .$$

Semantically a formula F (that may contain a variable X) is interpreted as a function $\mathcal{O}_F : 2^{\mathsf{Proc}} \to 2^{\mathsf{Proc}}$ that, given a set of processes that are assumed to satisfy X, gives us the set of processes that satisfy F.

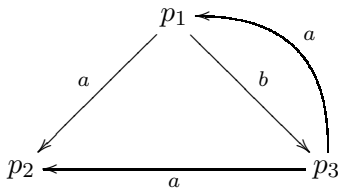


Figure 6.2: A process

Example 6.2 Consider the formula $F = \langle a \rangle X$ and let 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

$$\mathcal{O}_{\langle a \rangle X}(\{p_1\}) \;=\; \{p_3\} \;\; .$$

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

$$\mathcal{O}_{\langle a \rangle X}(\{p_1, p_2\}) \;=\; \{p_1, p_3\} \;\; .$$

What is the set $\mathcal{O}_{[b]X}(\{p_2\})$? ⧫

The above intuition is captured formally in the following definition.

Definition 6.1 Let $(\mathsf{Proc}, \mathsf{Act}, \{\stackrel{a}{\rightarrow} \mid a \in \mathsf{Act}\})$ be a labelled transition system. For each $S \subseteq \mathsf{Proc}$ and formula F, we define $\mathcal{O}_F(S)$ inductively as follows:

$$\begin{array}{rcl}
\mathcal{O}_X(S) & = & S \\
\mathcal{O}_{tt}(S) & = & \mathsf{Proc} \\
\mathcal{O}_{ff}(S) & = & \emptyset \\
\mathcal{O}_{F_1 \wedge F_2}(S) & = & \mathcal{O}_{F_1}(S) \cap \mathcal{O}_{F_2}(S) \\
\mathcal{O}_{F_1 \vee F_2}(S) & = & \mathcal{O}_{F_1}(S) \cup \mathcal{O}_{F_2}(S) \\
\mathcal{O}_{\langle a \rangle F}(S) & = & \langle \cdot a \cdot \rangle \mathcal{O}_F(S) \\
\mathcal{O}_{[a]F}(S) & = & [\cdot a \cdot] \mathcal{O}_F(S) \;\; .
\end{array}$$

⧫

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 tt 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* $\mathcal{O}_{[b]ff \wedge [a]X}(\{p_2\})$. ♦

One can show that for every formula F, the function $\mathcal{O}_F$ is *monotonic* (see Definition 4.4) over the complete lattice $(2^{\mathsf{Proc}}, \subseteq)$. In other words, for all subsets S_1, S_2 of Proc, if $S_1 \subseteq S_2$ then $\mathcal{O}_F(S_1) \subseteq \mathcal{O}_F(S_2)$.

Exercise 6.5 *Show that* $\mathcal{O}_F$ *is monotonic for all* F. *Consider what will happen if we introduce negation into our logic. Hint: Use structural induction on* F. ♦

As mentioned before, the idea underlying the definition of the function $\mathcal{O}_F$ is that if $[\![X]\!] \subseteq \mathsf{Proc}$ gives the set of processes that satisfy X, then $\mathcal{O}_F([\![X]\!])$ 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

$$X \stackrel{\min}{=} F_X \quad \text{or} \quad X \stackrel{\max}{=} F_X \ .$$

As shown in the previous section, such an equation can be interpreted as the set equation

$$[\![X]\!] = \mathcal{O}_{F_X}([\![X]\!]) \ . \tag{6.6}$$

As $\mathcal{O}_{F_X}$ is a monotonic function over a complete lattice we know that (6.6) has solutions, i.e., that $\mathcal{O}_{F_X}$ 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 $\mathrm{FIX}\ \mathcal{O}_{F_X}$, and also a unique *least* one, denoted by $\mathrm{fix}\ \mathcal{O}_{F_X}$, given respectively by

$$\begin{aligned} \mathrm{FIX}\ \mathcal{O}_{F_X} &= \bigcup \{S \subseteq \mathsf{Proc} \mid S \subseteq \mathcal{O}_{F_X}(S)\} \quad \text{and} \\ \mathrm{fix}\ \mathcal{O}_{F_X} &= \bigcap \{S \subseteq \mathsf{Proc} \mid \mathcal{O}_{F_X}(S) \subseteq S\} \ . \end{aligned}$$

A set S with the property that $S \subseteq \mathcal{O}_{F_X}(S)$ is called a *post-fixed point* for $\mathcal{O}_{F_X}$. Correspondingly S is a *pre-fixed point* for $\mathcal{O}_{F_X}$ if $\mathcal{O}_{F_X}(S) \subseteq S$.

In what follows, for a function $f : 2^{\mathsf{Proc}} \longrightarrow 2^{\mathsf{Proc}}$ we define

$$f^0 = \text{id} \quad (\text{the identity function on } 2^{\mathsf{Proc}}), \text{ and}$$
$$f^{m+1} = f \circ f^m \ .$$

When Proc is finite we have the following characterization of the largest and least fixed points.

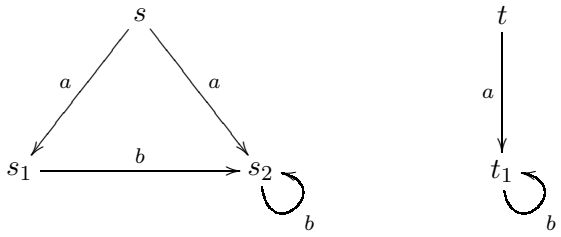
Theorem 6.1 If Proc is finite then $\text{FIX}\, \mathcal{O}_{F_X} = (\mathcal{O}_{F_X})^M(\mathsf{Proc})$ for some M and $\text{fix}\, \mathcal{O}_{F_X} = (\mathcal{O}_{F_X})^m(\emptyset)$ for some m.

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

$$X \stackrel{\max}{=} F_X \ ,$$

where $F_X = \langle b \rangle tt \wedge [b]X$. The set of processes in the labelled transition system



that satisfy this property is the largest solution to the equation

$$[\![X]\!] = (\langle \cdot b \cdot \rangle \{s, s_1, s_2, t, t_1\}) \cap [\cdot b \cdot][\![X]\!] \ .$$

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

$$\mathcal{O}_{F_X}(S) = (\langle \cdot b \cdot \rangle \{s, s_1, s_2, t, t_1\}) \cap [\cdot b \cdot] S \ .$$

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

$$\begin{aligned}\mathcal{O}_{F_X}(\{s,s_1,s_2,t,t_1\}) &= (\langle\cdot b\cdot\rangle\{s,s_1,s_2,t,t_1\}) \cap [\cdot b\cdot]\{s,s_1,s_2,t,t_1\} \\ &= \{s_1,s_2,t_1\} \cap \{s,s_1,s_2,t,t_1\} \\ &= \{s_1,s_2,t_1\} \ .\end{aligned}$$

Note that our candidate solution to the equation has shrunk in size, since an application of $\mathcal{O}_{F_X}$ to the set of all processes has removed the states s and t from our candidate solution. Intuitively, this is because, by applying $\mathcal{O}_{F_X}$ to the set of all states, we have found a reason why s and t do *not* afford the property specified by

$$X \stackrel{\max}{=} \langle b\rangle tt \wedge [b]X \ ,$$

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 the largest fixed point, we now apply the function $\mathcal{O}_{F_X}$ to the new candidate largest solution, namely $\{s_1,s_2,t_1\}$. We now have that

$$\begin{aligned}\mathcal{O}_{F_X}(\{s_1,s_2,t_1\}) &= (\langle\cdot b\cdot\rangle\{s,s_1,s_2,t,t_1\}) \cap [\cdot b\cdot]\{s_1,s_2,t_1\} \\ &= \{s_1,s_2,t_1\} \cap \{s,s_1,s_2,t,t_1\} \\ &= \{s_1,s_2,t_1\} \ .\end{aligned}$$

(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 $\mathcal{O}_{F_X}$. 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

$$X \stackrel{\max}{=} \langle b\rangle tt \wedge [b]X \ .$$

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*

$$Y \stackrel{\min}{=} \langle b\rangle tt \vee \langle\{a,b\}\rangle Y \ .$$

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

$$X \stackrel{\max}{=} F_X \ ,$$

and define $[\![X]\!] \subseteq \mathsf{Proc}$ by

$$[\![X]\!] \ = \ \mathrm{FIX}\, \mathcal{O}_{F_X} \ .$$

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 $Inv(F)$ is obtained as the largest fixed point to the recursive equation

$$X = F \wedge [\mathsf{Act}]X \ .$$

We will now show that $Inv(F)$ defined in this way indeed expresses that F *holds at all states in all transitions sequences.*

For this purpose we let $\mathcal{I} : 2^{\mathsf{Proc}} \longrightarrow 2^{\mathsf{Proc}}$ be the corresponding semantic function, i.e.,

$$\mathcal{I}(S) = [\![F]\!] \cap [\cdot \mathsf{Act} \cdot]S \ .$$

By Tarski´s fixed point theorem this equation has exactly one largest solution given by

$$\mathrm{FIX}\, \mathcal{I} = \bigcup \{S \mid S \subseteq \mathcal{I}(S)\} \ .$$

To show that $\mathrm{FIX}\, \mathcal{I}$ 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 Inv defined as follows:

$$Inv = \{p \mid p \stackrel{\sigma}{\rightarrow} p' \text{ implies } p' \in [\![F]\!], \text{ for each } \sigma \in \mathsf{Act}^* \text{ and } p' \in \mathsf{Proc}\} \ .$$

The correctness of $Inv(F)$ with respect to this description can now be formulated as follows.

