The application of a resource logic to the non-temporal analysis of processes acting on resources

Jonathan Hayman
HP Laboratories Bristol
HPL-2003-194
October 2nd , 2003*

E-mail: jmh00@doc.ic.ac.uk

business process analysis, bunched implications, concurrency, demos

Many systems exist that can be modelled as processes interacting through shared resources -people, factories and computer systems to name but a few. Building upon a logic capable of reasoning about *static* resource models, the Logic of Bunched Implications, we tentatively define the semantics of a logic capable of reasoning about *dynamic* resource models. The logic shall not consider the influence of timing on process interaction. We give a brief overview of other process logics- Hennessy-Milner logic, Computation Tree Logic and modal- μ - which indicate how we may automate the system. We illustrate how our dynamic resource logic may be applied to the analysis of Petri nets. In order to conduct an analysis of processes, we must have a formal representation for them: we choose Demos, and present its salient characteristics. We give a formal definition of its non-time semantics, and prove that this is, in some sense, correct. Following consideration of how we apply our process logic to Demos models, we propose a method for reducing the state-space of models generated by considering a (partially) synchronous execution. We show that such a system at least correctly detects deadlock.

* Internal Accession Date Only
Approved for External Publication

Acknowledgements

Firstly, I would like to thank the Enterprise Solutions Department, and HP Labs as a whole, for providing such a pleasant working environment.

Thanks also to David Pym for a great deal of useful discussion on the logical side of this work; his contribution to this cannot be overestimated.

Last, but by no means least, I would like to thank Chris Tofts for proposing this project, guiding me into the subject area, providing a wealth of knowledge and, generally, making the time I spent producing this so enjoyable.

Contents

List of Figures

List of Tables

Introduction

When we design complex systems, we should ask questions of them before they are implemented — will it run fast enough; would it be more efficient for a human to do a particular sub-task; *etc.* To answer such questions will require automation.

In the general case, then, how are we to automate such analyses of systems? Perhaps the most obvious manner (at least to the computer scientist with a disposition towards the practical) is to simulate the design, probably observing several 'runs' of the simulation, thereby repeating an experiment. Though, in principle, this approach generalises to all types of system by Turing completeness (and normally requires little background knowledge other than an understanding of the system and the ability to program), in some ways it is grossly unsatisfactory. What if, for example, we are simulating a network and no run of the simulation has a very large file being transmitted (or any other *rare-event*)? Clearly, our results may be overly optimistic, which could lead, for example, to us breaching the terms of a service-level agreement. Conversely, what if our simulation has a few too many rare-events (large files)? This time, our results may be overly pessimistic, resulting in the over-specification of the network at financial cost to us. One answer to this is to run the simulation for longer, so as to increase the probability that rare-events are encountered. Of course, though, given a finite amount of time, our results will always be approximate.

An alternative approach is to construct an analytical model of the system to be analysed. Returning to the network example, a model would probably result in a set of equations to solve in order to obtain some measurements of the system's performance (*metrics*). The advantage of this is that one analysis gives exact results, taking rare-events into fair consideration, and, very often, can be obtained much faster than by simulation. The flip-side is that, generally, we are only able to do this for particular classes of system. For example, the simplest analyses of queuing networks (*e.g.,* computer networks that transmit packets) assume a (negative) exponential probability distribution of service time and arrival rate; that is,

$$P(\text{Service time} \leqslant t) \propto e^{-\mu t}.$$

Fortuitously, many systems approximate very closely the assumptions made to build such models. Proving that the assumptions behind such models are met (for example, that the time between packet arrivals follows a negative exponential distribution), though, is somewhat more complicated and subjective;

simulation has the same problem, manifested in the problem of how to choose the distributions or sample data that we use to parameterise the simulation.

We must now move on to decide upon the type of system that we are to model. Naturally, we cannot hope to provide a mathematical model for *all* types of system, able to answer *all* types of question! We shall consider a notion again familiar to computer scientists, that of *processes*, and give the definition found in [OED73]:

> **Process** ... A continous and regular action or succession of actions, taking place or carried on in a definite manner; a continuous (natural or artificial) operation or series of operations...

