An Introduction to Milner's CCS

Luca Aceto Kim G. Larsen
Anna Ingólfsdóttir
BRICS, Department of Computer Science
Aalborg University
9220 Aalborg Ø, Denmark

March 10, 2005

These notes are intended to support the course on Semantics and Verification, and may be read independently or in conjuction with Milner's textbook [12]. They are under constant revision, and their most recent version is available at the URL

http://www.cs.auc.dk/∼luca/SV/intro2ccs.pdf.

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

`luca@cs.aau.dk` and `srba@cs.aau.dk`

with subject line "CCS Notes".

Contents

1 Introduction

The aim of this collection of notes is to offer some support for the course on *Semantics and Verification* by introducing three of the basic notions that we shall use to describe, specify and analyze reactive systems, namely

- Milner's Calculus of Communicating Systems (CCS) [12],
- the model of Labelled Transition Systems (LTSs) [9] and
- Hennessy-Milner Logic (HML) [7] and its extension with recursive definitions of formulae [11].

The *Semantics and Verification* course presents a general theory of reactive systems and its applications. Our aims in this course are 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 a formulae in HML and

3. to manipulate these descriptions, possibly (semi-)automatically in order to obtain an analysis of the behaviour of the model of the system under consideration.

At the end of the course, the students 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 Concurrency Workbench and UPPAAL.

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 course need to have a formal syntax and semantics. 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 these notes, and from the lectures that they are supposed to complement, 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".

2 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 giving its collection of legal inputs, and, for each legal input, its expected output. In an imperative setting, an abstract view of a computing system may 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 state—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 partial 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.

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 [6] 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, unlike with "standard" computing systems, as again nicely exemplified by the skeleton of a control program given above, non-termination is a *desirable* feature of a reactive system. 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 these notes 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/LTSs,
- 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 analysis 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 course are becoming widely accepted in the academic and industrial communities that develop reactive systems.

3 Process Algebras

The first ingredient in the approach to the theory of reactive systems presented in this course 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 [1]. 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, then 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. 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 CSP [8], a highly influential proposal for a programming language for parallel programs, and Milner's CCS [12], the paradigmatic process algebra.

4 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 Sect. 4.1. We then proceed to put our developments on a more formal footing in Sect. 4.2.

4.1 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

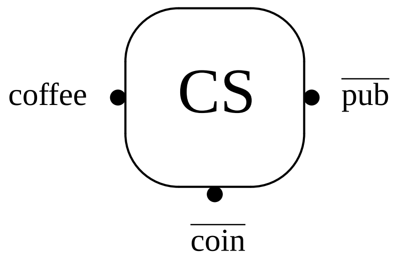


Table 1: The interface for process CS

information. For example, the drawing in Table 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 Table 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 process and its behaviour.

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

The most basic process constructor in CCS is *action prefixing*. Two example processes built using **0** and action prefixing are

a match: strike.**0** and

a complex match: take.strike.**0**.

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

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

The idea is that a label, like strike or $\overline{\text{pub}}$, 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:

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

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:

$$\text{Clock} \overset{\text{def}}{=} \text{tick.Clock} \quad .$$

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

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

for each positive integer n.

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

$$\text{CM} \overset{\text{def}}{=} \text{coin.}\overline{\text{coffee}}\text{.CM} \quad . \tag{1}$$

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

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 denoted "$+$". For example, a vending machine offering either tea or coffee may be described thus:

$$\mathrm{CTM} \overset{\mathrm{def}}{=} \mathrm{coin}.(\overline{\mathrm{coffee}}.\mathrm{CTM} + \overline{\mathrm{tea}}.\mathrm{CTM}) \ . \tag{2}$$

The idea here is that, after having input a coin, 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 4.1 *Give a CCS process that describes a coffee machine that may behave like that given by (1), but may also steal the money it receives and fail at any time.*

Exercise 4.2 *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".

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

$$\mathrm{CS} \overset{\mathrm{def}}{=} \overline{\mathrm{pub}}.\overline{\mathrm{coin}}.\mathrm{coffee}.\mathrm{CS} \ . \tag{3}$$

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

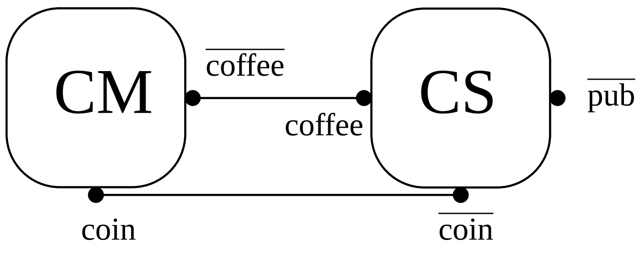


Table 2: The interface for process CM | CS

with each other, CCS offers the *parallel composition operation* |. For example, the CCS expression CM | CS describes a system consisting of two processes—the coffee machine CM and the computer scientist CS—that run in parallel one with the other. These two processes may communicate via the communication ports they share and use in complementary fashion, namely coffee and coin. By complementary, we mean that one of the processes uses the port for input and the other for output. Potential communications are represented in Table 2 by the solid lines linking complementary ports. The port pub is instead used by the computer scientist to communicate with its research environment, or, more prosaically, with other processes that may be present in its environment and that are willing to input along that port. One important thing to note is that the link between complementary ports in Table 2 denotes that it is *possible* for the computer scientist and the coffee machine to communicate in the parallel composition CM | CS. However, we do *not* require that they must communicate with one another. Both the computer scientist and the coffee machine could use their complementary ports to communicate with other reactive systems in their environment. For example, another computer scientist CS′ can use the coffee machine CM, and, in so doing, make sure that he can produce publications to beef up his curriculum vitae, and thus be a worthy competitor for CS in the next competition for a tenured position. (See Table 3.) Alternatively, the computer scientist may have access to another coffee machine in its environment, as pictured in Table 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 and
- may communicate via complementary ports.

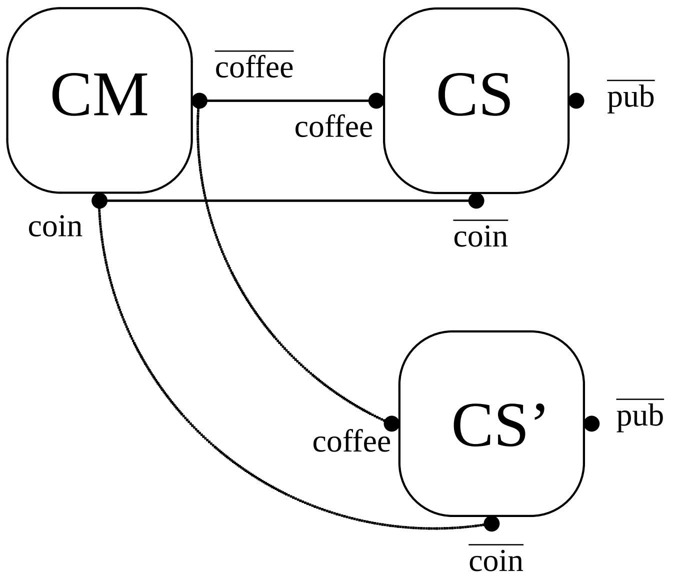


Table 3: The interface for process CM | CS | CS′

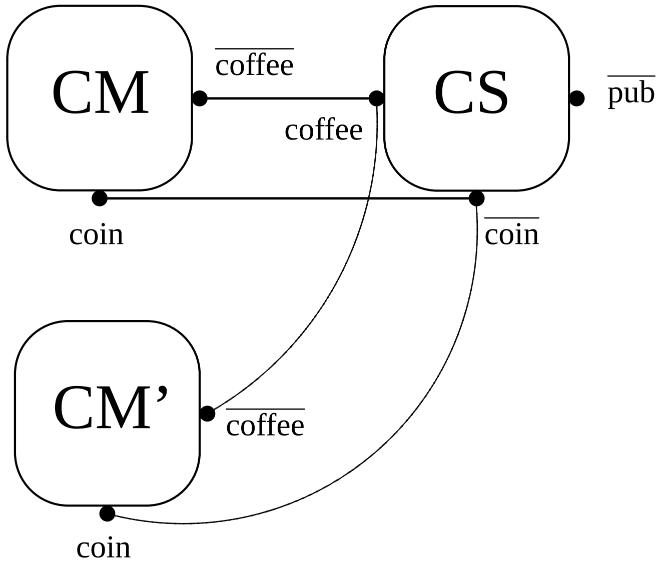


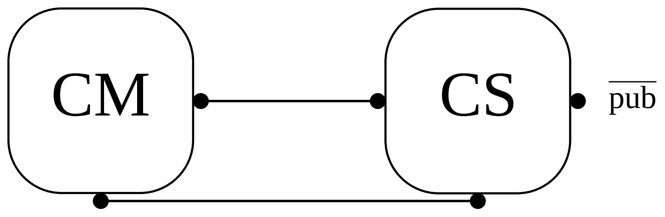
Table 4: The interface for process CM | CS | CM$'$

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

$$\text{SmUni} \stackrel{\text{def}}{=} (\text{CM} \mid \text{CS}) \setminus \text{coin} \setminus \text{coffee} \quad . \tag{4}$$

As pictured in Table 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:

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



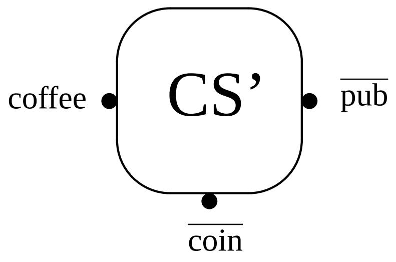


Table 5: The interface for processes SmUni and SmUni | CS′

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 (1). For instance, we may define the processes

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

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 inputs a coin, 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 Sect. 4.2, 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 4.2. 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 LTSs will be presented in Sect. 4.2 using the tools of Structural Operational Semantics.

4.1.1 The Behaviour of Processes

The key idea underlying the semantics of CCS is that a process passes through states during an execution; processes change their state by performing actions. For instance, the process CS defined in (3) can perform action $\overline{\text{pub}}$ and evolve into a process whose behaviour is described by the CCS expression

$$\text{CS}_1 \stackrel{\text{def}}{=} \overline{\text{coin}}.\text{coffee.CS}$$

in doing so. Process CS_1 can then output a coin, thereby evolving into a process whose behaviour is described by the CCS expression

$$\text{CS}_2 \stackrel{\text{def}}{=} \text{coffee.CS} \quad .$$

Finally, this process can input coffee, 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

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

which says that CS can perform action $\overline{\text{pub}}$, and become CS_1 in doing so. The operational behaviour of our computer scientist CS is therefore completely described by the LTS

$$\text{CS} \stackrel{\overline{\text{pub}}}{\rightarrow} \text{CS}_1 \stackrel{\overline{\text{coin}}}{\rightarrow} \text{CS}_2 \stackrel{\text{coffee}}{\rightarrow} \text{CS} \quad .$$

In much the same way, we can make explicit the set of states of the coffee machine described in (1) by rewriting that equation thus:

$$\begin{array}{rcl} \text{CM} & \stackrel{\text{def}}{=} & \text{coin.CM}_1 \\ \text{CM}_1 & \stackrel{\text{def}}{=} & \overline{\text{coffee}}.\text{CM} \quad . \end{array}$$

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

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

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

$$\text{CM} \stackrel{\text{coin}}{\rightarrow} \text{CM}_1 \quad .$$

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

$$\text{CM} \mid \text{CS}_1 \xrightarrow{?} \text{CM}_1 \mid \text{CS}_2 \ \ .$$

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

$$\text{CM} \mid \text{CS}_1 \xrightarrow{\tau} \text{CM}_1 \mid \text{CS}_2 \ \ .$$

In this way, the behaviour of the process SmUni defined by (4) can be described by the following LTS:

$$\begin{tikzcd}
\text{SmUni} \arrow[d, "\overline{\text{pub}}"] \\
(\text{CM} \mid \text{CS}_1) \setminus \text{coin} \setminus \text{coffee} \arrow[d, "\tau"] \\
(\text{CM}_1 \mid \text{CS}_2) \setminus \text{coin} \setminus \text{coffee} \arrow[d, "\tau"] \\
(\text{CM} \mid \text{CS}) \setminus \text{coin} \setminus \text{coffee} \arrow[uu, bend right=60, "\overline{\text{pub}}"']
\end{tikzcd}$$

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:

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

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 course 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.

4.2 CCS, Formally

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

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.