Theorem 6.2 For every labelled transition system $(\mathsf{Proc}, \mathsf{Act}, \{ \stackrel{a}{\rightarrow} \mid a \in \mathsf{Act}\})$, it holds that $Inv \ = \ \mathrm{FIX}\, \mathcal{I}$.

Proof: We show the statement by proving each of the inclusions $Inv \subseteq \mathrm{FIX}\, \mathcal{I}$ and $\mathrm{FIX}\, \mathcal{I} \subseteq Inv$ separately.

$Inv \subseteq \text{FIX}\,\mathcal{I}$: To prove this inclusion it is sufficient to show that $Inv \subseteq \mathcal{I}(Inv)$ (Why?). To this end, let $p \in Inv$. Then, for all $\sigma \in \mathsf{Act}^*$ and $p' \in \mathsf{Proc}$,

$$p \xrightarrow{\sigma} p' \text{ implies that } p' \in [\![F]\!] \ . \tag{6.7}$$

We must establish that $p \in \mathcal{I}(Inv)$, or equivalently that $p \in [\![F]\!]$ and that $p \in [\cdot\mathsf{Act}\cdot]Inv$. We obtain the first one of these two statements by taking $\sigma = \varepsilon$ in (6.7) because $p \xrightarrow{\varepsilon} p$ always holds.

To prove that $p \in [\cdot\mathsf{Act}\cdot]Inv$, we have to show that, for each process p' and action a,

$p \xrightarrow{a} p'$ implies $p' \in Inv$.

This is equivalent to proving that, for each sequence of actions σ' and process p'',

$p \xrightarrow{a} p'$ and $p' \xrightarrow{\sigma'} p''$ imply $p'' \in [\![F]\!]$.

However, this follows immediately by letting $\sigma = a\sigma'$ in (6.7).

$\text{FIX}\,\mathcal{I} \subseteq Inv$: First we note that, since $\text{FIX}\,\mathcal{I}$ is a fixed point of $\mathcal{I}$, it holds that

$$\text{FIX}\,\mathcal{I} = [\![F]\!] \cap [\cdot\mathsf{Act}\cdot]\text{FIX}\,\mathcal{I} \ . \tag{6.8}$$

To prove that $\text{FIX}\,\mathcal{I} \subseteq Inv$, assume that $p \in \text{FIX}\,\mathcal{I}$ and that $p \xrightarrow{\sigma} p'$. We shall show that $p' \in [\![F]\!]$ by induction on $|\sigma|$, the length of σ.

Base case $\sigma = \varepsilon$: Then $p = p'$ and therefore, by (6.8) and our assumption that $p \in \text{FIX}\,\mathcal{I}$, it holds that $p' \in [\![F]\!]$, which was to be shown.

Inductive step $\sigma = a\sigma'$: Then $p \xrightarrow{a} p'' \xrightarrow{\sigma'} p'$ for some p''. By (6.8) and our assumption that $p \in \text{FIX}\,\mathcal{I}$, it follows that $p'' \in \text{FIX}\,\mathcal{I}$. As $|\sigma'| < |\sigma|$ and $p'' \in \text{FIX}\,\mathcal{I}$, 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. □

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

$$F ::= X \mid tt \mid ff \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid \langle a\rangle F \mid [a]F \ ,$$

where $a \in \mathsf{Act}$ and there is exactly one defining equation for the variable X, which is of the form

$$X \stackrel{\min}{=} F_X$$

or

$$X \stackrel{\max}{=} F_X \ ,$$

where F_X is a formula of the logic which may contain occurrences of the variable X.

Let $(\mathsf{Proc}, \mathsf{Act}, \{\stackrel{a}{\rightarrow} \mid a \in \mathsf{Act}\})$ be a labelled transition system and F a formula of Hennessy-Milner logic with one (recursively defined) variable X. Let $s \in \mathsf{Proc}$. 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 $s \in \mathsf{Proc}$ 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 Proc):

- (s, tt) and (s, ff) have no successor configurations,
- $(s, F_1 \wedge F_2)$ and $(s, F_1 \vee F_2)$ both have two successor configurations, namely (s, F_1) and (s, F_2),
- $(s, \langle a \rangle F)$ 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 $X \stackrel{\max}{=} F_X$ or $X \stackrel{\min}{=} F_X$.

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 a successor configuration for every current configuration of the form $(s, F_1 \wedge F_2)$ and $(s, [a]F)$.
- The defender picks a successor configuration for every current configuration of the form $(s, F_1 \vee F_2)$ and $(s, \langle a \rangle F)$.

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 by $\stackrel{A}{\rightarrow}$ moves and those chosen by the defender by $\stackrel{D}{\rightarrow}$ moves.

We also notice that every play either

- terminates in (s, tt) or (s, ff), 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 $s \stackrel{a}{\not\rightarrow}$, 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, ff) 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, tt) 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 $X \stackrel{\min}{=} F_X$; the defender is a winner in every infinite play provided that X is defined via $X \stackrel{\max}{=} F_X$.

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 in finitely 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. ⧫

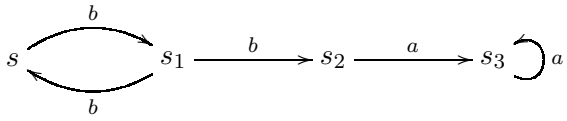
Theorem 6.3 [Game characterization] Let $(\mathsf{Proc}, \mathsf{Act}, \{\stackrel{a}{\rightarrow} | \ a \in \mathsf{Act}\})$ be a labelled transition system and F a formula of Hennessy-Milner logic with one (recursively defined) variable X. Let $s \in \mathsf{Proc}$. Then the following statements hold.

- State s satisfies F if and only if the defender has a universal winning strategy starting from (s, F).
- State s does not satisfy 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 $s \models [b](\langle b\rangle[b]ff \wedge \langle b\rangle[a]ff)$ by defining a universal winning strategy for the defender. As remarked before, we will use $\stackrel{A}{\rightarrow}$ to denote that the successor configuration was selected by the attacker and $\stackrel{D}{\rightarrow}$ to denote that it was selected by the defender. The game starts from

$$(s, [b](\langle b\rangle[b]ff \wedge \langle b\rangle[a]ff)) \ .$$

Because $[b]$ is the topmost operation, the attacker selects the successor configuration and he has only one possibility, namely

$$(s, [b](\langle b\rangle[b]ff \wedge \langle b\rangle[a]ff)) \stackrel{A}{\rightarrow} (s_1, \langle b\rangle[b]ff \wedge \langle b\rangle[a]ff) \ .$$

Now the topmost operation is $\wedge$ so the attacker has two possibilities:

$$(s_1, \langle b\rangle[b]ff \wedge \langle b\rangle[a]ff) \stackrel{A}{\rightarrow} (s_1, \langle b\rangle[b]ff)$$

or

$$(s_1, \langle b\rangle[b]ff \wedge \langle b\rangle[a]ff) \stackrel{A}{\rightarrow} (s_1, \langle b\rangle[a]ff) \ .$$

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 $(s_1, \langle b\rangle[b]ff) \stackrel{D}{\rightarrow} (s_2, [b]ff)$. Now the attacker should continue but $s_2 \stackrel{b}{\not\rightarrow}$ 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 $(s_1, \langle b\rangle[a]ff) \stackrel{D}{\rightarrow} (s, [a]ff)$. Now the attacker should continue but $s \stackrel{a}{\not\rightarrow}$ so he is stuck again and the defender wins this play.