Central to our analysis of processes, of course, shall be the environment in which they operate. In particular, a process may be acting in an environment with other processes; we shall say that the processes are acting in *parallel* or *concurrently*. It has long been known — before, even, the first parallel computer was built — that it is difficult to reason about parallel processes, as noted in the very first item to appear in the *Communications of the ACM* [Per58]. Our work to provide some degree of automation should, therefore, be of some value.

In order to further restrict the the scope of our work, we shall analyse the behaviour of processes when they interact through the acquisition and release of resources. To borrow Robin Milner's famous jobshop example [Mil89], consider two workers, called *jobbers*, a hammer, a mallet and a stream of jobs to be done. Considering the workers to be processes and the hammer and mallet to be resources[1], we might want to ask all sorts of questions. For example, would we see a gain in productivity if we gave each jobber their own hammer and mallet?

A different class of question relates to classical concurrency: do the jobbers deadlock (not be able to take any action without having terminated)? Let us suppose that the jobbers are simple folk; if they work in isolation, they will work happily until the end of their day and will not deadlock. They are also tidy folk, leaving the resources as they found them at the start of the day (so they return the tools to their correct place). If the jobbers work at different times of the day, we may conclude that they will not deadlock. Should, though, our analysis of the activity of the jobbers take into account the times that they actually work? C.A.R. Hoare [Hoa85] argues not: our analysis would not take into account the outcome of a jobber arriving at work late and then working late (a not inconceivable scenario!). Thus, perhaps, we should not argue that the system never deadlocks.

To give a more concrete example and further justification, consider the communication network of Figure 1(a). Nodes represent intelligence agencies in the UK and the US, and arcs represent dedicated, secure communication channels (perhaps based upon quantum encryption) that can only be 'owned' by one node at a time; in a sense, they are *resources*. The hub in the centre of the diagram is capable of serving many lines at once and represents no limitation on the system. It is important to abstract away the irrelevant when modelling, so we

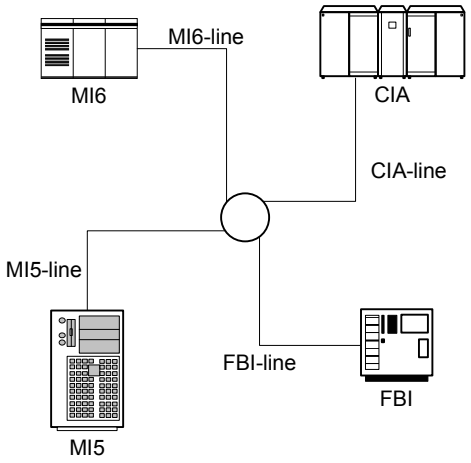
[1] We could also consider the hammer and mallet to be (very simple) processes, which alludes to the consideration of resources as processes under certain circumstances

shall ignore its behaviour. Suppose that MI6 has a process that starts every day at 12:43GMT which sends some data from MI5 to the FBI. To do this, it claims ownership of the MI6 line, followed by the MI5 line, then the FBI line. Once a process (in this case, an intelligence agency) starts to get a line, no other process can start to get the same line. Between each acquisition, a 2-minute pause occurs to allow some handshaking protocol to occur. Similarly, the CIA has a process that starts every day at 12:40GMT which sends information from the FBI to MI5: in doing this, it claims the CIA line, the FBI line and then the MI5 line. These processes are rpesented in in Figure 1(b). What normally happens is that the CIA process has control of the MI5 line before MI6 tries to acquire it. Consequently, the MI6 process waits until the MI5 line is released thereby avoiding deadlock. This execution is illustrated in 2(a).

Now, suppose that the CIA process takes an extra two minutes to gain control over the FBI line — possibly because of interference during the handshaking period. At 12:46, the CIA process will try to gain control over the MI5 line but will fail — it is now owned by MI6 — and so will wait for it to become available. At 12:47, the MI6 process will try to acquire the FBI line but will also fail — it is owned by the CIA — and so will wait. Both the CIA and the MI6 processes, then, will wait indefinitely: they are in deadlock. This execution is shown in 2(b).