Definition 4.1 [Labelled Transition Systems] A *labelled transition system (LTS)* is a triple $(\mathsf{Proc}, \mathsf{Act}, \{\xrightarrow{a} \mid a \in \mathsf{Act}\})$, where:

- Proc is a set of *states*, ranged over by s;
- Act is a set of *actions*, ranged over by a;
- $\xrightarrow{a} \subseteq \mathsf{Proc} \times \mathsf{Proc}$ is a *transition relation*, for every $a \in \mathsf{Act}$. As usual, we shall use the more suggestive notation $s \xrightarrow{a} s'$ in lieu of $(s, s') \in \xrightarrow{a}$, and write $s \overset{a}{\nrightarrow}$ (read "s refuses a") iff $s \xrightarrow{a} s'$ for no state s'.

For example, the LTS for the process SmUni defined by (4) is formally specified thus:

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

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 [15]. (The history of the development of the ideas that led to SOS is recounted by Plotkin himself in [16].) 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 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.

The first 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 $\mathcal{L}$, 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 4.2 The collection $\mathcal{P}$ of *CCS expressions* is given by the following grammar:

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

where

- K is a process name in $\mathcal{K}$;
- α is an action in Act;

- I is an index set;
- f : Act $\rightarrow$ 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.

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 is given by a defining equation

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

As it was already made clear by the previous informal discussion, the constant K may appear in P.

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:

$$\begin{array}{rcll} \text{Counter}_0 & \stackrel{\text{def}}{=} & \text{up.Counter}_1 & (5) \\ \text{Counter}_n & \stackrel{\text{def}}{=} & \text{up.Counter}_{n+1} + \text{down.Counter}_{n-1} \quad (n > 0) \ . & (6) \end{array}$$

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 formally capture our understanding of the semantics of the language CCS, we therefore introduce the collection of SOS rules in Table 6. A transition $P \xrightarrow{\alpha} Q$ holds for CCS expressions P, Q if, and only if, it can be proven using these rules.

Table 6: SOS Rules for CCS ($\alpha \in \mathsf{Act}$, $a \in \mathcal{L}$)

$$\text{[Def]}\ \frac{P \xrightarrow{\alpha} P'}{K \xrightarrow{\alpha} P'}\ K \overset{\text{def}}{=} P \qquad \text{[Act]}\ \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{[Sum]}\ \frac{P_j \xrightarrow{\alpha} P'_j}{\sum_{i\in I} P_i \xrightarrow{\alpha} P'_j}\ j \in I$$

$$\text{[LPar]}\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \text{[RPar]}\ \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \qquad \text{[Com]}\ \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\bar{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\text{[Rel]}\ \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \qquad \text{[Res]}\ \frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P' \setminus L}\ \alpha, \bar{\alpha} \notin L$$

A rule like

$$\frac{}{\alpha.P \xrightarrow{\alpha} P}$$

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

$$\overline{\text{pub}}.\overline{\text{coin}}.\text{coffee}.\text{CS} \xrightarrow{\overline{\text{pub}}} \overline{\text{coin}}.\text{coffee}.\text{CS} \tag{7}$$

is provable using the above rule for action prefixing.

On the other hand, a rule like

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

has a non-empty set of premises. This rule says that to establish that constant K affords the transition mentioned in the conclusion of the rule, we have to first prove that the body of the defining equation for K, namely the process P, affords the transition $P \xrightarrow{\alpha} P'$. Using this rule, pattern matching and transition (7), we can prove the transition

$$\text{CS} \xrightarrow{\overline{\text{pub}}} \overline{\text{coin}}.\text{coffee}.\text{CS}\ ,$$

which we had informally derived before.

The aforementioned rule for constants has a *side condition*, namely $K \overset{\text{def}}{=} P$, that describes a constraint that must be met in order for the rule to be applicable.

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

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

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

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

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

$$\text{C} \overset{\text{def}}{=} \text{up}.(\text{C} \mid \text{down}.\mathbf{0}) \ . \tag{8}$$

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

$$\text{C} \xrightarrow{\text{up}} \text{C} \mid \text{down}.\mathbf{0} \ . \tag{9}$$

What next? Observing that down.$\mathbf{0} \xrightarrow{\text{down}} \mathbf{0}$, we can infer that

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

Since it is reasonable to expect that the process 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 (9), we have that the transition

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

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 (5). Indeed, we shall prove later on that these two processes exhibit the same behaviour in a very strong sense.

Exercise 4.3 *Use the rules of the SOS semantics for CCS to derive the LTS for the process SmUni defined by (4).*

Exercise 4.4 *Draw (part of) the transition graph for the process name A whose behaviour is given by the defining equation*

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

The resulting transition graph should have infinitely many states. Can you think of a CCS term that generates a finite state automaton that should intuitively have the same behaviour of A?

Exercise 4.5 *Draw (part of) the transition graph for the process name A whose behaviour is given by the defining equation*

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

where we assume that the set of labels 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 automaton that could intuitively have the same behaviour of A?

Exercise 4.6

1. *Draw the transition graph for the process name Mutex$_1$ whose behaviour is given by the defining equation*

$$\begin{array}{rcl} Mutex_1 & \overset{def}{=} & (User \mid Sem) \setminus \{p, v\} \\ User & = & \bar{p}.enter.exit.\bar{v}.User \\ Sem & \overset{def}{=} & p.v.Sem \ . \end{array}$$

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

$$Mutex_2 \overset{def}{=} ((User|Sem)|User) \setminus \{p, v\}$$

   where User and Sem are defined as before.

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

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

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?

4.2.1 Value Passing CCS

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 [12, Sect. 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 input one datum 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:

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

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 input a non-negative integer n, 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

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

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

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

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 rule

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

To become familiar with these rules, you should apply them to the one-place buffer B, and derive its possible transitions.

In what follows, we shall restrict ourselves to CCS expressions that have no free occurrences of value variables—that is, to CCS expressions in which each occurrence of a value variable, say y, is within the scope of an input prefix of the form $a(y)$ or of a parameterized constant $\mathrm{A}(x_1, \ldots, x_n)$ with $x = x_i$ 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:

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

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

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

and

$$\frac{Q \xrightarrow{\alpha} Q'}{\textbf{if } \text{bexp } \textbf{then } P \textbf{ else } Q \xrightarrow{\alpha} Q'} \quad \text{bexp is false}$$

Exercise 4.7 *Consider a one place buffer defined by*

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

Use the Cell to define a two-place bag and a two-place FIFO queue. 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 4.8 *Consider the process B defined thus:*

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

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 4.9 (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 [12, Sect. 2.8].*

5 Behavioural Equivalence

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

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 (4) 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 (8) 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 5.1 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 relation 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, transitive relation is a *pre-order*.

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

Exercise 5.1 *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* $M_2 = \{(n, m) \mid n, m \in \mathbb{N}, \ n \mod 2 = m \mod 2\}$.

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 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:

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

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 these notes, 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 $P \mathrel{R} Q$ and $C[\,]$ is a program fragment with "a hole", then

$$C[P] \mathrel{R} C[Q] \ .$$

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 [5], where van Glabbeek presents the linear time-branching time spectrum—a lattice of known behavioural equivalences and preorders over LTSs, ordered by inclusion.) In our search for a reasonable notion of behavioral relation to support implementation verification, we shall limit ourselves to presenting a tiny sample of these.

So let's begin our search!

5.1 Trace Equivalence: A First Attempt

Labelled transition systems (LTSs) [9] 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 [15] and, following Milner's pioneering work on CCS [12], 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

$$P = P_0 \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_k} P_k \ ,$$

for some $P_1, \ldots, P_k$. We write $\mathit{Traces}(P)$ for the collection of traces of P. Since $\mathit{Traces}(P)$ 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 communications that can be performed with one, but not with the other. This means that, for all processes P and Q, we require that

$$\text{if } P \text{ and } Q \text{ are behaviourally equivalent, then } \mathit{Traces}(P) = \mathit{Traces}(Q) \quad . \tag{10}$$

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 as in (2), and compare it with the following one:

$$\mathrm{CTM}' \overset{\mathrm{def}}{=} \mathrm{coin}.\overline{\mathrm{coffee}}.\mathrm{CTM}' + \mathrm{coin}.\overline{\mathrm{tea}}.\mathrm{CTM}' \quad . \tag{11}$$

You should be able to convince yourselves that CTM and CTM′ afford the same traces. 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

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

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:

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

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 5.2 *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*

$$P = P_0 \stackrel{\alpha_1}{\rightarrow} P_1 \stackrel{\alpha_2}{\rightarrow} \cdots \stackrel{\alpha_k}{\rightarrow} P_k \not\rightarrow \ ,$$

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?*

5.2 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 5.2, 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 inputting a coin. Indeed, after accepting a coin the machine CTM always enters a state in which it is willing to output either coffee or 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 they should 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 inputting a coin are *not* trace equivalent.

The classic notion of strong bisimulation equivalence, introduced by David Park in [14], formalizes the informal requirements introduced above in a very elegant way.

Definition 5.2 [Strong Bisimulation] A binary relation $\mathcal{R}$ over the set of states of an LTS is a *bisimulation* iff whenever $s_1 \mathrel{\mathcal{R}} s_2$ and α is an action:

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

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 5.2.

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.

Consider, for instance, 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

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

So, by the second requirement in Definition 5.2, there must be a transition

$$\mathrm{CTM} \overset{\mathrm{coin}}{\rightarrow} P$$

for some process P such that $P\ \mathcal{R}\ \overline{\mathrm{tea}}.\mathrm{CTM}'$. A moment of thought should be enough to convince yourselves that the only process that CTM can reach by inputting a coin is $\overline{\mathrm{coffee}}.\mathrm{CTM} + \overline{\mathrm{tea}}.\mathrm{CTM}$, so we are requiring that

$$\overline{\mathrm{coffee}}.\mathrm{CTM} + \overline{\mathrm{tea}}.\mathrm{CTM}\ \mathcal{R}\ \overline{\mathrm{tea}}.\mathrm{CTM}' \ .$$

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

$$\overline{\mathrm{coffee}}.\mathrm{CTM} + \overline{\mathrm{tea}}.\mathrm{CTM} \overset{\overline{\mathrm{coffee}}}{\rightarrow} \mathrm{CTM} \ ,$$

but $\overline{\mathrm{tea}}.\mathrm{CTM}'$ cannot output coffee. Thus the first requirement in Definition 5.2 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 5.1 Consider the processes P and Q defined thus:

$$\begin{array}{rcl} P & \overset{\mathrm{def}}{=} & a.P_1 + b.P_2 \\ P_1 & \overset{\mathrm{def}}{=} & c.P \\ P_2 & \overset{\mathrm{def}}{=} & c.P \end{array}$$

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 5.3 *Consider the processes P and Q defined thus:*

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

and

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

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

Exercise 5.4 *Consider the processes*

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

Show that P and Q are not strongly bisimilar.

Before looking 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 these notes.

Theorem 5.1 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 $s_1 \xrightarrow{\alpha} s_1'$, then there is a transition $s_2 \xrightarrow{\alpha} s_2'$ such that $s_1' \sim s_2'$;
   - if $s_2 \xrightarrow{\alpha} s_2'$, then there is a transition $s_1 \xrightarrow{\alpha} s_1'$ such that $s_1' \sim s_2'$.

Proof: Consider an LTS $(\mathsf{Proc}, \mathsf{Act}, \{\xrightarrow{\alpha} \mid \alpha \in \mathsf{Act}\})$. 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 5.1.)

   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
$$\sim \quad = \quad \bigcup\{\mathcal{R} \mid \mathcal{R} \text{ is a bisimulation}\} \ .$$
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
$$(s_1, s_2) \in \bigcup\{\mathcal{R} \mid \mathcal{R} \text{ is a bisimulation}\} \text{ and } s_1 \overset{\alpha}{\rightarrow} s_1' \ , \qquad (12)$$
   then there is a state s_2' such that $s_2 \overset{\alpha}{\rightarrow} s_2'$ and
$$(s_1', s_2') \in \bigcup\{\mathcal{R} \mid \mathcal{R} \text{ is a bisimulation}\} \ .$$
   Assume, therefore, that (12) 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 $s_1 \overset{\alpha}{\rightarrow} s_1'$, we have that there is a state s_2' such that $s_2 \overset{\alpha}{\rightarrow} s_2'$ and $(s_1', s_2') \in \mathcal{R}$. Observe now that 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 $s_2 \overset{\alpha}{\rightarrow} s_2'$ and
$$(s_1', s_2') \in \{\mathcal{R} \mid \mathcal{R} \text{ is a bisimulation}\} \ ,$$
   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 $s_1 \overset{\alpha}{\rightarrow} s_1'$, then there is a transition $s_2 \overset{\alpha}{\rightarrow} s_2'$ such that $s_1' \sim s_2'$;
   - if $s_2 \overset{\alpha}{\rightarrow} s_2'$, then there is a transition $s_1 \overset{\alpha}{\rightarrow} s_1'$ 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 $s_1 \xrightarrow{\alpha} s_1'$, then there is a transition $s_2 \xrightarrow{\alpha} s_2'$ such that $s_1' \sim s_2'$;
- if $s_2 \xrightarrow{\alpha} s_2'$, then there is a transition $s_1 \xrightarrow{\alpha} s_1'$ such that $s_1' \sim s_2'$.