Hence the defender has a universal winning strategy. ⧫

Example 6.4 Let $X \overset{\min}{=} \langle a \rangle tt \vee \langle b \rangle X$. 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 $\overset{D}{\rightarrow}$ or the referee's $\rightarrow$ moves for expanding the variable X, so it is truly a universal winning strategy):

$$(s, X) \rightarrow (s, \langle a \rangle tt \vee \langle b \rangle X) \overset{D}{\rightarrow} (s, \langle b \rangle X) \overset{D}{\rightarrow} (s_1, X)$$

$$\rightarrow (s_1, \langle a \rangle tt \vee \langle b \rangle X) \overset{D}{\rightarrow} (s_1, \langle b \rangle X) \overset{D}{\rightarrow} (s_2, X)$$

$$\rightarrow (s_2, \langle a \rangle tt \vee \langle b \rangle X) \overset{D}{\rightarrow} (s_2, \langle a \rangle tt) \overset{D}{\rightarrow} (s_3, tt) \ .$$

According to the definition (s_3, tt) is a winning configuration for the defender. ⧫

Example 6.5 Let $X \overset{\max}{=} \langle b \rangle tt \wedge [b] X$. 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.

$$(s, X) \rightarrow (s, \langle b \rangle tt \wedge [b] X) \overset{A}{\rightarrow} (s, [b] X) \overset{A}{\rightarrow} (s_1, X)$$

$$\rightarrow (s_1, \langle b \rangle tt \wedge [b] X) \overset{A}{\rightarrow} (s_1, [b] X) \overset{A}{\rightarrow} (s_2, X)$$

$$\rightarrow (s_2, \langle b \rangle tt \wedge [b] X) \overset{A}{\rightarrow} (s_2, \langle b \rangle tt) \ .$$

From the last configuration $(s_2, \langle b \rangle tt)$ the defender is supposed to continue but he is stuck as $s_2 \overset{b}{\not\rightarrow}$ and hence the attacker wins. ⧫

Example 6.6 Let $X \overset{\max}{=} \langle a \rangle tt \wedge [a] X$. 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 $(s_2, X) \rightarrow (s_2, \langle a \rangle tt \wedge [a] X)$ and in the second round the attacker can play either

$$(s_2, \langle a \rangle tt \wedge [a] X) \overset{A}{\rightarrow} (s_2, \langle a \rangle tt)$$

or

$$(s_2, \langle a \rangle tt \wedge [a] X) \overset{A}{\rightarrow} (s_2, [a] X) \ .$$

It is easy to see that the defender wins from the configuration $(s_2, \langle a \rangle tt)$ by the move $(s_2, \langle a \rangle tt) \stackrel{D}{\rightarrow} (s_3, tt)$, so we shall investigate only the continuation of the game from $(s_2, [a]X)$. The attacker has only the move $(s_2, [a]X) \stackrel{A}{\rightarrow} (s_3, X)$. After expanding the variable X the game continues from $(s_3, \langle a \rangle tt \wedge [a]X)$. Again the attacker can play either

$$(s_3, \langle a \rangle tt \wedge [a]X) \stackrel{A}{\rightarrow} (s_3, \langle a \rangle tt)$$

or

$$(s_3, \langle a \rangle tt \wedge [a]X) \stackrel{A}{\rightarrow} (s_3, [a]X) \ .$$

In the first case the attacker loses as before. In the second case, the only continuation of the game is $(s_3, [a]X) \stackrel{A}{\rightarrow} (s_3, X)$. 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. ♦

Example 6.7 Let $X \stackrel{\min}{=} \langle a \rangle tt \vee ([b]X \wedge \langle b \rangle tt)$. 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

$$(s_1, X) \rightarrow (s_1, \langle a \rangle tt \vee ([b]X \wedge \langle b \rangle tt)) \ ,$$

and then the defender has two options, namely

$$(s_1, \langle a \rangle tt \vee ([b]X \wedge \langle b \rangle tt)) \stackrel{D}{\rightarrow} (s_1, \langle a \rangle tt)$$

or

$$(s_1, \langle a \rangle tt \vee ([b]X \wedge \langle b \rangle tt)) \stackrel{D}{\rightarrow} (s_1, [b]X \wedge \langle b \rangle tt) \ .$$

In the first case the defender loses as he is supposed to pick an a-successor of the state s_1 but $s_1 \stackrel{a}{\not\rightarrow}$. In the second case the attacker proceeds as follows.

$$(s_1, [b]X \wedge \langle b \rangle tt) \stackrel{A}{\rightarrow} (s_1, [b]X) \stackrel{A}{\rightarrow} (s, X) \ .$$

The game now continues from (s, X) by the move

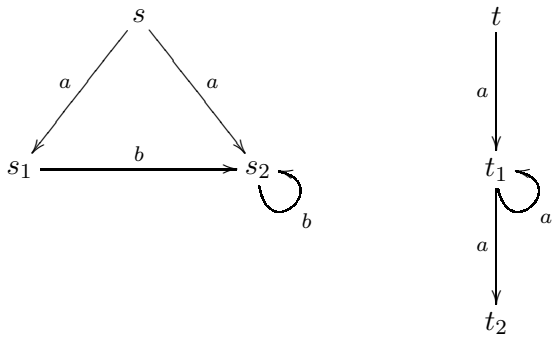
$$(s, X) \rightarrow (s, \langle a \rangle tt \vee ([b]X \wedge \langle b \rangle tt)) \ .$$

Again, if the defender plays $(s, \langle a \rangle tt \vee ([b]X \wedge \langle b \rangle tt)) \stackrel{D}{\rightarrow} (s, \langle a \rangle tt)$ then he loses in the next round, so the defender has to play

$$(s, \langle a \rangle tt \vee ([b]X \wedge \langle b \rangle tt)) \stackrel{D}{\rightarrow} (s, [b]X \wedge \langle b \rangle tt) \ .$$

The attacker continues by $(s, [b]X \wedge \langle b \rangle tt) \xrightarrow{A} (s, [b]X) \xrightarrow{A} (s_1, X)$ 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*



Use the game characterization for HML with recursion to show that

1. s_1 *satisfies the formula*

$$X \stackrel{\max}{=} \langle b \rangle tt \wedge [b]X \ ;$$

2. s *satisfies the formula*

$$Y \stackrel{\min}{=} \langle b \rangle tt \vee \langle \{a, b\} \rangle Y \ ,$$

   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 of formulae. However, as is the case in the specification of process behaviours in CCS, it is sometimes useful, when not altogether necessary, to define formulae recursively using two or more variables.

By way of example, consider the following property:

> It is always the case that a process can perform an a-labelled transition leading to a state where b-transitions can be executed forever.

Using the template for the specification of invariance properties we presented on page 120, we can specify the above requirement in terms of the following recursive equation

$$Inv(\langle a\rangle \text{Forever}(b)) \stackrel{\max}{=} \langle a\rangle \text{Forever}(b) \wedge [\mathsf{Act}] Inv(\langle a\rangle \text{Forever}(b)) \ ,$$

where the formula $\text{Forever}(b)$ is one that expresses that b-transitions can be executed forever. The formula $\text{Forever}(b)$ is, however, itself specified recursively thus:

$$\text{Forever}(b) \stackrel{\max}{=} \langle b\rangle \text{Forever}(b) \ .$$

Our informally specified requirement is therefore formally expressed by means of two recursive equations.

In general, a *mutually recursive equational system* has the form

$$\begin{aligned} X_1 &= F_{X_1} \\ &\vdots \\ X_n &= F_{X_n} \ , \end{aligned}$$

where $\mathcal{X} = \{X_1, \ldots, X_n\}$ is a set of variables and, for $1 \leq i \leq n$, 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

$$\begin{aligned} X &= [a]Y \\ Y &= \langle a\rangle X \ . \end{aligned}$$

An equational system is sometimes given by specifying a (finite) set of variables $\mathcal{X}$ together with a declaration. A *declaration* is a function $D : \mathcal{X} \to \mathcal{M}_{\mathcal{X}}$ that associates a formula with each variable—$D(X) = F_X$ in the notation used above. As in the example presented above, in this section we shall assume that either the largest solution is sought for all of the equations in an equational system (or the declaration that represents it) or the least solution is sought for all of them. This requirement will be lifted in Section 6.7, where we shall briefly discuss formulae specified by equational systems where largest and least fixed points can be mixed.

An equational system generalizes the recursive definition of a single formula in much the same way as, in standard algebra, a system of equations is a generalization of a single equation. The intended meaning of an equational system we now proceed to present formally is therefore, naturally enough, a generalization of the semantics of a single, recursively defined formula.

Recall that an equation of the form

$$X = F \ ,$$

where F is a formula in Hennessy-Milner logic that may contain occurrences of the variable X, is intended to describe a set of processes S such that

$$S = \mathcal{O}_F(S) \ .$$

As a natural generalization of this idea, an equational system like

$$\begin{aligned} X_1 &= F_{X_1} \\ &\vdots \\ X_n &= F_{X_n} \ , \end{aligned}$$

where each formula F_{X_i} may contain any of the variables $X_1, \ldots, X_n$, is meant to describe a vector of sets of processes $(S_1, \ldots, S_n)$ such that

$$\begin{aligned} S_1 &= \mathcal{O}_{F_{X_1}}(S_1, \ldots, S_n) \\ &\vdots \\ S_n &= \mathcal{O}_{F_{X_n}}(S_1, \ldots, S_n) \ . \end{aligned}$$

To define the semantics of an equational system over a set of variables $\mathcal{X}$, it is therefore 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 $\mathcal{D} = (2^{\mathsf{Proc}})^n$ (n-times cross product of 2^{Proc} with itself) with a partial order defined component wise:

$$(S_1, \ldots, S_n) \sqsubseteq (S'_1, \ldots, S'_n) \text{ if } S_1 \subseteq S'_1 \text{ and } S_2 \subseteq S'_2 \text{ and } \ldots \text{ and } S_n \subseteq S'_n \ .$$

$(\mathcal{D}, \sqsubseteq)$ defined in this way yields a complete lattice with the least upper bound and the greatest lower bound also defined component wise:

$$\bigsqcup\{(A^i_1, \ldots, A^i_n) \mid i \in I\} = (\bigcup\{A^i_1 \mid i \in I\}, \ldots, \bigcup\{A^i_n \mid i \in I\}) \text{ and}$$

$$\bigsqcap\{(A^i_1, \ldots, A^i_n) \mid i \in I\} = (\bigcap\{A^i_1 \mid i \in I\}, \ldots, \bigcap\{A^i_n \mid i \in I\}) \ ,$$

where I is an index set.