Unfortunately, if we had taken time into account when deciding upon deadlock freedom, from our model we would never have known that deadlock could occur, and would fail to see the inherent flaw in the system. This could be dangerous for MI6 (and, similarly, the CIA) — MI5, which has responsibility for counterespionage, might notice that its secure link with the outside world was never available and investigate why this was so.

We have considered a *discrete-event* system — we do not consider actions on resources to happen over a period of time. Further, our consideration shall be of *discrete-resource* systems. So, rather than having 2.4 litres of oil that we can partition to allocate to processes as we wish, we shall consider n units of oil where $n \in \mathbb{N}$ (n is a non-negative integer). However, our theoretical approach could well be extended to a notion of continuous resource. Thus the system developed herein shall operate on *discrete-event*, *discrete-resource* processes. Not coincidentally, we shall consider a simulation language, Demos, that has both these properties.

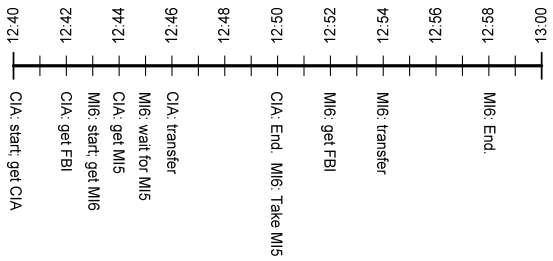


(a) Communication network

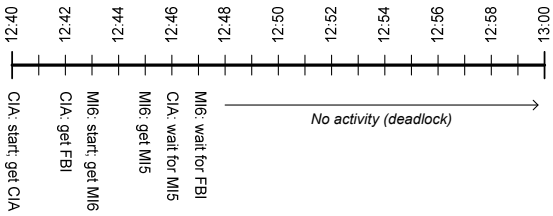
MI6	**CIA**
Get MI6 line	Get CIA line
Get MI5 line	Get FBI line
Get FBI line	Get MI5 line
(transfer data)	(transfer data)
Release all	Release all

(b) Communication processes

Figure 1: Intelligence agency network and process design



(a) Normal execution



(b) Deadlocking execution

Figure 2: Intelligence agency process execution

Chapter 1

Logical Framework

1.1 The (static) Logic of Bunched Implications

The Logic of Bunched Implications [OP99], **BI**, allows us to reason about resources. Given a particular collection of resources (a *model*), we ask whether propositional formulæ hold.

Technically, **BI** is a logic based around *bunches*, tree-like structures of propositions (and, in the predicate case, constants). It seeks to combine the additive semantics of intuitionistic logic (giving standard meaning to the operators $\wedge, \vee, \rightarrow$) with the multiplicative semantics of Girard's Linear Logic [Gir87] (giving the operators $*, -\!\!*$). The multiplicative connectives, essentially, allow us to make the kind of assertions that we would like to make about systems with resources. When we write formulæ in **BI**, the multiplicatives shall bind more strongly than the additives; conjunction shall bind more strongly than disjunction, which in turn binds more strongly than implication, and negation binds strongest of all. All binary operators are taken to be right-associative (this is only of importance to the implications — the other binary operators are commutative and associative). For example,

$$\phi * \psi \wedge \psi -\!\!* \gamma \equiv (\phi * \psi) \wedge (\psi -\!\!* \gamma)$$

Logics are defined in two parts, syntax and semantics. The syntax of the logic is just a way to write down sentences that relate to the semantics. Attached to the syntax may be a deduction system, taking sentences written according in the language defined by the syntax to derive other syntactically correct sentences. The semantics of the logic defines what the formulæ written in the syntax actually *mean*. It may well be the case (as it is with **BI**) that we can define several semantics for the logic whilst preserving the 'correctness' of the deduction system. We choose to consider one of the simplest (and the one that can have the simplest notion of resource): the Kripke semantics.

The Kripke semantics of **BI** allows us to define the semantics through a mathematical structure called a *monoid*,

$$\mathcal{M} = \langle M, e, \circ, \sqsubseteq \rangle,$$

where M represents a domain for our monoid, e represents the identity element under composition, $\circ$ is commutative and represents the composition of elements of M, and $\sqsubseteq$ is a pre-order on M. The semantics of **BI** based upon this monoid is given in Table 1.1.