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 it to prove $s_1 \sim s_2$. Since $\mathcal{R}$ should be a bisimulation, each transition $s_1 \xrightarrow{\alpha} s_1'$ from s_1 should be matched by a transition $s_2 \xrightarrow{\alpha} s_2'$ from s_2, for some state s_2' such that $(s_1', s_2') \in \mathcal{R}$. 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 5.5 *Prove that the relations we have built in the proof of Theorem 5.1 are indeed bisimulations.*

Exercise 5.6 *Assume that the defining equation for the constant K is $K \overset{\mathit{def}}{=} P$. Show that $K \sim P$ holds.*

Exercise 5.7 *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 (10). [Hint: Use induction on the length of the trace $\alpha_1 \cdots \alpha_k$ ($k \geq 0$) to show that*

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

Exercise 5.8 *Show that the following relations are strong bisimulations:*

$$\begin{array}{l}\{(P \mid Q, Q \mid P) \mid \textit{where } P, Q \textit{ are CCS processes}\} \\ \{(P \mid \mathbf{0}, P) \mid \textit{where } P \textit{ is a CCS process}\} \\ \{((P \mid Q) \mid R, P \mid (Q \mid R)) \mid \textit{where } P, Q, R \textit{ are CCS processes}\} \ .\end{array}$$

Conclude that, for all P, Q, R,

$$\begin{aligned} P \mid Q &\sim Q \mid P && (13)\\ P \mid \mathbf{0} &\sim P \quad \textit{and} && (14)\\ (P \mid Q) \mid R &\sim P \mid (Q \mid R) \ . && (15)\end{aligned}$$

In what follows, we shall sometimes 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 proposition states that this is indeed possible for strong bisimilarity.

Proposition 5.1 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 $P' \mid R' \stackrel{\alpha}{\to} S$ for some action α and CCS process S, then $Q' \mid R' \stackrel{\alpha}{\to} T$ 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 $P' \mid R' \stackrel{\alpha}{\to} S$ 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 $P' \mid R' \stackrel{\alpha}{\to} S$. Recall that the transition we are considering must be provable using the SOS rules for parallel composition given in Table 6. Therefore there are three possible forms that the transition $P' \mid R' \stackrel{\alpha}{\to} S$ may take, namely:

1. P' is responsible for the transition and R' "stands still"—that is, $P' \mid R' \stackrel{\alpha}{\to} S$ because $P' \stackrel{\alpha}{\to} P''$ and $S = P'' \mid R'$, for some P'',
2. R' is responsible for the transition and P' "stands still"—that is, $P' \mid R' \stackrel{\alpha}{\to} S$ because $R' \stackrel{\alpha}{\to} R''$ 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, $P' \mid R' \stackrel{\alpha}{\to} S$ because $\alpha = \tau$, $P' \stackrel{a}{\to} P''$, $R' \stackrel{\bar{a}}{\to} R''$ and $S = P'' \mid R''$, for some P'' and R''.

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

1. Since $P' \stackrel{\alpha}{\to} P''$ and $P' \sim Q'$, we have that $Q' \stackrel{\alpha}{\to} Q''$ and $P'' \sim Q''$, for some Q''. Using the transition $Q' \stackrel{\alpha}{\to} Q''$ as premise in the first rule for parallel composition in Table 6, we can infer that

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

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

   $$(P'' \mid R', Q'' \mid R') \in \mathcal{R} \ .$$

   We can therefore take $T = Q'' \mid R'$, and we are done.

2. In this case, we have that $R' \xrightarrow{\alpha} R''$. Using this transition as premise in the second rule for parallel composition in Table 6, we can infer that

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

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

$$(P' \mid R'', Q' \mid R'') \in \mathcal{R} \ .$$

We can therefore take $T = Q' \mid R''$, and we are done.

3. Since $P' \xrightarrow{a} P''$ and $P' \sim Q'$, we have that $Q' \xrightarrow{a} Q''$ and $P'' \sim Q''$, for some Q''. Using the transitions $Q' \xrightarrow{a} Q''$ and $R' \xrightarrow{\bar{a}} R''$ as premises in the third rule for parallel composition in Table 6, we can infer that

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

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

$$(P'' \mid R'', Q'' \mid R'') \in \mathcal{R} \ .$$

We can therefore take $T = Q'' \mid R''$, and we are done.

Therefore the relation $\mathcal{R}$ is a bisimulation, as claimed.

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

  Consider the relation

$$\mathcal{R} = \{(P' \setminus L, Q' \setminus L) \mid P' \sim Q'\} \ .$$

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

You are strongly encouraged to fill in the missing details in the proof. □

Exercise 5.9 *Prove that $\sim$ is preserved by action prefixing, summation and relabelling.*

Exercise 5.10 *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:*

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

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

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,

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

Recall that we defined the specification of a counter thus:

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

Moreover, we hinted at the fact that that process was "behaviourally equivalent" to the process C defined by

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

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

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

is a strong bisimulation. (Can you see why?) Indeed we have that:

Proposition 5.2 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 $C \mid \Pi_{i=1}^{k} P_i \xrightarrow{\alpha} P$ for some action α and process P, then there is some process Q such that $\text{Counter}_n \xrightarrow{\alpha} Q$ and $P \; \mathcal{R} \; Q$, and

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

We establish these two claims separately.

1. Assume that $C \mid \Pi_{i=1}^{k} P_i \xrightarrow{\alpha} P$ 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

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

   In the latter, argue that the matching transition is

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

2. Assume that $\text{Counter}_n \xrightarrow{\alpha} Q$ for some some action 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.

□

Exercise 5.11 *Fill in the missing details in the above proof.*

Exercise 5.12 (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 $s_1 \xrightarrow{\alpha} s_1'$, then there is a transition $s_2 \xrightarrow{\alpha} s_2'$ such that $s_1' \mathcal{R} s_2'$.*

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

1. *Prove that $\sqsubseteq$ is a preorder.*

2. *Build simulations showing that*

$$\begin{aligned} a.\mathbf{0} &\sqsubseteq a.a.\mathbf{0} \quad \textit{and} \\ a.b.\mathbf{0} + a.c.\mathbf{0} &\sqsubseteq a.(b.\mathbf{0} + c.\mathbf{0}) \ . \end{aligned}$$

   Do the converse relations hold?

3. *Show that strong bisimilarity is included in simulation equivalence. Does the converse inclusion also hold?*

Exercise 5.13 (For the theoretically minded) *Consider the processes*

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

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 5.2);*

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 relation may be necessary to validate some desirable equivalences.

5.3 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 Section 5.) 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 establish equivalences between process descriptions and it suffices to establish several natural equivalences. For instance, you used strong bisimilarity in Exercise 5.8 to justify the expected equalities

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

Moreover, a wealth of other "structural equivalences" like the ones above may be proven to hold modulo strong bisimilarity. (See [12, 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 these notes 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

$$\begin{tikzcd}
 & \mathrm{Start} \arrow[d, "\overline{\mathrm{pub}}"] & \\
 & (\mathrm{CM}_b \mid \mathrm{CS}') \setminus \{\mathrm{coin}, \mathrm{coffee}\} \arrow[dl, "\tau"'] \arrow[dr, "\tau"] & \\
\mathrm{Good} \arrow[d, "\tau"'] & & \mathrm{Bad} \\
\mathrm{Start} & &
\end{tikzcd}$$

where

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\mathrm{Start} & \overset{\mathrm{def}}{=} & (\mathrm{CM}_b \mid \mathrm{CS}) \setminus \{\mathrm{coin}, \mathrm{coffee}\} & \mathrm{CS} & \overset{\mathrm{def}}{=} & \overline{\mathrm{pub}}.\mathrm{CS}' \\
\mathrm{Good} & \overset{\mathrm{def}}{=} & (\overline{\mathrm{coffee}}.\mathrm{CM}_b \mid \mathrm{CS}'') \setminus \{\mathrm{coin}, \mathrm{coffee}\} & \mathrm{CS}' & \overset{\mathrm{def}}{=} & \overline{\mathrm{coin}}.\mathrm{CS}'' \\
\mathrm{Bad} & \overset{\mathrm{def}}{=} & (\mathrm{CM}_b \mid \mathrm{CS}'') \setminus \{\mathrm{coin}, \mathrm{coffee}\} & \mathrm{CS}'' & \overset{\mathrm{def}}{=} & \mathrm{coffee}.\mathrm{CS} \ .
\end{array}$$

Table 7: The behaviour of $(\mathrm{CM}_b \mid \mathrm{CS}) \setminus \{\mathrm{coin}, \mathrm{coffee}\}$

"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 (3), interact with a nasty variation on the coffee machine CM from (1). This latest version of the coffee machine delivered to the computer scientist's office is given by:

$$\mathrm{CM}_b \overset{\mathrm{def}}{=} \mathrm{coin}.\overline{\mathrm{coffee}}.\mathrm{CM}_b + \mathrm{coin}.\mathrm{CM}_b \ . \tag{16}$$

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 7 describe the possible behaviours of the system $(\mathrm{CM}_b \mid \mathrm{CS}) \setminus \{\mathrm{coin}, \mathrm{coffee}\}$. Note that, there are two possible τ-transitions that stem from the process $(\mathrm{CM}_b \mid \mathrm{CS}') \setminus \{\mathrm{coin}, \mathrm{coffee}\}$, and that one of them, namely

$$(\mathrm{CM}_b \mid \mathrm{CS}') \setminus \{\mathrm{coin}, \mathrm{coffee}\} \xrightarrow{\tau} (\mathrm{CM}_b \mid \mathrm{CS}'') \setminus \{\mathrm{coin}, \mathrm{coffee}\} \ ,$$

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 about 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. 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 5.3 Let P and Q be CCS processes. We write $P \overset{\epsilon}{\Rightarrow} Q$ iff there is a (possibly empty) sequence of τ-labelled transitions that leads from P to Q. (If the sequence is empty, then $P = Q$.)

For each action α, we write $P \overset{\alpha}{\Rightarrow} Q$ iff there are processes P' and Q' such that

$$P \overset{\epsilon}{\Rightarrow} P' \xrightarrow{\alpha} Q' \overset{\epsilon}{\Rightarrow} Q \ .$$

For each action α, we use $\hat{\alpha}$ to stand for ϵ if $\alpha = \tau$, and for α otherwise.

Thus $P \overset{\alpha}{\Rightarrow} Q$ holds if P can reach Q by performing an α-labelled transition, possibly preceded and followed by sequences of τ-labelled transitions. For example, $a.\tau.\mathbf{0} \overset{a}{\Rightarrow} \mathbf{0}$ and $a.\tau.\mathbf{0} \overset{a}{\Rightarrow} \tau.\mathbf{0}$ both hold.

In the LTS depicted in Table 7, apart from the obvious one step $\overline{\mathrm{pub}}$-labelled transition, we have that

$$\begin{array}{lcl} \text{Start} & \overset{\overline{\text{pub}}}{\Rightarrow} & \text{Good} \\ \text{Start} & \overset{\overline{\text{pub}}}{\Rightarrow} & \text{Bad} \quad \text{and} \\ \text{Start} & \overset{\overline{\text{pub}}}{\Rightarrow} & \text{Start} \ . \end{array}$$

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 5.4 [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:

- if $s_1 \xrightarrow{\alpha} s_1'$, then there is a transition $s_2 \overset{\hat{\alpha}}{\Rightarrow} s_2'$ such that $s_1' \mathcal{R} s_2'$;
- if $s_2 \xrightarrow{\alpha} s_2'$, then there is a transition $s_1 \overset{\hat{\alpha}}{\Rightarrow} s_1'$ 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*.

We can readily argue that $a.\mathbf{0} \approx a.\tau.\mathbf{0}$ by establishing a weak bisimulation that relates these two processes. On the other hand, there is no weak bisimulation that relates the process SmUni and the process Start in Table 7. In fact, the process SmUni is observationally equivalent to the process

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

but the process Start is not.