Let D be a declaration over the set of variables $\mathcal{X} = \{X_1, \ldots, X_n\}$ that associates a formula F_{X_i} with each variable X_i ($1 \leq i \leq n$). The semantic function $[\![D]\!] : \mathcal{D} \to \mathcal{D}$ 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:

$$\begin{aligned} &[\![D]\!](S_1, \ldots, S_n) = \\ &\quad (\mathcal{O}_{F_{X_1}}(S_1, \ldots, S_n), \ldots, \mathcal{O}_{F_{X_n}}(S_1, \ldots, S_n)) \ , \end{aligned} \tag{6.9}$$

where each argument S_i ($1 \leq i \leq n$) is an arbitrary subset of the set of processes Proc. By analogy with our previous developments, for each formula F in $\mathcal{M}_{\mathcal{X}}$, the set

$$\mathcal{O}_F(S_1, \ldots, S_n)$$

stands for the set of processes that satisfy F under the assumption that S_i 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 $\mathcal{O}_F(S_1, \ldots, S_n)$ is defined exactly as in Definition 6.1, but with

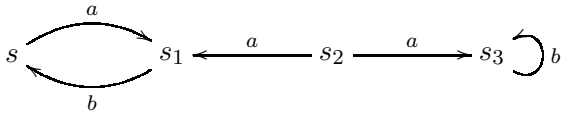
$$\mathcal{O}_{X_i}(S_1, \ldots, S_n) = S_i \quad (1 \leq i \leq n) \ .$$

The function $[\![D]\!]$ turns out to be monotonic over the complete lattice $(\mathcal{D}, \sqsubseteq)$, 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:

$$\begin{array}{lcl} X & \stackrel{\max}{=} & \langle a \rangle Y \wedge [a] Y \wedge [b] \mathit{ff} \\ Y & \stackrel{\max}{=} & \langle b \rangle X \wedge [b] X \wedge [a] \mathit{ff} \ . \end{array}$$

We wish to find out the set of states in the following labelled transition that satisfies the formula X.



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 sets of states

$$(\langle \cdot a \cdot \rangle S_2 \cap [\cdot a \cdot] S_2 \cap \{s, s_2\}, \langle \cdot b \cdot \rangle S_1 \cap [\cdot b \cdot] S_1 \cap \{s_1, s_3\}) \ . \tag{6.10}$$

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,
- $\langle \cdot a \cdot \rangle S_2 \cap [\cdot a \cdot] S_2 \cap \{s, s_2\}$ is the set of states that satisfy the right-hand side of the defining equation for X under these assumptions, and

- $\langle \cdot b \cdot \rangle S_1 \cap [\cdot b \cdot] S_1 \cap \{s_1, s_3\}$ is the set of states that satisfy 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

$$(\{s, s_1, s_2, s_3\}, \{s, s_1, s_2, s_3\}) \ .$$

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

$$(\{s, s_2\}, \{s_1, s_3\}) \ .$$

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 $((2^{\mathsf{Proc}})^n, \sqsubseteq)$, with $\sqsubseteq$, $\bigsqcup$ and $\bigsqcap$ defined as described in the text above, is a complete lattice.*

2. *Show that (6.9) defines a monotonic function*

$$[\![D]\!] : (2^{\mathsf{Proc}})^n \longrightarrow (2^{\mathsf{Proc}})^n \ .$$

3. *Compute the least and largest solutions of the system of equations*

$$\begin{aligned} X &= [a]Y \\ Y &= \langle a\rangle X \end{aligned}$$

over the transition system associated with the CCS term

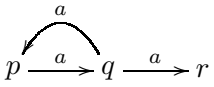
$$\begin{aligned} A_0 &= a.A_1 + a.a.0 \\ A_1 &= a.A_2 + a.0 \\ A_2 &= a.A_1 \ . \end{aligned}$$

♦

Exercise 6.9 *Compute the largest solution of the equational system*

$$\begin{aligned} X &= [a]Y \\ Y &= \langle a\rangle X \end{aligned}$$

over the following labelled transition system.



♦

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

$$[p]_\sim = \{q \mid p \models F \text{ implies } q \models F, \text{ for each } F \in \mathcal{M}\} \ .$$

Exercise 6.10 *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, following (Ingolfsdottir, Godskesen and Zeeberg, 1987; Steffen and Ingolfsdottir, 1994), 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. (See also (Graf and Sifakis, 1986) for a translation from CCS expressions describing terminating behaviours into modal formulae.) 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 $F_p \in \mathcal{M}_\mathcal{X}$ for a suitable set of variables $\mathcal{X}$, such that for all processes q

$$q \models F_p \text{ iff } q \sim p \ .$$

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 $\mathsf{Act} = \{a\}$ and that the process p is given by the equation

$$X \stackrel{\text{def}}{=} a.X \ .$$

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 $F_p \in \mathcal{M}$,

$$[\![F_p]\!] = [p]_\sim \ . \tag{6.11}$$

In particular we have that

$$p \models F_p \quad \text{and} \quad (q \models F_p \text{ implies } q \sim p, \text{ for each } q) \ . \tag{6.12}$$

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.

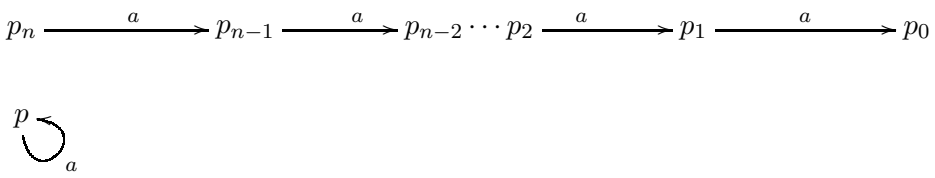


Figure 6.3: The processes p and p_n

Recall that, by the *modal depth* of a formula F, notation $md(F)$, we mean the maximum number of nested occurrences of the modal operators in F. Formally this is defined by the following recursive definition:

1. $md(tt) = md(ff) = 0$,
2. $md([a]F) = md(\langle a \rangle F) = 1 + md(F)$,
3. $md(F_1 \vee F_2) = md(F_1 \wedge F_2) = \max\{md(F_1), md(F_2)\}$.

Next we define a sequence $p_0, p_1, p_2, \ldots$ of processes inductively as follows:

1. $p_0 = \mathbf{0}$,
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:

$$p \models F \text{ implies } p_{md(F)} \models F, \text{ for each } F \ . \tag{6.13}$$

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. ♦

Exercise 6.11 *Prove statement (6.13).* ♦

Exercise 6.12 (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 $i \geq 0$, construct an HML formula that is a characteristic formula for process p_i in Figure 6.3. Hint: First give a characteristic formula for p_0. Next show how to construct a characteristic formula for p_{i+1} from that for p_i. ♦

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

$$(\{p_1, \ldots, p_n\}, \mathsf{Act}, \{\stackrel{a}{\rightarrow} | \ a \in \mathsf{Act}\})$$

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 actions 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.

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.

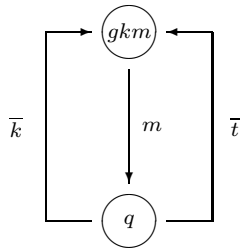


Figure 6.4: The coffee machine gkm

If we let X_{gkm} and X_q denote the characteristic formula for q and gkm respectively, X_{gkm} can be expressed as

$$X_{gkm} \equiv \langle m \rangle X_q \wedge [m] X_q \wedge [\{\overline{t}, \overline{k}\}] ff \ ,$$

where $\langle m \rangle X_q$ expresses property 1 above, $[m] X_q$ expresses property 2, and the last conjunct $[\{\overline{t}, \overline{k}\}] ff$ 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

$$X_q \equiv \langle \overline{t} \rangle X_{gkm} \wedge \langle \overline{k} \rangle X_{gkm} \wedge [\{\overline{t}, \overline{k}\}] X_{gkm} \wedge [m] ff \ .$$

In the recursive formula above, the first conjunct $\langle \overline{t} \rangle X_{gkm}$ states that a process that is bisimilar to q should be able to perform a $\overline{t}$-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 $\overline{t}$ or $\overline{k}$ 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

$$Der(a, p) = \{p' \mid p \xrightarrow{a} p'\}$$

be the set of states that can be reached from p by performing action a. If $p' \in Der(a, p)$ 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 $p \stackrel{a}{\rightarrow} p'$ then $p' \in Der(a,p)$. Therefore p has the property

$$[a]\Big(\bigvee_{p'.p \stackrel{a}{\rightarrow} p'} X_{p'}\Big) \ ,$$

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 $Der(a,p)$. (Note that if $p \stackrel{a}{\not\rightarrow}$, then $Der(a,p)$ is empty. In that case, since an empty disjunction is just the formula ff, the above formula becomes simply $[a]ff$—which is what we would expect.)