Let $m \in M$, p be an atomic proposition, φ, ψ range over propositions and $\|\mathrm{p}\| \subseteq M$ be the set of states in which the proposition p holds. Then the Kripke semantics is given by:

$m \models \mathrm{p}$	iff	$m \in \|\mathrm{p}\|$
$m \models \varphi \wedge \psi$	iff	$m \models \varphi$ and $m \models \psi$
$m \models \varphi \vee \psi$	iff	$m \models \varphi$ or $m \models \psi$
$m \models \varphi \to \psi$	iff	for all $n \sqsubseteq m$, $\quad n \models \varphi$ implies $n \models \psi$
$m \models \varphi * \psi$	iff	for some $n, n' \in M \quad m \sqsubseteq n \circ n'$ and $n \models \varphi$ and $n' \models \psi$
$m \models \varphi \mathrel{-\!\!*} \psi$	iff	for all $n \in M \quad n \models \varphi$ implies $m \circ n \models \psi$

Table 1.1: Kripke semantics of (static) **BI**

There are two types of operator in the logic: additive and multiplicative. The additive operators, $\wedge$ and $\to$, are just as in intuitionistic propositional logic: $\wedge$ is used to indicate that more than one formula holds in the state, and $\to$ is implication.

More interesting are the multiplicative operators. In $\varphi * \psi$, $*$ is used to indicate that we can partition (using $\circ$) our current state; one part satisfies φ and the other ψ. $\varphi \mathrel{-\!\!*} \psi$ indicates that if we add all that we need to satisfy φ to our current state, ψ will hold.

It may be helpful to consider a monoid based upon a notion of *resource*. If we let R denote a multiset of discrete[1] resources, $\mathcal{P}(R)$ denote the set of all sub-multisets of R and multiset union be $\uplus$, then we can define the monoid

$$\mathcal{M} = \langle \mathcal{P}(R), \{\!|\}, \uplus, = \rangle.$$

The consequent intuitive meaning of $m \models \varphi * \psi$ is that we can split the resource multiset m in two, one part making φ true and the other making ψ true. $m \models \varphi \mathrel{-\!\!*} \psi$ intuitively means that given the resources to make φ true, if we add them to what we have in m, ψ will be true.

Example (Discrete resources) For each resource $r \in R$, define $\|r\| = \{\!|r|\!\}$; so, for every resource r, r holds iff r is the only available resource. Also define a proposition $move_{bolt}$, holding with $\{\!|wrench, mallet|\!\}$; it signifies that we can move a bolt iff we have a hammer and a mallet available.

[1] As we have said, not, for example, n metres of string that can be split into m and m' metres of string where $m + m' = n$ for arbitrary $m, m' \in \mathbb{R}$.

The Kripke discrete resource semantics gives:

$$\begin{array}{rcl}
hammer & \models & hammer \\
hammer, mallet & \not\models & hammer \\
hammer, mallet & \models & hammer * mallet \\
hammer, mallet & \models & hammer * \top \\
hammer, mallet & \models & hammer * \top \wedge hammer * \top \\
hammer, mallet & \not\models & hammer * hammer * \top \\
hammer, mallet & \not\models & (hammer * \top) * (hammer * \top) \\
hammer, mallet & \models & wrench \mathrel{-\!\!*} move_{bolt} * \top \\
hammer, mallet & \models & (wrench \mathrel{-\!\!*} move_{bolt}) * \top \\
hammer, mallet & \models & move_{bolt} \mathrel{-\!\!*} mallet * mallet * \top \\
hammer, mallet & \not\models & mallet \to mallet * mallet * \top
\end{array}$$

The monograph [Pym02] provides a substantially more detailed, technical and comprehensive account of **BI**. Though it is possible to encode action (Petri nets are discussed in [POY02, Pym02]) into **BI** by attaching certain semantics, the resource interpretation above does not have a notion of how resource evolves with the action of processes; we are only able to make *static* judgements. The necessary tools to describe action, transition systems and modal logics, are introduced below.