Exercise 5.14 *Prove the claims that we have just made.*

Exercise 5.15 *Prove that the behavioural equivalences claimed in Exercise 4.6 hold with respect to observational equivalence.*

The definition of weak bisimulation and observational equivalence 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

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

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

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

(Prove this!) This seems to be non-controversial until we note that A? and B? have a livelock (that is, a possibility of divergence), 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".

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

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

Table 8: The sender, receiver and medium in (17)

(See [2], 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, where the communication media may lose messages, and messages may have to be retransmitted some arbitrary number of times in order to ensure their delivery.

Note moreover that **0** is observationally equivalent to the process

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

This means that a process that can only diverge is observationally equivalent to deadlock. This may seem odd at first sight. However, you will probably agree that, assuming that we can only observe a process by communicating with it, these two systems 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

$$(\text{Send} \mid \text{Med} \mid \text{Rec}) \setminus L \qquad (L = \{\text{send}, \text{error}, \text{trans}, \text{ack}\}) \tag{17}$$

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 8. No that the potentially faulty behaviour of the medium Med is described abstractly in the defining equation for process Med$'$ by means of an internal transition to an "error state". When it has entered that state, the medium informs the sender process that it has lost a message, and therefore that the message must be retransmitted. The sender will receive this message when in state Wait, and will proceed to retransmit the message.

We expect the protocol to behave like a one-place buffer described thus:

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

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 5.16 *Build the aforementioned weak bisimulation.*

Theorem 5.2 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 $s_1 \xrightarrow{\alpha} s_1'$, then there is a transition $s_2 \overset{\hat{\alpha}}{\Rightarrow} s_2'$ such that $s_1' \approx s_2'$;
   - if $s_2 \xrightarrow{\alpha} s_2'$, then there is a transition $s_1 \overset{\hat{\alpha}}{\Rightarrow} s_1'$ such that $s_1' \approx s_2'$.

Proof: The proof follows the lines of that of Theorem 5.1, and is therefore omitted. □

Exercise 5.17 *Fill in the details of the proof of the above theorem.*

Exercise 5.18 *Show that strong bisimilarity is included in observational equivalence.*

Exercise 5.19 *Show that, for all P, Q, the following equivalences, that are usually referred to as Milner's τ-laws, hold:*

$$\alpha.\tau.P \approx \alpha.P \tag{18}$$
$$P + \tau.P \approx \tau.P \tag{19}$$
$$\alpha.(P + \tau.Q) \approx \alpha.(P + \tau.Q) + \alpha.Q \enspace . \tag{20}$$

Exercise 5.20 *Show that, for all P, Q, if $P \overset{\epsilon}{\Rightarrow} Q$ and $Q \overset{\epsilon}{\Rightarrow} P$, then $P \approx Q$.*

Exercise 5.21 *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 5.22 *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?*

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

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

In fact, the transition $a.\mathbf{0} + \tau.\mathbf{0} \xrightarrow{\tau} \mathbf{0}$ can only be matched by $a.\mathbf{0} + \mathbf{0} \overset{\epsilon}{\Rightarrow} a.\mathbf{0} + \mathbf{0}$, and the processes $\mathbf{0}$ and $a.\mathbf{0} + \mathbf{0}$ are not observationally equivalent. Fortunately, however, we have that:

Proposition 5.3 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 $R \mid P \sim R \mid Q$, 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 5.1, and is left as an exercise for the reader. □

Exercise 5.23 *Prove Proposition 5.3.*

In light of Proposition 5.3, 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 won't touch upon in these notes. The interested reader is referred to [12, Chapter 7] for an in depth treatment of this beautiful topic.

6 Hennessy-Milner Logic

In the previous section we have seen that implementation verification is a natural approach to establishing the correctness of (models of) reactive systems described 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 [3] for an investigation of this issue.) In fact, checking whether a process affords this property seems best done by first constructing the collection of initial a-labelled transitions that are possible for the process under consideration, and then checking whether this collection is empty.

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 either a $\overline{\text{coffee}}$-labelled transition or a $\overline{\text{tea}}$-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 [7]. 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 6.1 The set of Hennessy-Milner formulae over a set of actions Act (from now on referred to as $\mathcal{M}$) is given by the following abstract syntax:

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

where $a \in \mathsf{Act}$. 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* have 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 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 given 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 6.2 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 $\langle\cdot a\cdot\rangle, [\cdot a\cdot] : \mathcal{P}(\mathsf{Proc}) \to \mathcal{P}(\mathsf{Proc})$ defined by

$$\begin{array}{rcl} \langle\cdot a\cdot\rangle S &=& \{p \in \mathsf{Proc} \mid \exists p'.\, p \xrightarrow{a} p' \text{ and } p' \in S\} \quad \text{and} \\ {[\cdot a\cdot]} S &=& \{p \in \mathsf{Proc} \mid \forall p'.\, p \xrightarrow{a} p' \implies p' \in S\}. \end{array}$$

We write $p \models F$ iff $p \in [\![F]\!]$.

Two formulae are *equivalent* if, and only if, they are satisfied by the same processes in every transition system.

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 $\xrightarrow{a}$. 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 $P \xrightarrow{a} Q$ for some Q.

How to 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:

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

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:

$$\begin{aligned} [\![[\text{tea}]ff]\!] &= [\cdot \text{tea} \cdot][\![ff]\!] \\ &= [\cdot \text{tea} \cdot]\emptyset \\ &= \{P \mid \forall P'. \, P \overset{\text{tea}}{\to} P' \implies P' \in \emptyset\} \\ &= \{P \mid P \overset{\text{tea}}{\not\to}\} \ . \end{aligned}$$

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

$$P \overset{\text{tea}}{\not\to} \text{ iff } (\forall P'. \, P \overset{\text{tea}}{\to} P' \implies P' \in \emptyset) \ .$$

To see that this holds, observe first of all that if $P \overset{\text{tea}}{\to} Q$ for some Q, then it is not true that $P' \in \emptyset$ for all P' such that $P \overset{\text{tea}}{\to} P'$. 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 $P \overset{\text{tea}}{\not\to}$. Then it is vacuously true that $P' \in \emptyset$ for all P' such that $P \overset{\text{tea}}{\to} P'$—indeed, since there is no such P', there is no counter-example to that statement!

To sum up, we can express that a process can*not* perform action $a \in \mathsf{Act}$ with the formula $[a]\mathit{ff}$.

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

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

Exercise 6.1

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 after having drunk two coffees in a row.*
3. *What do the formulae $\langle a\rangle\mathit{ff}$ and $[a]\mathit{tt}$ express?*

Exercise 6.2 *Consider an everlasting clock whose behaviour is defined thus:*

$$\mathit{Clock} \overset{\mathit{def}}{=} \mathit{tick}.\mathit{Clock} \quad .$$

Prove that the process Clock satisfies the formula

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

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

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

Exercise 6.3 (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 6.2. This can be obtained by defining the binary relation $\models$ relating processes to formulae by structural induction on formulae thus:

- $P \models tt$, for each P,
- $P \models ff$, 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 $P \xrightarrow{a} P'$ for some P' such that $P' \models F$, and
- $P \models [a]F$ iff $P' \models F$, for each P' such that $P \xrightarrow{a} P'$.

Exercise 6.4 *Show that the above definition of the satisfaction relation is equivalent to that given in Definition 6.2. [Hint: Use induction on the structure of formulae.]*

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 by structural recursion on 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 6.1 Let $(\mathsf{Proc}, \mathsf{Act}, \{\xrightarrow{a} \mid a \in \mathsf{Act}\})$ 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 6.5

1. *Prove Proposition 6.1.*
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 6.1, 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 6.3 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

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

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

$$\mathrm{A} \xrightarrow{a} \mathrm{A} \ .$$

This transition can only be matched by either

$$\mathrm{B} \xrightarrow{a} \mathbf{0}$$

or

$$\mathrm{B} \xrightarrow{a} a.\mathrm{B} \ .$$

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,

$$a.\mathrm{B} \xrightarrow{a} \mathrm{B}$$

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 $\langle a\rangle\langle a\rangle[a]\mathit{ff}$—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 beautiful, 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 6.3 [Image Finite Process] A process P is *image finite* iff the collection $\{P' \mid P \xrightarrow{a} P'\}$ is finite for each action a.

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

For example, the process $!A$ defined thus:

$$!A \overset{\text{def}}{=} a.\mathbf{0}|!A$$

is *not* image finite. In fact, you should be able to prove by induction on n that, for each $n \geq 0$,

$$!A \xrightarrow{a} \underbrace{a.\mathbf{0} \mid \cdots \mid a.\mathbf{0}}_{n \text{ times}} |\mathbf{0}|!A \quad .$$

Another example of a process that is not image finite is

$$\mathrm{A}^{<\omega} \overset{\text{def}}{=} \sum_{i \geq 0} a^i \quad , \tag{21}$$

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 6.1 [Hennessy and Milner [7]] Let $(\mathsf{Proc}, \mathsf{Act}, \{\xrightarrow{a} \mid a \in \mathsf{Act}\})$ 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 $\mathcal{M}$.

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 $Q \xrightarrow{a} Q'$ for some Q'. We wish to show that $Q' \models G$. Now, since $P \sim Q$ and $Q \xrightarrow{a} Q'$, there is a process P' such that $P \xrightarrow{a} P'$ 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 $Q \xrightarrow{a} Q'$ 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. To this end, assume that $R \; \mathcal{R} \; S$ and $R \xrightarrow{a} R'$. We shall now argue that there is a process S' such that $S \xrightarrow{a} S'$ and $R' \; \mathcal{R} \; S'$. Since $\mathcal{R}$ is symmetric, this suffices to establish that $\mathcal{R}$ is a strong bisimulation.

  Assume, towards a contradiction, that there is no S' such that $S \xrightarrow{a} S'$ and S' satisfies the same properties as R'. Since S is image finite, the set of processes S can reach by performing an a-labelled transition is finite, say $\{S_1, \ldots, S_n\}$ with $n \geq 0$. By our assumption, none of the processes in the above set satisfies the same formulae as R'. So, for each $i \in \{1, \ldots, n\}$, there is a formula F_i such that

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

  (Why? Couldn't it 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 6.6 *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 6.1 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.

Exercise 6.7 (For the Theoretically Minded) *Consider the process A^ω given by:*

$$A^\omega \overset{def}{=} a.A^\omega \quad .$$

Show that the processes $A^{<\omega}$ and $A^\omega + A^{<\omega}$, where $A^{<\omega}$ was defined in (21),

1. *are not strongly bisimilar, but*
2. *satisfy the same properties in $\mathcal{M}$.*

Conclude that Theorem 6.1 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.]

7 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. 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 Sect. 7.6, where a formal definition of the modal depth of a formula will be given.)

However, we often wish to describe properties that describe states of affairs that may or must occur in arbitrarily long computations of a process. If we want to express properties as, for example, that a process is *always* able to perform a given

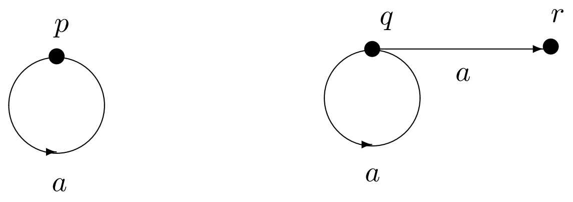


Figure 1: Two processes.

action, we have to extend the logic. One way of doing this is to allow for infinite conjunctions and disjunctions in our property language.

Example 7.1 Consider the processes p and q in Figure 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:

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

Since a difference in the behaviour of the two processes can already be found by examining their behaviour after two transitions, a formula that distinguishes them is “small”.

Assume, however, that we modify the labelled transition system for q by adding a sequence of transitions to r thus:

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

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:

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

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 1 can always (i.e., at any point in a computation) 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 HML as an infinite conjunction thus:

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

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 $[a]^i\langle a\rangle tt$).

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:

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

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 state in which an a can be refused by performing a sequence of i actions (as expressed by the disjunct $\langle a\rangle^i[a]ff$).