Since action a is arbitrary, we have that

$$p \models \bigwedge_a [a]\Big(\bigvee_{p'.p \stackrel{a}{\rightarrow} p'} X_{p'}\Big) \ .$$

If we summarize the above requirements, we have that

$$p \models \bigwedge_{a,p'.p \stackrel{a}{\rightarrow} p'} \langle a\rangle X_{p'} \ \wedge \ \bigwedge_a [a]\Big(\bigvee_{p'.p \stackrel{a}{\rightarrow} p'} X_{p'}\Big) \ .$$

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 $q \in \mathsf{Proc}$:

$$X_q = \bigwedge_{a,q'.q \stackrel{a}{\rightarrow} q'} \langle a\rangle X_{q'} \ \wedge \ \bigwedge_a [a]\Big(\bigvee_{q'.q \stackrel{a}{\rightarrow} q'} X_{q'}\Big) \ . \tag{6.14}$$

The solution can either be the least or the largest one (or, in fact, any other fixed point 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

$$X_p = \langle a\rangle X_p \wedge [a]X_p \ .$$

Since $\langle\cdot a\cdot\rangle\emptyset = \emptyset$, you should be able to convince yourselves that $[\![X_p]\!] = \emptyset$ is the least solution to this equation. This corresponds to taking $X_p = ff$ as the characteristic formula for p. However, p does not have the property ff, which therefore cannot be the characteristic property for p. ♦

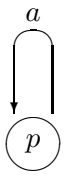


Figure 6.5: Simple infinite process p

In what follows we will show that the largest solution to (6.14) yields the characteristic property for all $p \in \mathsf{Proc}$. (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 $(\mathsf{Proc}, \mathsf{Act}, \{\stackrel{a}{\rightarrow} \mid a \in \mathsf{Act}\})$ be a finite transition system and, for each $p \in \mathsf{Proc}$, let X_p be defined by

$$X_p \stackrel{\max}{=} \bigwedge_{a,p'.p \stackrel{a}{\rightarrow} p'} \langle a \rangle X_{p'} \ \wedge \ \bigwedge_{a} [a] \Big(\bigvee_{p'.p \stackrel{a}{\rightarrow} p'} X_{p'} \Big) \ . \tag{6.15}$$

Then X_p is the characteristic property for p—that is, $q \models X_p$ iff $p \sim q$, for each $q \in \mathsf{Proc}$.

The assumption that Proc and 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

$$D_K(X_p) = \bigwedge_{a,p.p \stackrel{a}{\rightarrow} p'} \langle a \rangle X_{p'} \ \wedge \ \bigwedge_{a} [a] \Big(\bigvee_{p'.p \stackrel{a}{\rightarrow} p'} X_{p'} \Big) \ .$$

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 $p \in \mathsf{Proc}$, we have that $[\![X_p]\!] = [p]_\sim$. In what follows, we write $q \models_{max} X_p$ 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 $q \in \mathsf{Proc}$:

1. if $q \models_{max} X_p$, then $p \sim q$, and
2. if $p \sim q$, then $q \models_{max} X_p$.

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 $q \in \mathsf{Proc}$, we have that

$$q \models_{max} X_p \Rightarrow p \sim q \ .$$

Proof: Let $R = \{(p, q) \mid q \models_{max} X_p\}$. We will prove that R is a bisimulation, and thus that $p \sim q$ whenever $q \models_{max} X_p$. To this end, we have to prove the following two claims, where b is an arbitrary action in Act and p_1, q_1 are processes in 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

$$q \models_{max} X_p \ \text{ and } \ p \xrightarrow{b} p_1 \ .$$

From equation (6.15), it follows that

$$q \models_{max} \Big(\bigwedge_{a, p' . p \xrightarrow{a} p'} \langle a \rangle X_{p'} \Big) \wedge \Big(\bigwedge_{a} [a] \big(\bigvee_{p' . p \xrightarrow{a} p'} X_{p'} \big) \Big) \ .$$

As $p \xrightarrow{b} p_1$, we obtain, in particular, that $q \models_{max} \langle b \rangle X_{p_1}$, which means that, for some $q_1 \in \mathsf{Proc}$,

$$q \xrightarrow{b} q_1 \text{ and } q_1 \models_{max} X_{p_1} \ .$$

Using the definition of R, we have 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

$$q \models_{max} X_p \text{ and } q \xrightarrow{b} q_1 \ .$$

As before, since $q \models_{max} X_p$, we have that

$$q \models_{max} \Big(\bigwedge_{a,p'.p \xrightarrow{a} p'} \langle a \rangle X_{p'} \Big) \wedge \Big(\bigwedge_{a} [a] (\bigvee_{p'.p \xrightarrow{a} p'} X_{p'}) \Big) \ .$$

In particular, it follows that

$$q \models_{max} [b] \bigvee_{p'.p \xrightarrow{b} p'} X_{p'} \ .$$

As we know that $q \xrightarrow{b} q_1$, we obtain that

$$q_1 \models_{max} \bigvee_{p'.p \xrightarrow{b} p'} X_{p'} \ .$$

Therefore there must exist a p_1 such that $q_1 \models_{max} X_{p_1}$ and $p \xrightarrow{b} p_1$.

We have therefore proven that

$$\exists p_1 .\, p \xrightarrow{b} p_1 \text{ and } (p_1, q_1) \in R \ ,$$

which was to be shown

We have now shown that R is a bisimulation, and therefore that

$$q \models_{max} X_p \text{ implies } p \sim q \ .$$

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 $\mathsf{Proc} = \{p_1, \ldots, p_n\}$.

Lemma 6.2 $([p_1]_\sim, \ldots, [p_n]_\sim) \sqsubseteq [\![D_K]\!]([p_1]_\sim, \ldots, [p_n]_\sim)$, where D_K is the declaration defined on page 145.

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

$$q \in \Big(\bigcap_{a,p'.p \xrightarrow{a} p'} \langle\cdot a\cdot\rangle [p']_\sim \Big) \cap \Big(\bigcap_a [\cdot a\cdot] \big(\bigcup_{p'.p \xrightarrow{a} p'} [p']_\sim \big) \Big) \ .$$

(Can you see why?) The proof can be divided into two parts, namely:

$$\begin{array}{ll} \text{a)} & q \in \bigcap_{a,p'.p \xrightarrow{a} p'} \langle\cdot a\cdot\rangle [p']_\sim \quad \text{and} \\ \text{b)} & q \in \bigcap_a [\cdot a\cdot] \big(\bigcup_{p'.p \xrightarrow{a} p'} [p']_\sim \big) \ . \end{array}$$

We proceed by proving these claims in turn.

a) We recall that $q \sim p$. Assume that $p \xrightarrow{a} p'$ for some action a and process p'. Then there is a q', where $q \xrightarrow{a} q'$ and $q' \sim p'$. We have therefore shown that, for all a and p', there is a q' such that

$$q \xrightarrow{a} q' \ \text{ and } \ q' \in [p']_\sim \ .$$

This means that, for each a and p' such that $p \xrightarrow{a} p'$, we have that

$$q \in \langle\cdot a\cdot\rangle [p']_\sim \ .$$

We may therefore conclude that

$$q \in \bigcap_{a,p'.p \xrightarrow{a} p'} \langle\cdot a\cdot\rangle [p']_\sim \ ,$$

which was to be shown.

b) Let $a \in \mathsf{Act}$ 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

$$q \in \bigcap_a [\cdot a\cdot] \big(\bigcup_{p'.p \xrightarrow{a} p'} [p']_\sim \big) \ .$$

Statements a) and b) above yield

$$([p_1]_\sim, \ldots, [p_n]_\sim) \sqsubseteq [\![D_K]\!]([p_1]_\sim, \ldots, [p_n]_\sim) \ ,$$

which was to be shown. □

Theorem 6.4 can now be expressed as the following lemma, whose proof completes the argument for that result.

Lemma 6.3 For each $p \in \mathsf{Proc}$, we have that $[\![X_p]\!] = [p]_\sim$.