Notice that we have defined neither negation nor falsity in this system. We could have included falsity, $\bot$, but this makes the associated natural deduction system incomplete. With $\bot$, we could define intuitionistic negation: $\neg\varphi = \varphi \to \bot$. We obtain a classical logic (thence negation) if $\sqsubseteq$ is equality.

1.2 Transition systems

Since the dissemination of Gordon Plotkin's notes on Structural Operational Semantics, [Plo81], as detailed in [Plo03], it has become commonplace to describe the activity of programming languages as a one-step relation between states defined *structurally*. That is, we give an inductive definition of the effect of each command on the state of the system. This is normally presented in the proof-tree natural deduction style:

$$\text{(RULE-NAME)} : \frac{\textsc{Premises}}{\textsc{Conclusions}}\,(Condition)$$

This asserts that if we can derive by the set of rules all the premises then the conclusions will hold, provinding the side-consdition is met. Thus, as we have defined the transition relation structually, we can prove properties of the relation by structural induction.

Consider, as an example, an elementary imperative programming language. The one-step transition relation shall be written $\Longrightarrow$. A transition system shall represent state as a function from variables to values, with $s[x \mapsto c]$ representing the *system* state s but with x having the value c. If we let $\mathcal{A}[\![e]\!]\,s$ represent

the result of evaluating the arithmetic expression e in state s, we could define assignment as follows:

$$(\text{ASS}) : \frac{}{\langle x := e, s\rangle \Longrightarrow s[x \mapsto c]} \, (\mathcal{A}\,[\![e]\!]\,s = c)$$

This expresses that the program `x := e` executing in an arbitrary state s terminates after one step leaving the same state other than x is now $c = \mathcal{A}\,[\![e]\!]\,s$, the result of evaluating e in s.

Together with a series of other rules for the various programming language constructs, for example the sequential composition of commands,

$$\left(\text{COMP}^{\text{I}}\right) : \frac{\langle s, P_1\rangle \Longrightarrow \langle s', P_1'\rangle}{\langle s, P_1; P_2\rangle \Longrightarrow \langle s, P_1'; P_2\rangle}$$

this would define the transition relation $\Longrightarrow$.

1.2.1 Labelled transition systems

It is sometimes useful to label the the transitions between states. As noted in [Pit02], when we define the semantics of languages we usually have *reduction* transitions (which describe real action within the system) and *simplification* actions (which describe, for example, modifications to syntax). Though it may be considered desirable to define the transition relation solely in terms of reduction transitions, using a number of auxilliary relations to describe simplification, labelling allows us to make explicit the rôle of each transition. Furthermore, labelling shall allow finer queries to be made in the modal logic (as shall be indicated later) though non-labelled systems are commonplace *e.g.,* [MP83].

As we have already stated, when we come to desribe Demos, we shall be considering the interaction between processes working on resources. In particular, we shall consider the *available* resources within the system; that is, those that have not been acquired by any process. This shall be described by a multiset; we let $\mathbf{R}, \mathbf{R}', \ldots$ range over such multisets of resource, and $P, P', Q, \ldots$ range over the code remaining to be executed. We shall call $\mathbf{R} \triangleleft [\![P]\!]$ a *state* of the transition system. Transitions are of the form $\mathbf{R} \triangleleft [\![P]\!] \xrightarrow{\alpha} \mathbf{R}' \triangleleft [\![P']\!]$. We read this as:

> Initially, all the resources in $\mathbf{R}$ are available and we have an active process P. After an action α, the resources in $\mathbf{R}'$ are available and the active process is P'.

We distinguish two classes of transition: σ-transitions and τ-transitions. σ-transitions indicate action, and τ-transitions indicate silent or invisible actions, related to the simplification relations described above, that perform syntactic operations to expand loops and skip by commands that are not considered.

An alternative would have been to define an equivalence relation, based upon syntax, to indicate that we consider pairs of states to be the same. For example, given that our consideration is of action rather than time, we would consider $\mathbf{R} \triangleleft [\![\texttt{hold}(t); \texttt{putR}(r, 1)]\!]$ and $\mathbf{R} \triangleleft [\![\texttt{putR}(r, 1)]\!]$ to be equivalent. Such an equivalence relation, though, would decrease the tractability of search in a language such

as Prolog: the arbitrary introduction of hold commands, `do` 0 loops *etc.* (the symmetry of the equivalence relation being the key problem) would be allowed, so we choose to define the τ relation, which is not symmetric.