Even if it is theoretically possible to extend HML with infinite conjunctions and disjunctions, infinite formulas 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 simple; 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:

$$X \equiv \langle a\rangle tt \wedge [a]X, \qquad (22)$$

where we write $F \equiv G$ if and only if the formulas F and G are satisfied by exactly the same processes (i.e., $[\![F]\!] = [\![G]\!]$.) The above recursive equation captures the intuition that a process that can invariantly perform an a-labelled transition—that is, 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. However, the mere fact of writing down an equation like (22) 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

$$x = x + 1 \tag{23}$$

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

$$X = \mathbb{N} \setminus X \ . \tag{24}$$

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

$$X = \{2\} \cup X \ , \tag{25}$$

namely all of the sets that contain the number 2. There are also equations that have a finite number of solutions, but not a unique one. As an example, consider the equation

$$X = \{10\} \cup \{n - 1 \mid n \in X, \ n \neq 0\} \ . \tag{26}$$

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 7.1 *Check the claims that we have just made.*

Since an equation like (22) is meant to describe a formula, it is therefore natural to ask ourselves the following questions:

- Does (22) have a solution? And what precisely do we mean by that?
- If (22) has more than one solution, which one do we choose?
- How can we compute whether a process satisfies the formula described by (22)?

Precise answers to these questions will be given in the remainder of this section. However, 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 (22) 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 $[a] ff$). But $p \notin \emptyset$ so this cannot be the solution we are looking for. Actually it turns out that it is the *maximal* solution we need here, namely where $S = \{p\}$. The set $S = \emptyset$ is the *minimal solution*.

In other cases it is the minimal 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 maximal solution is $Y = \{p, q, r\}$ but, as p cannot terminate at all, this is not the solution we are interested in. The minimal solution is $Y = \{q, r\}$ and is exactly the set of processes that intuitively satisfy $Pos([a] ff)$.

When we write down a recursively defined property, we can indicate whether we desire the minimal or the maximal solution by adding this information to the equality sign. For $Inv(\langle a \rangle tt)$ we want the maximal solution, and in this case we write

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

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

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

More generally we can express that the formula F holds for each reachable state (written $Inv(F)$, and read "invariantly F") by means of the equation

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

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

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

Intuitively, we use maximal 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 minimal 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 7.2 *Give a formula, built using HML and the temporal operators Pos and/or Inv, that expresses a property distinguishing the processes in Exercise 6.7.*

7.1 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 $\mathit{Safe}(F)$ that is satisfied by a process p whenever it has a complete transition sequence

$$p = p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} p_2 \cdots$$

where each of the processes p_i satisfies F. (A transition sequence is *complete* if it is infinite or its last state affords no transition.) This *invariance of F under some computation* can be expressed in the following way:

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

It turns out to be the maximal solution that is of interest here as we will argue for formally later.

A process p satisfies the property $\mathit{Even}(F)$ 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 can eventually reach a state that has property F. This can be expressed by means of the following equation:

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

In this case we are interested in the minimal solution because $\mathit{Even}(F)$ should only be satisfied by those processes that can be reached from p by a finite number of transitions.

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

- $F \, \mathcal{U}^s \, G$, 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 reaches 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 reaches until it gets into state where G holds (but maybe this will never happen!).

We express these operators as follows:

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

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 $Inv(F)$. Thus $Even(G) \equiv tt\,\mathcal{U}^s\,G$ 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 and 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 Aristoteles. Within the last 20 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 [4, 17].

The *modal μ-calculus* [10] is a generalization of Hennessy-Milner logic with recursion that allows for maximal and minimal 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 minimal fixed points are used to express that something will happen sooner or later, whereas the maximal fixed points are used to express invariance of some state of affairs during computations, or that something does *not* happen as a system evolves.

7.2 Syntax and semantics of Hennessy-Milner logic with recursion

The first step towards introducing recursion in HML is to add variables to the syntax. To start with we only consider *one* recursively defined property. We will return to the more general case of properties defined by *mutual recursion* later.

The syntax for Hennessy-Milner-logic with one variable X, $\mathcal{M}_{\{X\}}$, is given by the following syntax:

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

$$\begin{tikzcd}
 & p_1 \arrow[dl, "a"'] \arrow[dr, shift right, "b"'] & \\
p_2 & & p_3 \arrow[ul, shift right, "a"'] \arrow[ll, "a"]
\end{tikzcd}$$

Figure 2: A process

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

Example 7.2 Consider the formula $F = \langle a \rangle X$ and let Proc be the set of states in the transition graph in Figure 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

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

The above intuition is captured formally in the following definition.

Definition 7.1 Let $(\mathsf{Proc}, \mathsf{Act}, \{ \xrightarrow{a} \mid a \in \mathsf{Act}\})$ be a labelled transition system. For each $S \subseteq \mathsf{Proc}$ and formula F, we define $\mathcal{O}_F(S)$ inductively by:

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

Exercise 7.3 *Given the transition graph from Example 7.2, 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* over the complete lattice $(\mathcal{P}(\mathsf{Proc}), \subseteq)$. 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 7.4 *Show that* $\mathcal{O}_F$ *is monotonic for all* F. *Consider what will happen if we introduce negation into our logic.*

As mentioned before, the idea underlying the definition of the function $\mathcal{O}_F$ 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

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

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

$$[\![X]\!] = \mathcal{O}_{F_X}([\![X]\!]). \tag{27}$$

As $\mathcal{O}_{F_X}$ is a monotonic function over a complete lattice we know that (27) has solutions, i.e., that $\mathcal{O}_{F_X}$ has fixed points. In particular Tarski's Fixed Point Theorem gives us that there is a unique *maximal* fixed point, denoted by FIX $\mathcal{O}_{F_X}$, and also a unique *minimal* one, denoted by fix $\mathcal{O}_{F_X}$, given respectively by

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

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 *pre fixed point* for $\mathcal{O}_{F_X}$ if $\mathcal{O}_{F_X}(S) \subseteq S$.

In what follows, for a function $f : \mathcal{P}(\mathsf{Proc}) \longrightarrow \mathcal{P}(\mathsf{Proc})$ we define

$f^0 = id_{\mathcal{P}(\mathsf{Proc})}$ the identity function on $\mathcal{P}(\mathsf{Proc})$ and
$f^{m+1} = f \circ f^m$.

When Proc is finite we have the following characterization of the maximal and minimal fixed points.

Theorem 7.1 If Proc is finite then FIX $\mathcal{O}_{F_X} = (\mathcal{O}_{F_X})^M(\mathsf{Proc})$ for some M and fix $\mathcal{O}_{F_X} = (\mathcal{O}_{F_X})^m(\emptyset)$ for some m.

Proof: Follows directly from the fixed point theorem for finite complete lattices. See Appendix A for the details. □

7.3 Maximal fixed points and invariant properties

In this section we shall have a closer look at the meaning of formulae defined by means of maximal fixed point. More precisely we consider an equation of the form

$$X \overset{\max}{=} F_X$$

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

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

We have previously given an informal argument for why *invariant* properties are obtained as maximal 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 maximal 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 under 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 maximal solution given by

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

To show that FIX $\mathcal{I}$ indeed characterizes precisely the set of processes for which all derivation 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:

$$Inv = \{p \mid \forall s \in \mathsf{Act}^*, p' \in \mathsf{Proc}\ .\ p \xrightarrow{s} p' \Rightarrow p' \in [\![F]\!]\}.$$

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

Theorem 7.2 For every labelled transition system $(\mathsf{Proc}, \mathsf{Act}, \{\xrightarrow{a} \mid a \in \mathsf{Act}\})$, it holds that $Inv \ = \ \text{FIX}\,\mathcal{I}$.

Proof: We prove the statement by proving each of the inclusions $Inv \subseteq \text{FIX}\,\mathcal{I}$ and $\text{FIX}\,\mathcal{I} \subseteq Inv$ 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 s, p',

$$p \xrightarrow{s} p' \text{ implies that } p' \in [\![F]\!]. \qquad (28)$$

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 letting $s = \varepsilon$ in (28) 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,

$$p \xrightarrow{a} p' \text{ implies } p' \in Inv,$$

which is equivalent to proving that, for each sequence of actions s' and process p'',

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

However, this follows immediately by letting $s = as'$ in (28).

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

$$\text{FIX}\,\mathcal{I} = [\![F]\!] \cap [\cdot\mathsf{Act}\cdot]\text{FIX}\,\mathcal{I}. \qquad (29)$$

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

Base case $s = \varepsilon$: Then $p = p'$ and therefore, by (29), it holds that $p' \in [\![F]\!]$, which was to be shown.

Inductive step $s = as'$: Then $p \xrightarrow{a} p'' \xrightarrow{s'} p'$ for some p''. By (29), it follows that $p'' \in \text{FIX}\,\mathcal{I}$. As $|s'| < |s|$ 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.

□

7.4 Mutually recursive equational system

As you may have noticed, so far we have only allowed one equation with one variable in our recursive definitions. A *mutually recursive equational system* has the form

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

where $\mathcal{X} = \{X_1, \ldots, X_n\}$ is a set of variables and, for $i \leq n$, the formula F_{X_i} is in $\mathcal{M}_{\mathcal{X}}$, and can therefore contain any variable from $\mathcal{X}$. An example of such an equational system is

$$\begin{array}{l} X = [a]Y \\ Y = \langle a \rangle X \ . \end{array}$$

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.

To define the semantics of such an equational system it is not enough to consider simply the complete lattice consisting of subsets of processes. Instead such a system is interpreted over n*-dimensional vectors* of sets of processes, , where n is the number of variables in $\mathcal{X}$. Thus the new domain is $\mathcal{D} = (\mathcal{P}(\mathsf{Proc}))^n$ (n-times cross product of $\mathcal{P}(\mathsf{Proc})$ with itself) with a partial order defined "component wise":

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

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

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

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

The semantic function $[\![D]\!] : \mathcal{D} \to \mathcal{D}$ that is used to obtain the maximal and minimal solutions of the system of recursive equations described by the declaration D is obtained from the syntax in the following way:

$$\begin{array}{l} [\![D]\!]([\![X_1]\!], \ldots, [\![X_n]\!]) = \\ \qquad (\mathcal{O}_{F_{X_1}}([\![X_1]\!], \ldots, [\![X_n]\!]), \ldots, \mathcal{O}_{F_{X_n}}([\![X_1]\!], \ldots, [\![X_n]\!])) \ , \end{array} \tag{30}$$

where each argument $[\![X_i]\!]$ ($1 \leq i \leq n$) can be replaced by an arbitrary $S \subseteq \mathsf{Proc}$.

The function $[\![D]\!]$ turns out to be monotonic over the complete lattice $(\mathcal{D}, \leq)$, and we can obtain both the maximal and minimal fixed point for the equational system in the same way as for the case of one variable.

Exercise 7.5

1. *Show that* $(\mathcal{P}(\mathsf{Proc})^n, \leq, \bigsqcup, \bigsqcap)$*, with* $\leq$*,* $\bigsqcup$ *and* $\bigsqcap$ *defined as described in the text above, is a complete lattice.*

2. *Show that (30) defines a monotonic function*

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

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

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

7.5 A Proof System for Maximal Properties

In this section we will introduce a proof system that allows us to determine whether a process p satisfies a given property F, defined over variables which are declared as a maximal solution to a recursive equational system. In particular we will prove that the proof system is *sound* and *complete* in a sense that we will explain more precisely later. For the sake of simplicity, in our presentation we restrict ourselves to the setting in which there is only one recursion variable, namely X, with defining equation

$$X \overset{\max}{=} F_X \ .$$

The interested reader is referred to [11] for generalizations of the material we present in what follows.

The proof system is given in Table 9. It consists of a collection of inference rules of the form

$$\frac{\text{Premises}}{\text{Conclusion}}$$

As it is customary with inference rules, we can read them top-down or bottom-up. When read top-down, a rule intuitively states that if we have shown all of the premises above the solid line, then we can use the rule to infer the conclusion below the solid line. When read bottom-up, the rule says that in order to prove

TT	$\Gamma \vdash p : tt$	
AND	$\dfrac{\Gamma \vdash p : F_1 \quad \Gamma \vdash p : F_2}{\Gamma \vdash p : F_1 \wedge F_2}$	
OR	$\dfrac{\Gamma \vdash p : F_1}{\Gamma \vdash p : F_1 \vee F_2}$	$\dfrac{\Gamma \vdash p : F_2}{\Gamma \vdash p : F_1 \vee F_2}$
DIAMOND	$\dfrac{\Gamma \vdash p' : F}{\Gamma \vdash p : \langle a \rangle F}$	if $p \xrightarrow{a} p'$
BOX	$\dfrac{\Gamma \vdash p_1 : F \ \cdots \ \Gamma \vdash p_n : F}{\Gamma \vdash p : [a]F}$	if $\{p_1,\ldots,p_n\}=\{p \mid p \xrightarrow{a} p'\}$
MAX1	$\Gamma, p : X \vdash p : X$	
MAX2	$\dfrac{\Gamma, p : X \vdash p : F_X}{\Gamma \vdash p : X}$	$X \overset{\max}{=} F_X$

Table 9: Proof system for maximal properties

$$\cfrac{\emptyset \vdash p : X}{\cfrac{p : X \vdash p : \langle a\rangle \mathit{tt} \wedge [a]X}{\cfrac{p : X \vdash p : [a]X}{p : X \vdash p : X}\,\mathsf{BOX} \qquad \cfrac{p : X \vdash p : \langle a\rangle \mathit{tt}}{p : X \vdash p : \mathit{tt}}\,\mathsf{DIAMOND}}\,\mathsf{AND}}\,\mathsf{MAX2}$$

Figure 3: Proof for $p \models \mathit{Inv}(\langle a\rangle \mathit{tt})$

the conclusion, it is sufficient to establish the premises—which now become our sub-goals. A rule without premises is usually called an *axiom*.