Proof: Lemma 6.2 yields that

$$([p_1]_\sim, \ldots, [p_n]_\sim) \sqsubseteq ([\![X_{p_1}]\!], \ldots, [\![X_{p_n}]\!]) \ ,$$

which means that

$$[p]_\sim \subseteq [\![X_p]\!]$$

for each $p \in \mathsf{Proc}$. (Why?) Furthermore Lemma 6.1 gives that $[\![X_p]\!] \subseteq [p]_\sim$ for every $p \in \mathsf{Proc}$, which proves the statement of the lemma. □

Exercise 6.13 *What are the characteristic formulae for the processes p and q in Figure 6.1?* ⧫

Exercise 6.14 *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 $Pos(F)$, namely

$$Pos(F) \stackrel{\min}{=} F \vee \langle \mathsf{Act} \rangle Pos(F) \ .$$

Therefore, all that we need to 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

$$p = p_0 \stackrel{\tau}{\rightarrow} p_1 \stackrel{\tau}{\rightarrow} p_2 \stackrel{\tau}{\rightarrow} p_3 \stackrel{\tau}{\rightarrow} \cdots$$

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 τ-labelled transition leading to a state p_1 which has a livelock now. This immediately suggests the following recursive specification of the property LivelockNow:

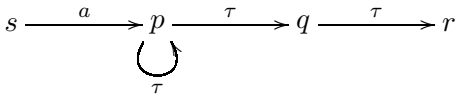
$$\text{LivelockNow} = \langle\tau\rangle\text{LivelockNow} \ .$$

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

$$\text{LivelockNow} \stackrel{\max}{=} \langle\tau\rangle\text{LivelockNow} \ .$$

Exercise 6.15 *What would be the least solution of the above equation?* ♦

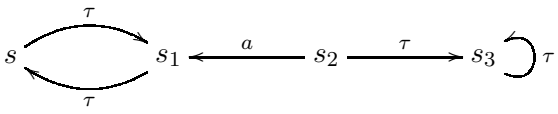
Exercise 6.16 (Mandatory) *Consider the labelled transition system below.*



Use the iterative algorithm for computing the set of states in that labelled transition system that satisfies the formula LivelockNow defined above. ♦

Exercise 6.17 *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.



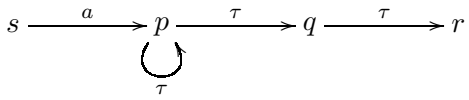
Use the iterative algorithm for computing the set of states in that labelled transition system that satisfies the formula LivelockNow defined above. ♦

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:

$$\begin{array}{rcl} Pos(\text{LivelockNow}) & \stackrel{\min}{=} & \text{LivelockNow} \vee \langle \mathsf{Act} \rangle Pos(\text{LivelockNow}) \\ \text{LivelockNow} & \stackrel{\max}{=} & \langle \tau \rangle \text{LivelockNow} \ . \end{array}$$

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 $Pos(\text{LivelockNow})$ above. How can we, for instance, compute the set of processes in the labelled transition system



that satisfy the formula $Pos(\text{LivelockNow})$? 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 $Pos(\text{LivelockNow})$ once we have in our hands the collection of processes satisfying the formula LivelockNow. As you saw in Exercise 6.16, 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 $Pos(\text{LivelockNow})$ as the *least* solution of the set equation

$$S = \{p\} \cup \langle \cdot \mathsf{Act} \cdot \rangle S \ , \tag{6.16}$$

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

$$S^{(1)} = \{p\} \cup \langle \cdot \mathsf{Act} \cdot \rangle \emptyset = \{p\} \ .$$

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

$$S^{(2)} = \{p\} \cup \langle \cdot \mathsf{Act} \cdot \rangle \{p\} = \{s, p\} \ .$$

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

$$\{s, p\} = \{p\} \cup \langle \cdot \mathsf{Act} \cdot \rangle \{s, p\} \ .$$

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 $Pos(\text{LivelockNow})$. 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 τ-labelled transitions is possible. (In case of p, this sequence is empty since p can embark in a τ-loop immediately.)

Note that we could find the set of states satisfying $Pos(\text{LivelockNow})$ by first computing $[\![\text{LivelockNow}]\!]$, and then using this set to compute

$$[\![Pos(\text{LivelockNow})]\!] \ ,$$

because the specification of the formula LivelockNow was independent of that $Pos(\text{LivelockNow})$. 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

$$\langle\, (D_1, \mathcal{X}_1, m_1), (D_2, \mathcal{X}_2, m_2), \ldots, (D_n, \mathcal{X}_n, m_n) \,\rangle,$$

where the $\mathcal{X}_i$s are pairwise disjoint, finite sets of variables, and, for each $1 \leq i \leq n$,

- D_i is a declaration mapping the variables in the set $\mathcal{X}_i$ to formulae in HML with recursion that may use variables in the set $\bigcup_{1 \leq j \leq i} \mathcal{X}_j$,
- $m_i = \max$ or $m_i = \min$, and
- $m_i \neq m_{i+1}$.

We refer to $(D_i, \mathcal{X}_i, m_i)$ as the ith *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 $Pos(\text{LivelockNow})$ is given in terms of a 2-nested mutually recursive equational system. In fact, take $\mathcal{X}_1 = \{\text{LivelockNow}\}$ and $\mathcal{X}_2 = \{Pos(\text{LivelockNow})\}$. You can now easily check that the constraints in the above definition are met. On the other hand, the mixed equational system

$$\begin{aligned} X &\stackrel{\max}{=} \langle a \rangle Y \\ Y &\stackrel{\min}{=} \langle b \rangle X \end{aligned}$$

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 μ-calculus* (Kozen, 1983), 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 employing 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 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

$$[\![F]\!] = \mathcal{O}_F([\![Y_1]\!], \ldots, [\![Y_k]\!]) \ , \tag{6.17}$$

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 6.18 *Consider the labelled transition system in Exercise 6.17. Use equation (6.17) to compute the set of states satisfying the formula*

$$F = \langle \mathsf{Act} \rangle Pos(\textit{LivelockNow}) \ .$$

♦

Exercise 6.19 *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 Section 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 μ-calculus (Kozen, 1983), 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 *computation 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, 1977) is probably the most studied logic of this type, in

particular with the connection to the automata-theoretic approach to model checking (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 to co-NP. In fact it was proved by Jurdziński (Jurdziński, 1998) that the problem is even in UP $\cap$ 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) μ-calculus is EXPTIME-complete (Kupferman et al., 2000).

If we consider the classes of sequential systems with infinitely many reachable states like, for example, pushdown automata, the model checking problem for the μ-calculus remains decidable. More precisely, it is EXPTIME-complete, as shown in (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 (Caucal, 1996; Muller and Schupp, 1985). The EXPTIME-hardness of model checking μ-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 processes in 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 the references (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.

while true do
begin
    'noncritical section';
    $b_i :=$ **true**;
    $k := j$;
    while (b_j **and** $k = j$) **do skip**;
    'critical section';
    $b_i :=$ **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 Section 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 $\mathrm{B}_{1\mathrm{t}}$ an $\mathrm{B}_{1\mathrm{f}}$. 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:

$$\begin{aligned} \mathrm{B_{1f}} &\stackrel{\text{def}}{=} \overline{\mathrm{b1rf}}.\mathrm{B_{1f}} + \mathrm{b1wf}.\mathrm{B_{1f}} + \mathrm{b1wt}.\mathrm{B_{1t}} \\ \mathrm{B_{1t}} &\stackrel{\text{def}}{=} \overline{\mathrm{b1rt}}.\mathrm{B_{1t}} + \mathrm{b1wf}.\mathrm{B_{1f}} + \mathrm{b1wt}.\mathrm{B_{1t}} \ . \end{aligned}$$

Intuitively, when in state $\mathrm{B_{1t}}$, 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:

$$\begin{aligned} \mathrm{B_{2f}} &\stackrel{\text{def}}{=} \overline{\mathrm{b2rf}}.\mathrm{B_{2f}} + \mathrm{b2wf}.\mathrm{B_{2f}} + \mathrm{b2wt}.\mathrm{B_{2t}} \\ \mathrm{B_{2t}} &\stackrel{\text{def}}{=} \overline{\mathrm{b2rt}}.\mathrm{B_{2t}} + \mathrm{b2wf}.\mathrm{B_{2f}} + \mathrm{b2wt}.\mathrm{B_{2t}} \ . \end{aligned}$$

The CCS representation of the behaviour of the variable k is as follows:

$$\begin{aligned} \mathrm{K_1} &\stackrel{\text{def}}{=} \overline{\mathrm{kr1}}.\mathrm{K_1} + \mathrm{kw1}.\mathrm{K_1} + \mathrm{kw2}.\mathrm{K_2} \\ \mathrm{K_2} &\stackrel{\text{def}}{=} \overline{\mathrm{kr2}}.\mathrm{K_2} + \mathrm{kw1}.\mathrm{K_1} + \mathrm{kw2}.\mathrm{K_2} \ . \end{aligned}$$

Again, the process representing the variable k has two states, denoted by the constants $\mathrm{K_1}$ and $\mathrm{K_2}$ 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:

$$\mathrm{P}_1 \stackrel{\text{def}}{=} \overline{\mathrm{b1wt}}.\overline{\mathrm{kw2}}.\mathrm{P}_{11} \ .$$

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:

while (b_j **and** $k = j$) **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

$$b_j \textbf{ and } k = j.$$

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

$$\mathrm{P}_{11} \stackrel{\text{def}}{=} \mathrm{b2rf}.\mathrm{P}_{12} + \mathrm{b2rt}.(\mathrm{kr2}.\mathrm{P}_{11} + \mathrm{kr1}.\mathrm{P}_{12}) \ .$$