An excellent account of labelled transition systems is given in [Pąc89], along with a method for analyzing assertions made of programs.

1.3 Modal and Temporal Logics

The Free Online Dictinary of Computing [FOL] defines a modal logic to be: *An extension of propositional calculus [propositional logic] with operators that express various "modes" of truth.* So, rather than just being able to express "φ is true", we might be able to express that "φ is possibly true," "φ is necessarily true," "φ is true at some point in the future," "φ is true in the next state," and so on.

Formally, adopting the Kripke semantics of modal logics, we define a set of states (or worlds), W, and a relation R between these states. This defines a *Kripke frame* (essentially, just a set of states with transitions between them). We then define an evaluation function for propositional atoms in each state, $h(s,p) : \{\top, \bot\}$ (an alternative notation to this is $s \in \|p\| \iff h(s,p)$). A modal logic allows us to write formulæ over the transitions between worlds in the frame. So, for example, we can write "given that I am in world s, in every reachable world, (something) holds".

Regarding worlds as program states, the relation between worlds as time (or program execution), we obtain an important type of modal logic: temporal logic. We see that temporal logics allow us to reason about the activity of processes. In their simplest form, they allow us to make assertions about the state of a process after one execution step; this is easily generalised to multiple-steps, as shall be outlined below.

1.3.1 Hennessy-Milner Logic

Here we shall give an account of Hennessy-Milner logic, which was introduced in [HM80, HM85]. Our notation shall follow that presented later in [Mil89]; in §1.3.2 we shall generalise the following definition to match the system of [Mil89].

Firstly, in order to create a well-founded inductive definition of modal formulæ, we shall need to define when an atomic proposition is satisfied. The definition is straightforward: a state s satisfies an atomic proposition p, written $s \models p$, iff $s \in \|p\|$, where $\|p\|$ is the set of states in which the proposition holds. Similarly, the definitions of conjunction and disjunction follow the additive cases of Table 1.1:

$$s \models p \wedge q \text{ iff } s \models p \text{ and } s \models q$$

and

$$s \models p \vee q \text{ iff } s \models p \text{ or } s \models q$$

Suppose that a process is in a state s. If, by some action in a set A, a transition from s can be derived to a state in which a (possibly temporal) formula φ

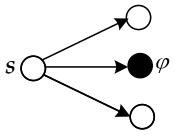
is true, we write $s \models \langle A \rangle \varphi$ (where $\langle A \rangle$ is pronounced "diamond-A"). If all transitions derivable from s by actions in A lead to states in which φ holds, we write $s \models [A]\varphi$ (pronounced "box-A"). In a more formal notation:

Definition: (Simple modalities)

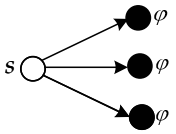
$$\begin{array}{rcl} s \models \langle A \rangle \varphi & \text{iff} & \exists \alpha, s'. \quad \alpha \in A \text{ and } s \xrightarrow{\alpha} s' \text{ and } s' \models \varphi \\ s \models [A]\varphi & \text{iff} & \forall \alpha, s'. \quad \alpha \in A \text{ and } s \xrightarrow{\alpha} s' \text{ implies } s' \models \varphi \end{array}$$

Pictorially, these express the following:

$s \models \langle - \rangle \varphi$:



$s \models [-]\varphi$:



Assuming the existence of negation within our logic, it is not hard to see that the following equivalence exists between the modalities:

$$\langle A \rangle \varphi \iff \neg [A] \neg \varphi$$

We may choose, then, only to define one of the above modalities within the basic logic, and define the other to be a 'derived' modality.

In the sequel it shall be convenient to use an abbreviated syntax to define the action set. Assuming that the set of all action labels is $\mathcal{A}$, we shall write:

$$\begin{array}{rcl} [a, a', \ldots] & \text{for} & [\{a, a', \ldots\}]; \\ [-A] & \text{for} & [\mathcal{A} \setminus A]; \\ [-a, a', \ldots] & \text{for} & [\mathcal{A} \setminus \{a, a', \ldots\}]; \\ [\,] & \text{for} & [\emptyset]. \end{array}$$

[2]

In particular, we shall make use of $[-]$, which abbreviates $[\mathcal{A}]$. The obvious symmetric abbreviations shall be made for the diamond modality as well.