The statements of the proof system are of the form $\Gamma \vdash p : F$, where Γ is a set of *hypotheses* of the form

$$\{p_1 : X, \ldots, p_n : X\} \ .$$

We also refer to such statements as *sequents*. The intuitive interpretation of the sequent $\Gamma \vdash p : F$ is as follows:

> Given that the sequence of hypotheses $\Gamma = \{p_1 : X, \ldots, p_n : X\}$ holds (i.e., $p_i \in [\![X]\!]$ for each i), the process p satisfies the property F (i.e., $p \in [\![F]\!]$).

For instance, the statement in axiom TT states the intuitively obvious fact that each process p satisfies the formula tt, no matter what our assumptions are. On the other hand, axiom MAX1 says that p satisfies X, if we assume so.

We say that a statement $\Gamma \vdash p : F$ is provable if there exists a proof tree with only axioms occurring in the leaves (application of the rules TT or MAX1) and whose root is $\Gamma \vdash p : F$. If $\Gamma \vdash p : F$ is provable, we simply write $\Gamma \vdash p : F$.

Example 7.3 Let us consider the process p from Example 7.1. We can use the proof system in Table 9 to show that $p \models \mathit{Inv}(\langle a\rangle \mathit{tt})$. In Figure 3, we have a proof for this statement from an empty set of hypotheses. The leftmost leaf in that proof is an instance of the axiom MAX1, and the rightmost one follows by using axiom TT.

Proof systems like the one Table 9 are meant allow for purely symbolic reasoning about some reality of interest. Therefore the most desirable property of a proof system is that it only allows us to prove statements that are true in the universe of discourse. This property is called *soundness*. If a proof system is powerful enough to prove each true statement, then it is called *complete*.

We would like to prove that the proof system in Table 9 is sound and complete, i.e., that the following holds:

$$\emptyset \vdash p : F \text{ iff } p \in [\![F]\!]. \tag{31}$$

The claim of soundness of the proof system is expressed by the implication in (31) from left to right, whereas its completeness is stated in the opposite implication.

Lemma 7.1 Let $(\mathsf{Proc}, \mathsf{Act}, \{\xrightarrow{a} | a \in \mathsf{Act}\})$ be a labelled transition system. Then, for every process p and formula F, it holds that

$$p \in [\![F]\!] \text{ iff there is an } S \subseteq \mathsf{Proc} \text{ such that } S \subseteq \mathcal{O}_{F_X}(S) \;\text{ and }\; p \in \mathcal{O}_F(S). \tag{32}$$

Exercise 7.6 *Prove that Lemma 7.1 holds. (Hint: Use induction on the structure of the formula F.)*

By Lemma 7.1 the soundness and completeness of the proof system reduces to the statement

$$\emptyset \vdash p : F \text{ iff there is an } S \subseteq \mathsf{Proc} \text{ such that } S \subseteq \mathcal{O}_{F_X}(S) \;\text{ and }\; p \in \mathcal{O}_F(S).$$

The result we prove is more general than (31) as it involves non-empty sets of assumptions.

Theorem 7.3 (Soundness) Assume that X is given by the equation $X \overset{\max}{=} F_X$ in a maximal equational system E. Then

> $p_1 : X, \ldots, p_n : X \vdash p : F$ implies $S \subseteq \mathcal{O}_{F_X}(S) \cup \{p_1, \ldots, p_n\}$ for some S such that $p \in \mathcal{O}_F(S)$.

Proof: We prove the statement by induction on the depth of the derivation tree needed to prove that $p_1 : X, \ldots, p_n : X \vdash p : F$. We proceed by case analysis by investigating the last rule used in the proof.

Base case: The statement follows by applying axiom TT or `Max1`.

We examine these two possibilities separately.

TT In this case $F = tt$ and $\mathcal{O}_F(S) = \mathsf{Proc}$ for all S. It is therefore easy to find a subset S of Proc such that $S \subseteq \mathcal{O}_{F_X}(S) \cup \{p_1, \ldots, p_n\}$ and $p \in \mathcal{O}_F(S)$; we can, for instance, take $S = \emptyset$.

MAX1 Here we have that $p_1 : X, \ldots, p_n : X, p : X \vdash p : X$. It is again easy to see that there exists an S with the wanted property; choose $S = \{p\}$.

Inductive step: Now we consider the cases where the proof is completed by an application of an inference rule. Our induction hypothesis is that the statement holds for the statements that appear in the premises of the rules.

MAX2 We have concluded that $p_1 : X, \ldots, p_n : X \vdash p : X$ from

$$p_1 : X, \ldots, p_n : X, p : X \vdash p : F_X \quad .$$

By the induction hypothesis it must be the case that $S \subseteq \mathcal{O}_{F_X}(S) \cup \{p_1, \ldots, p_n, p\}$ and $p \in \mathcal{O}_{F_X}(S)$, for some S. We want an S' such that $S' \subseteq \mathcal{O}_{F_X}(S') \cup \{p_1, \ldots, p_n\}$ and $p \in \mathcal{O}_X(S')$. To obtain this we can let $S' = S \cup \{p\}$.

The proofs for the missing rules are left to the reader as an exercise. □

Exercise 7.7 *Complete the proof for Theorem 7.3. Why is it sufficient to choose $S' = S \cup \{p\}$ in the case when the rule* MAX2 *was considered?*

Exercise 7.8 *Prove that Theorem 7.3 implies (31).*

Now we know that our proof system is *sound*—anything we can prove by using the proof rules holds in the denotational semantics. Fortunately it is also the case that the proof system is *complete*, i.e., each statement that holds in the semantics can be proven by the system—at least if the transition graph is finite.

Theorem 7.4 (Completeness) Let p be a process in a *finite* transition graph. If there is an $S \subseteq \mathsf{Proc}$ such that $S \subseteq \mathcal{O}_{F_X}(S) \cup \{p_1, \ldots, p_n\}$ and $p \in \mathcal{O}_F(S)$, then we have that $p_1 : X, \ldots, p_n : X \vdash p : F$.

Proof: Assume that $(\mathsf{Proc}, \mathsf{Act}, \{a \in \mathsf{Act} \mid \xrightarrow{a}\})$ is a finite transition graph, i.e., that $\mathsf{Proc} = \{q_1, \ldots, q_k\}$ for some k. Assume that there exists an S such that $S \subseteq \mathcal{O}_{F_X}(S) \cup \{p_1, \ldots, p_n\}$ and $p \in \mathcal{O}_F(S)$. Now we prove that

$$p_1 : X, \ldots, p_n \vdash p : F$$

by induction on $|\mathsf{Proc} \setminus \{p_1, \ldots p_n\}|$, i.e., the number of elements in the complement of $\{p_1, \ldots, p_n\}$.

Base case: Then we have that $\{p_1, \ldots, p_n\} = \mathsf{Proc}$. We prove this case by (an inner) structural induction on F. We only give the details of the proof for three of the cases; the remaining cases we leave to the reader as an exercise.

$F = tt$**:** Follows immediately from the rule TT.

$F = F_1 \wedge F_2$**:** Now we have that $S \subseteq \mathcal{O}_{F_X}(S) \cup \{p_1, \ldots, p_n\}$ and $p \in \mathcal{O}_{F_1 \wedge F_2}(S)$. The definition of $\mathcal{O}_{F_1 \wedge F_2}$ gives that $p \in \mathcal{O}_{F_1}(S)$ and $p \in \mathcal{O}_{F_1}(S)$. From the hypothesis for the inner induction it now follows that $p_1 : X, \ldots p_k : X \vdash F_1$ and $p_1 : X, \ldots p_k : X \vdash F_2$. The rule AND gives us immediately that $p_1 : X, \ldots, p_n \vdash p : F_1 \wedge F_2$.

$F = X$**:** Here it is clear that $p \in \{p_1, \ldots, p_n\}$, and the result follows by MAX1.

Inductive step: Assume that the theorem holds for all $P \subseteq \mathsf{Proc}$ with $|P| < n - k$. Towards proving the statement we also assume that there exists an S such that $S \subseteq \mathcal{O}_{F_X}(S) \cup \{p_1, \ldots, p_n\}$ and $p \in \mathcal{O}_F(S)$. We will show that $p_1 : X, \ldots, p_n \vdash p : F$. Again we proceed by structural induction on F.

The only interesting case is when $F = X$. If $p \in \{p_1, \ldots, p_n\}$, we simply use MAX1. Therefore we may assume that $p \notin \{p_1, \ldots, p_n\}$. By the inductive hypothesis we know that that,

$$\begin{array}{c}\exists R.(R \subseteq \mathcal{O}_{F_X}(R) \text{ and } p \in \mathcal{O}_{F_X}(R) \cup \{p_1, \ldots, p_n, p\}) \text{ implies} \\ p_1 : X, \ldots, p_n : X, p : X \vdash p : F_X\end{array} \tag{33}$$

By our assumption there is an S such that $S \subseteq \mathcal{O}_{F_X}(S) \cup \{p_1, \ldots, p_n\}$ and $p \in \mathcal{O}_X(S)$. Obviously

$$S \subseteq \mathcal{O}_{F_X}(S) \cup \{p_1, \ldots, p_n, p\}. \tag{34}$$

Furthermore, as $\mathcal{O}_X(S) = S$, we have that

$$p \in \mathcal{O}_X(S) = S \subseteq \mathcal{O}_{F_X}(S) \cup \{p_1, \ldots, p_n\}.$$

By assumption $p \notin \{p_1, \ldots, p_n\}$ which in turn implies that

$$p \in \mathcal{O}_{F_X}(S). \tag{35}$$

Therefore, by the induction hypothesis (33) and by (34) and (35) we have that

$$p_1 : X, \ldots, p_n : X, p : X \vdash p : F_X.$$

By using MAX2 we can conclude that

$$p_1 : X, \ldots, p_n : X \vdash p : X \quad ,$$

which was to be shown.

This completes the proof. □

7.6 Characteristic properties

The characterization theorem for bisimulation equivalence in terms of Hennessy-Milner logic tells us that if our transition system is image finite, the equivalence classes of bisimulation equivalence are completely characterized by the logic—see [7] 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 formulas in HML as p—that is, letting $[p]_\sim = \{q \mid q \sim p\}$,

$$[p]_\sim = \{q \mid \forall F \in \mathcal{M}. p \models F \implies q \models F\}.$$

Exercise 7.9 *Note that in the above rephrasing of the characterization theorem for HML, we only require that each formula satisfied by p is also satisfied by q, but not that the converse also holds. Show, however, that if q satisfies all the formulas in HML satisfied by p, then p and q satisfy the same formulas in HML.*

In this section we will show that if our transition system is finite, by extending the logic with recursion, we can characterize the equivalence classes for strong bisimulation with a *single* formula. The formula that characterizes the bisimulation equivalence class for p is called the *characteristic formula for p*. (That such a 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 $\phi_p \in \mathcal{M}_\mathcal{X}$ for a suitable set of variables $\mathcal{X}$, such that for all processes q

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

Let us start by giving an example that shows that in general bisimulation equivalence cannot be characterized by a finite or recursion free formula.