Exercise 7.2 *Would it have been a good idea to define P_{11} thus:*

$$P_{11} \stackrel{def}{=} b2rf.P_{12} + b2rt.kr2.P_{11} + b2rt.kr1.P_{12} \ ?$$

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:

$$\mathrm{P}_{12} \stackrel{\text{def}}{=} \mathrm{enter1}.\mathrm{exit1}.\overline{\mathrm{b1wf}}.\mathrm{P}_1 \ .$$

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:

$$\begin{aligned}
\mathrm{P}_2 &\stackrel{\text{def}}{=} \overline{\mathrm{b2wt}}.\overline{\mathrm{kw1}}.\mathrm{P}_{21} \\
\mathrm{P}_{21} &\stackrel{\text{def}}{=} \mathrm{b1rf}.\mathrm{P}_{22} + \mathrm{b1rt}.(\mathrm{kr1}.\mathrm{P}_{21} + \mathrm{kr2}.\mathrm{P}_{22}) \\
\mathrm{P}_{22} &\stackrel{\text{def}}{=} \mathrm{enter2}.\mathrm{exit2}.\overline{\mathrm{b2wf}}.\mathrm{P}_2 \ .
\end{aligned}$$

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

$$\mathrm{Peterson} \stackrel{\text{def}}{=} (\mathrm{P}_1 \mid \mathrm{P}_2 \mid \mathrm{B}_{1\mathrm{f}} \mid \mathrm{B}_{2\mathrm{f}} \mid \mathrm{K}_1) \setminus L \ .$$

Exercise 7.3 (Mandatory!) *Give a CCS process that describes the behaviour of Hyman's 'mutual exclusion' algorithm. Hyman's algorithm was proposed in the reference (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 := true;
    while k ≠ j do begin
                      while b_j do skip;
                      k:=i
                   end;
    'critical section';
    b_i := 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 $Inv(F)$, where F is the 'desirable property' that we wish to hold at all points in the execution of the process. Recall that $Inv(F)$ is nothing but a shorthand for the recursively defined formula

$$Inv(F) \stackrel{\max}{=} F \wedge [\mathsf{Act}]Inv(F) \ .$$

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 exit_i. So our formula F can be taken to be

$$F \stackrel{\text{def}}{=} [\mathrm{exit}_1]ff \vee [\mathrm{exit}_2]ff \ .$$

The formula $Inv(F)$ 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 $\stackrel{\alpha}{\Rightarrow}$ for any action α including τ. So a formula like $[\mathrm{exit}_1]ff$ is satisfied by all processes that do not afford an $\stackrel{\mathrm{exit}_1}{\Rightarrow}$-labelled transition—that is, by those processes that cannot perform action exit_1 no matter how many internal steps they do before.

Exercise 7.4 *Consider the formula $Inv(G)$, where G is*

$$([enter_1][enter_2]ff) \wedge ([enter_2][enter_1]ff) \ .$$

Would such a formula be a good specification for our correctness criterion? What if we took G to be the formula

$$(\langle enter_1\rangle[enter_2]ff) \wedge (\langle enter_2\rangle[enter_1]ff) \ ?$$

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 $Inv(F)$ 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

$$S \mapsto [\![F]\!] \cap [\cdot\mathsf{Act}\cdot]S \ ,$$

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 the 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 $Inv(F)$ 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 $Inv(G)$ 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 $Inv(F)$ 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 (or equivalence checking) 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 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, there is no single notion of behavioural equivalence that fits all purposes. We have already met notions of equivalence like trace equivalence (Section 3.2), strong bisimilarity (Section 3.3) and weak bisimilarity (Section 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 to rest 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

$$\mathrm{MutexSpec} \stackrel{\mathrm{def}}{=} \mathrm{enter}_1.\mathrm{exit}_1.\mathrm{MutexSpec} + \mathrm{enter}_2.\mathrm{exit}_2.\mathrm{MutexSpec} \ . \qquad (7.1)$$

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 trace equivalence or strong bisimilarity, as presented in Sections 3.2 and 3.3, 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

$$\text{Peterson} \stackrel{\tau}{\Rightarrow} (\mathrm{P}_{12} \mid \mathrm{P}_{21} \mid \mathrm{B}_{1\mathrm{t}} \mid \mathrm{B}_{2\mathrm{t}} \mid \mathrm{K}_1) \setminus L \ ,$$

and that the state that is the target of that transition affords an enter_1-labelled transition, but cannot perform any 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 τ-transitions will bring process Peterson into state $(P_{12} \mid P_{21} \mid B_{1t} \mid B_{2t} \mid K_1) \setminus L$? (You will need five τ-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

$$P = P_0 \stackrel{a_1}{\Rightarrow} P_1 \stackrel{a_2}{\Rightarrow} \cdots \stackrel{a_k}{\Rightarrow} P_k \ ,$$

for some $P_1, \ldots, P_k$. Moreover, each process affords the weak trace ε.

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 $\mathcal{R}$ over the set of states of an LTS is a *weak simulation* iff whenever $s_1 \mathcal{R} s_2$ and α is an action (including τ):

- if $s_1 \stackrel{\alpha}{\rightarrow} s_1'$, then there is a transition $s_2 \stackrel{\alpha}{\Rightarrow} s_2'$ such that $s_1' \mathcal{R} s_2'$.

We say that s' *weakly simulates* s iff there is a weak simulation $\mathcal{R}$ with $s \mathcal{R} s'$. ♦

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 α, 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 enter_1, and the next observable action might be enter_2—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,

$$\begin{aligned}
\text{MutexTest} &\stackrel{\text{def}}{=} \overline{\text{enter}_1}.\text{MutexTest}_1 + \overline{\text{enter}_2}.\text{MutexTest}_2 \\
\text{MutexTest}_1 &\stackrel{\text{def}}{=} \overline{\text{exit}_1}.\text{MutexTest} + \overline{\text{enter}_2}.\overline{\text{bad}}.\mathbf{0} \\
\text{MutexTest}_2 &\stackrel{\text{def}}{=} \overline{\text{exit}_2}.\text{MutexTest} + \overline{\text{enter}_1}.\overline{\text{bad}}.\mathbf{0} \ ,
\end{aligned}$$

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

$$(\text{Peterson} \mid \text{MutexTest}) \setminus \{\text{enter}_1, \text{enter}_2, \text{exit}_1, \text{exit}_2\}$$

can initially perform the action $\overline{\text{bad}}$. 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 $(P \mid \text{MutexTest}) \setminus L' \stackrel{\overline{\text{bad}}}{\Rightarrow}$ iff either $P \stackrel{\sigma}{\Rightarrow} P' \stackrel{\text{enter}_1}{\Rightarrow} P'' \stackrel{\text{enter}_2}{\Rightarrow} P'''$ or $P \stackrel{\sigma}{\Rightarrow} P' \stackrel{\text{enter}_2}{\Rightarrow} P'' \stackrel{\text{enter}_1}{\Rightarrow} P'''$, for some P', P'', P''' and some sequence of actions σ 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

$$P \stackrel{\sigma}{\Rightarrow} P' \stackrel{\text{enter}_1}{\Rightarrow} P'' \stackrel{\text{enter}_2}{\Rightarrow} P''' \ ,$$

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 $(P \mid \text{MutexTest}) \setminus L' \stackrel{\overline{\text{bad}}}{\Rightarrow}$. To see this, note that, using induction on the length of the sequence σ, it is not hard to prove that

$$(P \mid \text{MutexTest}) \setminus L' \stackrel{\tau}{\Rightarrow} (P' \mid \text{MutexTest}) \setminus L' \ .$$

Since $P' \stackrel{\text{enter}_1}{\Rightarrow} P'' \stackrel{\text{enter}_2}{\Rightarrow} P'''$, we have that

$$(P' \mid \text{MutexTest}) \setminus L' \stackrel{\tau}{\Rightarrow} (P'' \mid \text{MutexTest}_1) \setminus L' \stackrel{\tau}{\Rightarrow} (P''' \mid \overline{\text{bad}}.\mathbf{0}) \setminus L' \stackrel{\overline{\text{bad}}}{\rightarrow} \ .$$

Combining the above sequences of transitions, we may conclude that

$$(P \mid \text{MutexTest}) \setminus L' \stackrel{\overline{\text{bad}}}{\Rightarrow} \ ,$$

which was to be shown.