1.3.2 General Hennessy-Milner Logic

Though the inductive definition above allows formulæ to have nested modalities, so far we can only make statements about processes after a finite number of steps. Furthermore, to assert that a program terminates before, say, 1000 steps would require a very long sentence.

[2] There is no fixed convention for this abbreviation. Its use shall be avoided wherever possible.

With this in mind, suppose that we generalise conjuction (or its dual, disjunction) to require validity of all conjuncts indexed by a set I:

$$s \models \bigwedge_{i \in I} \varphi_i \quad \text{iff} \quad \forall i \in I.\ \ s \models \varphi_i$$

Given that we can express, for arbitrary n, that a program in state s terminates (or at least stops running) after exactly n transitions from some set A as

$$s \models \langle A \rangle^n [A] \bot,$$

we may express the above property (termination before 1000 steps) succinctly as

$$s \models \bigvee_{0 \leqslant i \leqslant 1000} \langle A \rangle^i [A] \bot.$$

Notice that we have abbreviated $i \in \{0, \ldots, 1000\}$ by $0 \leqslant i \leqslant 1000$.

If we allow infinitary conjunction ($i \in \mathbb{N} \cup \{\omega\}$, where ω is the first ordinal, the least number greater than every number in $\mathbb{N}$, otherwise written '$\mathbb{N}$' itself; we shall interchangably write $\mathbb{N}_\omega$ for $\mathbb{N} \cup \{\omega\}$), we may express deadlock freedom of processes by the simple formula:

$$\mathfrak{D}_f^{hml} \quad \stackrel{\text{def}}{\Longleftrightarrow} \quad \neg \bigvee_{i \in \mathbb{N} \cup \{\omega\}} \langle A \rangle^i [A] \bot$$

which is logically equivalent to

$$\bigwedge_{i \in \mathbb{N} \cup \{\omega\}} [A]^i \langle A \rangle \top$$

The reader may gain some intuition as to the meaning of this sentence by 'unfolding' the conjunction. It expresses that the process is not stuck in this state, that every state reachable in one transition is not stuck (and so we can perform another transition), that every state reachable in two transitions is not stuck (and so we can perform another transition), ...

1.3.3 CTL

The Computation Tree Logic of [CES86] allows some temporal properties to be expressed more simply than the Hennessy-Milner logic described above does, as well as being more amenable to model checking. The definition of CTL in [CES86] requires that the transition relation is total, *i.e.,* every state has a next state. We regard this as unreasonable for our purposes, so we shall *not* require this.

Letting AP be the set of all atomic propositions, the following two rules inductively define the class of CTL formulæ:

1. Every proposition $\mathrm{p} \in AP$ is a CTL formula.
2. If φ and ψ are CTL formulæ, so are $\neg\varphi, \varphi \wedge \psi, \langle - \rangle \varphi, [-]\varphi, \forall[\varphi \mathbf{U} \psi]$ and $\exists[\varphi \mathbf{U} \psi]$.

The semantics attached to the formulæ composed of first-order propositional logic operators is conventional, as for the box and diamond modalities. $\forall[\varphi \mathbf{U} \psi]$ intuitively means: *whatever path is chosen from the given state, φ will hold until ψ holds (and ψ actually does hold somewhere).* Similarly, $\exists[\varphi \mathbf{U} \psi]$ intuitively means: *a path can be chosen where φ holds until ψ holds (and ψ actually does hold somewhere).*

A *path* is an possibly infinite sequence of states, $[s_0, s_1, \ldots]$ where $\forall i \geqslant 0[\ s_i \xrightarrow{\sigma_i} s_{i+1}\ ]$ and $\xrightarrow{\sigma}$ is the transition relation of the labelled transition system. If the path is finite, there must be no derivable transition from the last state.[3] Using this definition, Table 1.3.3 defines the semantics of CTL.