Example 7.4 Assume that $\mathsf{Act} = \{a\}$ and that the process p is given by the equation

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

We will show that p cannot be characterized up to bisimulation equivalence by a single recursion free formula. To see this we assume that such a formula exists and show that this leads to a contradiction. Towards a contradiction, we assume that for some $\phi \in \mathcal{M}$,

$$[\![\phi]\!] = [p]_\sim. \tag{36}$$

In particular we have that

$$p \models \phi \quad \text{and} \quad \forall q.\, q \models \phi \implies q \sim p. \tag{37}$$

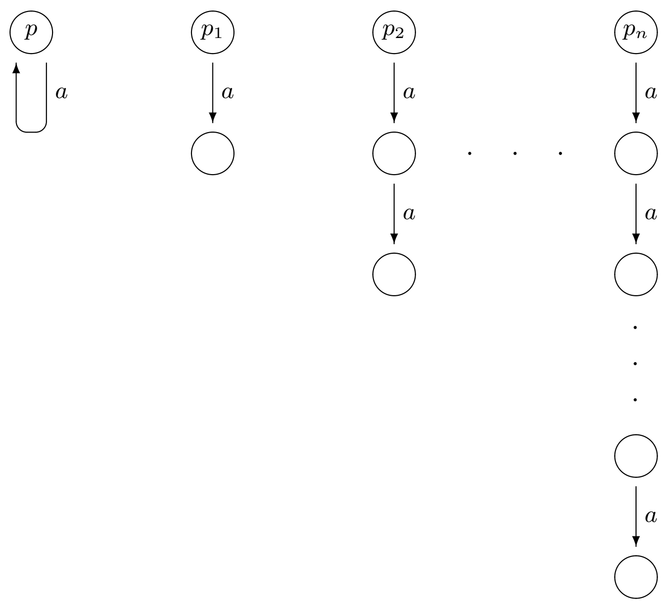


Figure 4: The processes p and p_i for $i \leq n$

We will obtain contradiction by proving that (37) cannot hold for any ϕ. Before we prove our statement we have to introduce some notation.

By the *modal depth* of the formula ϕ, notation $md(\phi)$, we mean the maximum number of nested occurrences of the model operators in ϕ. Formally this is defined by the following recursive definition:

1. $md(\mathit{tt}) = md(\mathit{ff}) = 0$,
2. $md([a]\phi) = md(\langle a \rangle \phi) = 1 + md(\phi)$,
3. $md(\phi_1 \vee \phi_2) = md(\phi_1 \wedge \phi_2) = \max\{md(\phi_1), md(\phi_2)\}$.

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 Fig. 7.4.) Now we can prove the following;

$$\forall \phi.\ p \models \phi \text{ implies } p_{md(\phi)} \models \phi. \tag{38}$$

The statement in (38) can be proven by structural induction on ϕ and is left as an exercise. As obviously p and p_n are not bisimulation equivalent for any n, the statement in (38) contradicts (37). As (37) is a consequence of (36), we can therefore conclude that no finite formula ϕ can characterize the process p up to bisimulation equivalence.

Example 7.4 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 both suggesting a (syntactic) equational system that describes the formula and to decide whether to adopt the minimal or the maximal solution to this system. We start by giving the equational system and choose the suitable interpretation for the fixed point afterwards.

We start by assuming that we have a finite transition system

$$(\{p_1, \ldots, p_n\}, \mathsf{Act}, \rightarrow)$$

and a set of variables $\mathcal{X} = \{X_{p_1}, \ldots, X_{p_n}, \ldots\}$ that contains (at least) as many variables as there are states in the transition system. Intuitively X_p is the syntactic symbol for the characteristic formula for p and its meaning will be given in terms of an equational system.

A characteristic formula for a process has to describe both which actions the process *can perform*, which action it *cannot perform* and what happens to it *after it has performed* each action. The following example illustrates this.

Example 7.5 If a coffee machine is given by Figure 5, we can construct a characteristic formula for it as follows.

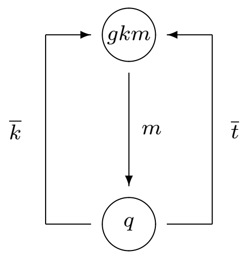


Figure 5: The nice coffee machine gkm.

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 gkm, by performing the m action necessarily will 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 other actions than m.

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

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

where $\langle m \rangle X_q$ expresses 1, $[m]X_q$ expresses 2 and $[\bar{t}, \bar{k}]\mathit{ff}$ expresses 3. To obtain the characteristic formula for gkm we have to express X_q following the same strategy. We observe that q can perform two actions $\bar{t}$ and $\bar{k}$ and in both cases it becomes gkm. X_q can therefore be expressed as

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

Now we can generalize the strategy employed in the above example as follows: Let

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

If $p' \in Der(a, p)$ and p' has a characteristic property $X_{p'}$, then p has the property $\langle a \rangle X_{p'}$. For any a we have that

$$p \models \bigwedge_{a, p'. p \xrightarrow{a} p'} \langle a \rangle X_{p'}$$

Furthermore, if $p \xrightarrow{a} p'$ then $p' \in Der(a, p)$. Therefore p has the property

$$[a] \bigvee_{p'. p \xrightarrow{a} p'} X_{p'} \ ,$$

and as a is arbitrary we have that

$$p \models \bigwedge_{a} [a] \bigvee_{p'. p \xrightarrow{a} p'} X_{p'}$$

If we summarize the above requirements, we have that

$$p \models \bigwedge_{a, p'. p \xrightarrow{a} p'} \langle a \rangle X_{p'} \ \wedge \ \bigwedge_{a} [a] \bigvee_{p'. p \xrightarrow{a} p'} X_{p'}$$

As this property is apparently a complete description p, this is our candidate for the characteristic property for p. X_p is therefore defined as a solution to the equational system obtained by giving the following equation for each $q \in \mathsf{Proc}$:

$$X_q \equiv \bigwedge_{a,q'.q \xrightarrow{a} q'} \langle a \rangle X_{q'} \wedge \bigwedge_{a} [a] \bigvee_{q'.q \xrightarrow{a} q'} X_{q'} \tag{39}$$

The solution can either be the minimal or the maximal one (or something in between).

The following example shows that the minimal solution to (39) in general does not yield the characteristic property for a process.

Example 7.6 Let p be the process given in Figure 6.

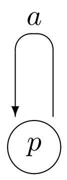


Figure 6: Simple infinite process p.

In this case the equational system obtained by (39) will have the form

$$X_p \overset{\min}{=} \langle a \rangle X_p \wedge [a] X_p$$

Obviously $X_p = ff$ is the minimal solution to this equation. However p does not have the property ff which therefore cannot be the characteristic property for p.

In what follows we will show that the maximal solution to (39) yields the characteristic property for all $p \in \mathsf{Proc}$. This is the content of the following theorem.

Theorem 7.5 Let $(\mathsf{Proc}, \mathsf{Act}, \rightarrow)$ be a finite transition system and for all $p \in \mathsf{Proc}$, let X_p be defined by

$$X_p \overset{\max}{=} \bigwedge_{a,p'.p \xrightarrow{a} p'} \langle a \rangle X_{p'} \wedge \bigwedge_{a} [a] \bigvee_{p'.p \xrightarrow{a} p'} X_{p'}. \tag{40}$$

Then X_p is the characteristic property for p.

The assumption about Proc and Act being finite ensures that there is only a finite number of variables involved in the definition of the characteristic formula and that each case that we only get a finite formula (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

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

From the previous discussion, we get that X_p is the characteristic property for p if and only if for the maximal solution $[\![X_p]\!]$, where $p \in$ 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 maximal solution for D_K.

As the first step in the proof of Theorem 7.5, we prove the following lemma:

Lemma 7.2 Let X_p be defined as in (40). Then 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 $p \sim q$. Therefore we have to prove the following:

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 $(p_1, q_1) \in R$.

a) Let $(p, q) \in R$ and $p \xrightarrow{b} p_1$, which in turn means that

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

From the definitions of X_p and D_K it follows that

$$q \models_{max} X_p \overset{\text{max}}{=} q \models_{max} (\bigwedge_{a,p'.p \xrightarrow{a} p'} \langle a \rangle X_{p'}) \wedge (\bigwedge_a [a] \bigvee_{p'.p \xrightarrow{a} p'} X_{p'})$$

As $p \xrightarrow{b} p_1$ we get that $q \models_{max} \langle b \rangle X_{p_1}$, which means that

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

or

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

as we wanted to prove.

b) Let $(p, q) \in R$ and $q \xrightarrow{b} q_1$, i.e.

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

As before we have

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

In particular we get

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

which means that

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

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

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

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

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

This proves the lemma. □

The following lemma completes the proof of our main theorem of this section.

Lemma 7.3 $([p_1]_\sim, \ldots, [p_n]_\sim) \sqsubseteq [\![D_K]\!]([p_1]_\sim, \ldots, [p_n]_\sim)$.

Proof: Let $q \in [p]_\sim$, where p is one of $p_1, \dots, p_n$. We have to show that

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

The proof can be divided into two parts:

$$\begin{array}{ll} 1) & q \in \displaystyle\bigcap_{a,p'.p \xrightarrow{a} p'} \langle\cdot a\cdot\rangle [p']_\sim \\ 2) & q \in \displaystyle\bigcap_a [\cdot a\cdot] \bigcup_{p'.p \xrightarrow{a} p'} [p']_\sim \end{array}$$

1) We recall that $q \sim p$. Assume that $p \xrightarrow{a} p'$. Then there is a q', where $q \xrightarrow{a} q'$ and $q' \sim p'$. We have therefore shown that

$$\forall a, p', p \xrightarrow{a} p'. \ (\exists q'. \ q \xrightarrow{a} q' \ \text{ and } \ q' \in [p']_\sim)$$

or that

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

2) Let $a \in \mathsf{Act}$ and $q \xrightarrow{a} q'$, we have to show that $q' \in \bigcup_{p'.p \xrightarrow{a} p'} [p']_\sim$. But as $p \sim q$, there exists a p', such that $p \xrightarrow{a} p'$ and $q' \sim p'$. The last statement means that $q' \in [p']_\sim$. We have therefore proven that

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

which is equivalent to

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

1) and 2) give that :

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

as we wanted to prove. □

The proof of Theorem 7.5 can now be expressed as the following lemma.

Lemma 7.4 For all $p \in \mathsf{Proc}$ we have that $[\![X_p]\!] = [p]_\sim$.

Proof: By Lemma 7.3 we get that

$$([p_1]_\sim, \dots, [p_n]_\sim) \leq ([\![X_{P_1}]\!], \dots, [\![X_{P_n}]\!])$$

which means that $[p]_\sim \subseteq [\![X_p]\!]$ for each $p \in \mathsf{Proc}$. Furthermore Lemma 7.2 gives that $[\![X_p]\!] \subseteq [p]_\sim$ for every $p \in \mathsf{Proc}$, which proves the statement of the lemma. □

7.7 Mixing maximal and minimal fixed points

The equational systems we have considered so far have only allowed us to express solutions as a pure maximal or a minimal solution. Our next question is whether we can extend our framework in such a way that it can treat *mixed solutions*, i.e., whether it is possible to decide the solution of, for instance,

$$X \overset{\max}{=} \langle a \rangle Y$$
$$Y \overset{\min}{=} \langle b \rangle X.$$

If we allow fixed points to be mixed completely freely we obtain *modal μ-calculus* [10], which was mentioned in Sect. 7.1. In this note we shall however not allow a full freedom in mixing fixed points in declarations but restrict the mixing to what we call *nested mutual recursion*. In this framework we are allowed to use solutions to purely maximal or purely minimal equational set in the definition of a new mutually recursive equational set.

Definition 7.2 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, for each $i \leq n$,

- $\mathcal{X}_i$ is a finite set of variables,
- $D_i : \mathcal{X}_i \longrightarrow \bigcup_{j \leq i} \mathcal{X}_i$,
- $m_i = \max$ 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 $m_i = max$ but a minimal block otherwise.

Note that by the theory described above, such a system has a unique solution, obtained by first solving the first block and then recursively proceeding with the others using the solution already obtained.

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

$$[\![F]\!] = \mathcal{O}_{F_X}([\![Y_1]\!], \ldots, [\![Y_k]\!]). \tag{41}$$

Exercise 7.10 *Prove the statement in (41).*

To be continued/updated/corrected tomorrow—if tomorrow ever comes, that is.

A Theory of Fixed Points and Tarski's Fixed Point Theorem

The aim of this appendix 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. We shall then use this theorem to give an alternative definition of strong bisimulation equivalence (also known as "strong equality"). This reformulation of the notion of strong bisimulation equivalence yields an algorithm for computing the largest strong bisimulation over finite labelled transition systems—i.e., labelled transition systems with only finitely many states and transitions.

The appendix is organized as follows. Section A.1 introduces partially ordered sets and complete lattices. We then proceed to state and prove Tarski's fixed point theorem (Section A.2). Finally, we show in Section A.3 how to define strong bisimulation equivalence using Tarski's fixed point theorem, and hint at the algorithm for computing strong bisimulation equivalence over finite labelled transition systems that results from this reformulation.