Conversely, assume that $(P \mid \text{MutexTest}) \setminus L' \stackrel{\overline{\text{bad}}}{\Rightarrow}$. Since $\overline{\text{bad}}.\mathbf{0}$ is the only state of process MutexTest that can perform a $\overline{\text{bad}}$-action, this means that, for some P''',

$$(P \mid \text{MutexTest}) \setminus L' \stackrel{\tau}{\Rightarrow} (P''' \mid \overline{\text{bad}}.\mathbf{0}) \setminus L' \stackrel{\overline{\text{bad}}}{\rightarrow} \ .$$

Because of the way MutexTest is constructed, this must be because, for some P' and P'' such that either $P' \stackrel{\text{enter}_1}{\Rightarrow} P'' \stackrel{\text{enter}_2}{\Rightarrow} P'''$ or $P' \stackrel{\text{enter}_2}{\Rightarrow} P'' \stackrel{\text{enter}_1}{\Rightarrow} P'''$,

$$\begin{aligned}(P \mid \text{MutexTest}) \setminus L' &\stackrel{\tau}{\Rightarrow} (P' \mid \text{MutexTest}) \setminus L' \\ &\stackrel{\tau}{\Rightarrow} (P'' \mid \text{MutexTest}_i) \setminus L' \quad (i \in \{1,2\}) \\ &\stackrel{\tau}{\Rightarrow} (P''' \mid \overline{\text{bad}}.\mathbf{0}) \setminus L' \ .\end{aligned}$$

Using induction on the number of $\stackrel{\tau}{\rightarrow}$-steps in the weak transition

$$(P \mid \text{MutexTest}) \setminus L' \stackrel{\tau}{\Rightarrow} (P' \mid \text{MutexTest}) \setminus L' \ ,$$

you can now argue that $P \stackrel{\sigma}{\Rightarrow} P'$, for some sequence of actions σ in the regular language $(\text{enter}_1\text{exit}_1 + \text{enter}_2\text{exit}_2)^*$. This completes the proof. □

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 $Inv(G)$, where G is

$$([\text{enter}_1][\text{enter}_2]ff) \wedge ([\text{enter}_2][\text{enter}_1]ff) \ .$$

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 $Inv(G)$. 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 $\overline{\text{bad}}$ 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 $\mathsf{Act} \cup \{\overline{\mathsf{bad}}\}$, where bad is a distinguished channel name not occurring in Act. We use $\mathsf{root}(T)$ 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 $\overline{\mathsf{bad}}$-labelled transition.

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 :

$$T ::= \mathbf{0} \mid \alpha.T \mid T + T \mid X \ ,$$

where α can be any action in Act as well as the distinguished action $\overline{\mathsf{bad}}$, 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

$$X \stackrel{\text{def}}{=} a.(b.\mathbf{0} \mid X)$$

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

  $$(s \mid \mathsf{root}(T)) \setminus \mathcal{L} \stackrel{\overline{\mathsf{bad}}}{\not\Rightarrow} \ .$$

  (Recall that $\mathcal{L}$ stands for the collection of observable actions in CCS except for the action bad.) 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 $\mathcal{T}$ and every state s of $\mathcal{T}$,

  $$s \models F \text{ iff } s \text{ passes the test } T \ .$$

- A collection of formulae in HML with recursion is testable iff each of the formulae in it is.

♦

Example 7.1 The formula $[a]ff$ is satisfied by those processes that do not afford an $\stackrel{a}{\Rightarrow}$-transition. We therefore expect that a suitable test for such a property is $T \equiv \bar{a}.\overline{\mathrm{bad}}.\mathbf{0}$. Indeed, the reader will easily realize that $(s \mid T) \setminus \mathcal{L} \stackrel{\overline{\mathrm{bad}}}{\Rightarrow}$ iff $s \stackrel{a}{\not\Rightarrow}$, 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

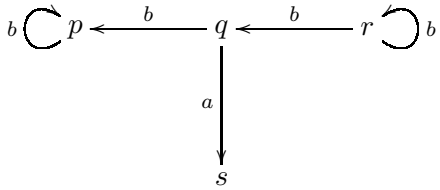
$$F \stackrel{\max}{=} [a]ff \wedge [b]F$$

is satisfied by those states which cannot perform any $\stackrel{a}{\Rightarrow}$-transition, no matter how they engage in a sequence of $\stackrel{b}{\Rightarrow}$-transitions. (Why?) A suitable test for such a property is

$$X \stackrel{\mathrm{def}}{=} \bar{a}.\overline{\mathrm{bad}}.\mathbf{0} + \bar{b}.X \ ,$$

and the recursively defined formula F is thus testable. ♦

Exercise 7.13 *Consider the following labelled transition system:*



Compute the set of states in this labelled transition system that satisfy the property

$$F \stackrel{\max}{=} [a]ff \wedge [b]F \ .$$

Which of the states in that labelled transition system passes the test

$$X \stackrel{def}{=} \bar{a}.\overline{bad}.\mathbf{0} + \bar{b}.X \ ?$$

Argue for your answers! ♦

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.

Proposition 7.3 [Two negative results]

1. For every action a in $\mathcal{L}$, the formula $\langle a\rangle tt$ is not testable.
2. Let a and b be two distinct actions in $\mathcal{L}$. Then the formula $[a]ff \vee [b]ff$ is not testable.

Proof: We prove each statement in turn.

- PROOF OF (1). Assume, towards a contradiction, that a test T tests for the formula $\langle a\rangle tt$. Since T tests for $\langle a\rangle tt$ and $\mathbf{0} \not\models \langle a\rangle tt$, we have that

$$(\mathbf{0} \mid \mathsf{root}(T)) \setminus \mathcal{L} \stackrel{\overline{\mathrm{bad}}}{\Rightarrow} .$$

  Consider now the term $P = a.\mathbf{0} + \tau.\mathbf{0}$. As $P \stackrel{a}{\rightarrow} \mathbf{0}$, the process P satisfies the formula $\langle a\rangle tt$. However, P fails the test T because

$$(P \mid \mathsf{root}(T)) \setminus \mathcal{L} \stackrel{\tau}{\rightarrow} (\mathbf{0} \mid \mathsf{root}(T)) \setminus \mathcal{L} \stackrel{\overline{\mathrm{bad}}}{\Rightarrow} .$$

  This contradicts our assumption that T tests for $\langle a\rangle tt$.

- PROOF OF (2). Assume, towards a contradiction, that a test T tests for the formula $[a]ff \vee [b]ff$, with $a \neq b$. Since the state $a.\mathbf{0} + b.\mathbf{0}$ does not satisfy the formula $[a]ff \vee [b]ff$, it follows that

$$((a.\mathbf{0} + b.\mathbf{0}) \mid \mathsf{root}(T)) \setminus \mathcal{L} \stackrel{\overline{\mathrm{bad}}}{\Rightarrow} . \tag{7.2}$$

  We now proceed to show that this implies that either the state $a.\mathbf{0}$ fails the test T or $b.\mathbf{0}$ does. This we do by examining the possible forms transition (7.2) may take.

  - CASE: $((a.\mathbf{0} + b.\mathbf{0}) \mid \mathsf{root}(T)) \setminus \mathcal{L} \stackrel{\overline{\mathrm{bad}}}{\Rightarrow}$ because $\mathsf{root}(T) \stackrel{\overline{\mathrm{bad}}}{\Rightarrow}$. In this case, every state of an LTS fails the test T, and we are done.
  - CASE: $((a.\mathbf{0} + b.\mathbf{0}) \mid \mathsf{root}(T)) \setminus \mathcal{L} \stackrel{\tau}{\Rightarrow} (\mathbf{0} \mid t) \setminus \mathcal{L} \stackrel{\overline{\mathrm{bad}}}{\rightarrow}$, because $\mathsf{root}(T) \stackrel{\bar{a}}{\Rightarrow} t$ for some state t of T. In this case, we may infer that

$$(a.\mathbf{0} \mid \mathsf{root}(T)) \setminus \mathcal{L} \stackrel{\tau}{\Rightarrow} (\mathbf{0} \mid t) \setminus \mathcal{L} \stackrel{\overline{\mathrm{bad}}}{\rightarrow}$$

    and thus that $a.\mathbf{0}$ fails the test T.
  - CASE: $((a.\mathbf{0} + b.\mathbf{0}) \mid \mathsf{root}(T)) \setminus \mathcal{L} \stackrel{\tau}{\Rightarrow} (\mathbf{0} \mid t) \setminus \mathcal{L} \stackrel{\overline{\mathrm{bad}}}{\rightarrow}$, because $\mathsf{root}(T) \stackrel{\bar{b}}{\Rightarrow} t$ for some state t of T. In this case, reasoning as above, it is easy to see that $b.\mathbf{0}$ fails the test T.

Hence, as previously claimed, either $a.\mathbf{0}$ fails the test T or $b.\mathbf{0}$ does. Since both $a.\mathbf{0}$ and $b.\mathbf{0}$ satisfy the formula $[a]ff \vee [b]ff$, this contradicts our assumption that T tests for it.

The proof is now complete. □

The collection of formulae in *safety HML* is the set of formulae in HML with recursion that do *not* contain occurrences of $\vee$, $\langle\alpha\rangle$ and variables defined using least fixed point recursion equations. For instance, the formula $\langle a\rangle X$ is not a legal formula in safety HML if X is defined thus:

$$X \stackrel{\min}{=} \langle b\rangle tt \vee \langle a\rangle X \ .$$

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 Ingolfsdottir, 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 Ingolfsdottir, 1999) for more details, further developments and references to the literature.