A.1 Complete Lattices

Definition A.1 [Partially Ordered Sets] A *partially ordered set (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 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 A.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 $\mathbb{N}$.
- $(A^*, \leq)$, 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 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:

    1. the length of s is smaller than that of t, or
    2. s and t have equal length, and there are strings $u, v, z \in A^*$ and letters $a, b \in A$ such that $s = uav$, $t = ubz$ and $a \leq b$.

- For each set S, the structure $(\mathcal{P}(S), \subseteq)$ is a poset, where $\mathcal{P}(S)$ stands for the set of all subsets of S.

Exercise A.1 *Which of the above posets is a totally ordered set?*

Definition A.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.

Exercise A.2 *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.*

Example A.2

- In the poset $(\mathbb{N}, \leq)$, all finite subsets of $\mathbb{N}$ have least upper bounds. 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 $(\mathcal{P}(S), \subseteq)$, *every* subset X of $\mathcal{P}(S)$ has a lub and a glb given by $\bigcup X$ and $\bigcap X$, respectively.

Exercise A.3

1. *Prove that the lub and the glb of a subset X of $\mathcal{P}(S)$ are indeed $\bigcup X$ and $\bigcap X$, respectively.*

2. *Give examples of subsets of A^* that have upper bounds in the poset $(A^*, \leq)$.*

Definition A.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 $\bot = \bigsqcap D$, and a top element $\top = \bigsqcup D$.

Exercise A.4 *Let $(D, \sqsubseteq)$ be a complete lattice. What are $\bigsqcup \emptyset$ and $\bigsqcap \emptyset$?*

Example A.3

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

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

$$\begin{tikzcd}
\infty \\
\vdots \\
2 \arrow[u] \\
1 \arrow[u] \\
0 \arrow[u]
\end{tikzcd}$$

- $(\mathcal{P}(S), \subseteq)$ is a complete lattice.

A.2 Tarski's Fixed Point Theorem

Definition A.4 [Monotonic Functions and Fixed Points] Let $(D, \sqsubseteq)$ be a poset. A function $f : D \to D$ is *monotonic* iff for all $d, d' \in D$, $d \sqsubseteq d'$ implies that $f(d) \sqsubseteq f(d')$.

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

The following important theorem is due to TARSKI [18], and was also independently proven by KNÄSTER.

Theorem A.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:

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

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

$$z_{\max} \sqsubseteq f(z_{\max}) \quad \text{and} \tag{42}$$
$$f(z_{\max}) \sqsubseteq z_{\max}. \tag{43}$$

   First of all, we shall show that (42) 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 (42).

   To prove that (43) holds, note that, from (42) 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 (42) and (43), 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. for every $d \in D$ that is a fixed point of f, $z_{\min} \sqsubseteq d$.

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

$$\begin{aligned} f(z_{\min}) &\sqsubseteq z_{\min} \quad \text{and} && (44)\\ z_{\min} &\sqsubseteq f(z_{\min}). && (45)\end{aligned}$$

Claim (44) can be shown following the proof for (42), and claim (45) can be shown following the proof for (43). 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). $\square$

Exercise A.5 *Reconsider equations (23)–(26) in the main body of the text.*

1. *Why doesn't Tarski's fixed point theorem apply to yield a solution to the first two of these equations?*
2. *Does equation (23) have a solution in the structure introduced in the second bullet of Example A.3?*
3. *Can you use Tarski's fixed point theorem to find the largest and least solutions of (26).*

Exercise A.6

1. *Prove that if $(D, \sqsubseteq)$ is a cpo and $f : D \to D$ is continuous (see [13, Page 103]), then the poset*

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

   which consists of the set of fixed points of f is 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*

$$\begin{tikzcd}
& \bullet & \\
& \bullet \arrow[u] & \\
\bullet \arrow[ur] & & \bullet \arrow[ul] \\
& \bullet \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

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 2 above.]*

5. *Let $(D, \sqsubseteq)$ be a complete lattice.*
   - *(a) Let $D \to_{mon} D$ be the set of monotonic functions from D to D and $\preceq$ be the relation defined on $D \to_{mon} D$ by*

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

     Show that $\preceq$ is a partial order on $D \to_{mon} D$.
   - *(b) Let $\bigvee$ and $\bigwedge$ be defined on $D \to_{mon} D$ by:*

     $$\text{If } \mathcal{F} \subseteq D \to_{mon} D \text{ then } \forall d \in D.(\bigvee \mathcal{F})(d) = \bigsqcup\{f(d) | f \in \mathcal{F}\}$$

     and

     $$\text{If } \mathcal{F} \subseteq D \to_{mon} D \text{ then } \forall d \in D.(\bigwedge \mathcal{F})(d) = \bigsqcap\{f(d) | f \in \mathcal{F}\}.$$

     Show that $(D \to_{mon} D, \preceq)$ is a complete lattice with $\bigvee$ and $\bigwedge$ as lub and glb.

The following theorem gives a characterization of the greatest and least fixed points for monotonic functions over *finite* complete lattices.

Theorem A.2 Let $(D, \sqsubseteq)$ be a *finite* complete lattice and let $f : D \to D$ be monotonic. Then the least fixed point for f is obtained as

$$z_{\min} \quad = \quad f^m(\bot)$$

for some m, where $f^0(\bot) = \bot$, and $f^{n+1}(\bot) = f(f^n(\bot))$. Furthermore the greatest fixed point for f is obtained as

$$z_{\max} \quad = \quad f^M(\top)$$

for some M, where $f^0(\top) = \top$, and $f^{n+1}(\top) = f(f^n(\top))$.

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 $\bot \sqsubseteq f(\bot) \sqsubseteq f(d) = d$. By repeating this reasoning $m-1$ more times we get that $f^m(\bot) \sqsubseteq d$. We can therefore conclude that $f^m(\bot)$ is the least fixed point for f.

The proof of the statement that characterizes largest fixed points is similar, and left as an exercise for the reader. $\Box$

A.3 Bisimulation as a Fixed Point

Let $(\mathsf{Proc}, \mathsf{Act}, \{ \xrightarrow{a} \mid a \in \mathsf{Act}\})$ be a labelled transition system. We recall that a relation $S \subseteq \mathsf{Proc} \times \mathsf{Proc}$ is a *strong bisimulation* [12] if the following holds:

If $(p, q) \in S$ then, for every $\alpha \in \mathsf{Act}$:

1. $p \xrightarrow{\alpha} p'$ implies $q \xrightarrow{\alpha} q'$ for some q' such that $(p', q') \in S$.
2. $q \xrightarrow{\alpha} q'$ implies $p \xrightarrow{\alpha} p'$ for some p' such that $(p', q') \in S$.

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

$$\sim \quad = \quad \bigcup \{S \in \mathcal{P}(\mathsf{Proc} \times \mathsf{Proc}) \mid S \text{ is a strong bisimulation}\}.$$

In what follows we shall describe the relation $\sim$ as a fixed point to a suitable monotonic function. First we note that $(\mathcal{P}(\mathsf{Proc} \times \mathsf{Proc}), \subseteq)$ (i.e., the set of binary relations over Proc ordered by set inclusion) is a complete lattice with $\bigcup$ and $\bigcap$ as the lub and glb. (Check this!) Next we define a function $\mathcal{F} : \mathcal{P}(\mathsf{Proc} \times \mathsf{Proc}) \longrightarrow \mathcal{P}(\mathsf{Proc} \times \mathsf{Proc})$ as follows.

$(p, q) \in \mathcal{F}(S)$ if and only if:

1. $p \xrightarrow{\alpha} p'$ implies $q \xrightarrow{\alpha} q'$ for some q' such that $(p', q') \in S$.
2. $q \xrightarrow{\alpha} q'$ implies $p \xrightarrow{\alpha} p'$ for some p' such that $(p', q') \in S$.

Then S is a bisimulation if and only if $S \subseteq \mathcal{F}(S)$ and consequently

$$\sim \ = \ \bigcup\{S \in \mathcal{P}(\mathsf{Proc} \times \mathsf{Proc}) \mid S \subseteq \mathcal{F}(S)\}.$$

We note that if $S, R \in \mathcal{P}(\mathsf{Proc} \times \mathsf{Proc})$ and $S \subseteq R$ then $\mathcal{F}(S) \subseteq \mathcal{F}(R)$ (check this!), i.e., $\mathcal{F}$ is monotonic over $(\mathcal{P}(\mathsf{Proc} \times \mathsf{Proc}), \subseteq)$. Therefore, as all the conditions for Tarski's Theorem are satisfied, we can conclude that $\sim$ is the greatest fixed point of $\mathcal{F}$. In particular, by Theorem A.2, if Proc is finite then $\sim$ is equal to $\mathcal{F}^M(\mathsf{Proc} \times \mathsf{Proc})$ for some $M \geq 0$. 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 $\mathcal{F}^i(\mathsf{Proc} \times \mathsf{Proc})$ for $i \geq 0$ until the sequence stabilizes.

Example A.4 Consider the labelled transition system described by the following equations:

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

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.

Exercise A.7

1. *Using the iterative algorithm described above, compute the largest strong bisimulation over the following transition system:*

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

2. *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?*

3. *Give a similar characterization for observational equivalence as a fixed point for a monotonic function.*

Acknowledgments These notes were partly written while Luca Aceto was on leave at Reykjavík University, and Anna Ingólfsdóttir was at deCODE Genetics. They thank these institutions for their hospitality and excellent working conditions.

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

We are most grateful to Jiri Srba for providing the figures in Chapter 4, and for using the notes in the installment of the Semantics and Verification course he taught in the spring of 2005 at Aalborg University. Jiri and the students who took the course offered valuable feedback on the text, and gave us a detailed list of errata.

Any remaining infelicity is solely our responsibility.

References

[1] J. BAETEN, *A brief history of process algebra*, Report CSR 04-02, Eindhoven University of Technology, 2004.

[2] J. BAETEN, J. BERGSTRA, AND J. W. KLOP, *On the consistency of Koomen's fair abstraction rule*, Theoretical Comput. Sci., 51 (1987), pp. 129–176.

[3] G. BOUDOL AND K. G. LARSEN, *Graphical versus logical specifications*, Theoretical Comput. Sci., 106 (1992), pp. 3–20.

[4] E. Clarke, E.A. Emerson, and A.P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Trans. Prog. Lang. Syst., 8 (1986), pp. 244–263.

[5] R. v. Glabbeek, *The linear time–branching time spectrum. I. The semantics of concrete, sequential processes*, in Handbook of Process Algebra, North-Holland, Amsterdam, 2001, pp. 3–99.

[6] D. Harel and A. Pnueli, *On the development of reactive systems*, in Logics and models of concurrent systems (La Colle-sur-Loup, 1984), vol. 13 of NATO Adv. Sci. Inst. Ser. F Comput. Systems Sci., Springer-Verlag, Berlin, 1985, pp. 477–498.

[7] M. Hennessy and R. Milner, *Algebraic laws for nondeterminism and concurrency*, J. Assoc. Comput. Mach., 32 (1985), pp. 137–161.

[8] C. Hoare, *Communicating sequential processes*, Comm. ACM, 21 (1978), pp. 666–677.

[9] R. Keller, *Formal verification of parallel programs*, Comm. ACM, 19 (1976), pp. 371–384.

[10] D. Kozen, *Results on the propositional mu-calculus*, Theoretical Comput. Sci., 27 (1983), pp. 333–354.

[11] K. G. Larsen, *Proof systems for satisfiability in Hennessy–Milner logic with recursion*, Theoretical Comput. Sci., 72 (1990), pp. 265–288.

[12] R. Milner, *Communication and Concurrency*, Prentice-Hall International, Englewood Cliffs, 1989.

[13] H. Nielson and F. Nielson, *Semantics with Applications: A Formal Introduction*, Wiley Professional Computing, John Wiley & Sons, Chichester, England, 1992.

[14] D. Park, *Concurrency and automata on infinite sequences*, in 5^{th} GI Conference, Karlsruhe, Germany, P. Deussen, ed., vol. 104 of Lecture Notes in Computer Science, Springer-Verlag, 1981, pp. 167–183.

[15] G. Plotkin, *A structural approach to operational semantics*, Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[16] ——, *The origins of structural operational semantics*, Journal of Logic and Algebraic Programming, (2004). To appear. The paper is available from http://www.dcs.ed.ac.uk/home/gdp/publications/.

[17] A. PNUELI, *The temporal logic of programs*, in Proceedings *18th* Annual Symposium on Foundations of Computer Science, IEEE, 1977, pp. 46–57.

[18] A. TARSKI, *A lattice-theoretical fixpoint theorem and its applications*, Pacific Journal of Mathematics, 5 (1955), pp. 285–309.