[CES86] gives a relatively straightforward algorithm for model-checking CTL formulæ where the state space of the given process is finite.

$s_0 \models \mathrm{p}$	iff	$s_0 \in \|\mathrm{p}\|$
$s_0 \models \neg\varphi$	iff	$s_0 \not\models \varphi$
$s_0 \models \varphi \wedge \psi$	iff	$s_0 \models \varphi$ and $s_0 \models \psi$
$s_0 \models \varphi \vee \psi$	iff	$s_0 \models \varphi$ or $s_0 \models \psi$
$s_0 \models \langle - \rangle \varphi$	iff	for some state s' and action $\alpha \in \mathcal{A}$, $s_0 \xrightarrow{\alpha} s'$ and $s' \models \varphi$.
$s_0 \models [-]\varphi$	iff	for all states s' and actions α, $s_0 \xrightarrow{\alpha} s'$ implies $s' \models \varphi$
$s_0 \models \forall[\varphi \mathbf{U} \psi]$	iff	for all paths $[s_0, s_1, \ldots]$, $\exists i \geqslant 0[\ s_i \models \psi \wedge \forall j[0 \leqslant j < i \implies s_j \models \varphi]\ ]$
$s_0 \models \exists[\varphi \mathbf{U} \psi]$	iff	for some path $[s_0, s_1, \ldots]$, $\exists i \geqslant 0[\ s_i \models \psi \wedge \forall j[0 \leqslant j < i \implies s_j \models \varphi]\ ]$

Table 1.2: Semantics of CTL formulæ

Deadlock freedom may be represented in CTL as:

$$\mathfrak{D}_f^{CTL} \iff^{\text{def}} \neg\exists[\top \mathbf{U} \neg\langle - \rangle \bot]$$

1.3.4 CTL$^-$

CTL provides a relatively clear language for expressing how propositions hold through the evolution of processes. In many cases, however, it is sufficient to use a subclass of CTL formulæ called CTL$^-$.

[3] As we do not require that the transition relation be total, our definition of path differs from that in [CES86] in that we allow finite sequences.

CTL$^-$ does not have the $\exists[-\mathbf{U}-]$ or $\forall[-\mathbf{U}-]$ modalities, but does have $\exists\mathbf{F}, \forall\mathbf{F}, \exists\mathbf{G}$ and $\forall\mathbf{G}$. These are defined as follows (through the semantics of the CTL modalities):

- $s \models \exists\mathbf{F}\varphi \iff^{\text{def}} s \models \exists[\top\mathbf{U}\varphi]$
  This means that there is a path in which φ holds in some future state — φ *potentially* holds.
- $s \models \forall\mathbf{F}\varphi \iff^{\text{def}} s \models \forall[\top\mathbf{U}\varphi]$
  This means that along every path from s, there is a state in which φ holds — φ is *inevitable*.
- $s \models \exists\mathbf{G}\varphi \iff^{\text{def}} s \models \neg\forall[\top\mathbf{U}\neg\varphi]$
  This means that there is a path from s where φ holds in every state.
- $s \models \forall\mathbf{G}\varphi \iff^{\text{def}} s \models \neg\exists[\top\mathbf{U}\neg\varphi]$
  This means that for every path from s, φ holds in every state — φ is *globally* true.

$\forall\mathbf{F}$ is frequently written `AF`; $\exists\mathbf{G}$ is frequently written `EG`; *etc.*

We can represent deadlock freedom in CTL$^-$as:

$$\mathfrak{D}_f^{CTL^-} \iff^{\text{def}} \forall\mathbf{G}\langle-\rangle\top$$

1.3.5 Modal-μ

The modal-μ calculus (a good introduction may be found in [BS01]; we summarize the essentials here) is a logic where formulæ may be defined recursively. Deadlock freedom, for example, may be recursively defined with respect to a set of actions A as:

$$\mathfrak{D}_f^{rec} = {}^4\langle A\rangle\top \wedge [A]\mathfrak{D}_f^{rec}$$

This states that a state is deadlock free if it is possible to go to another state and any state reached is also deadlock free.

As before, we define the set of states in which a proposition φ holds as $\|\varphi\| \in \mathcal{P}(S)$, where S is the set of all states and $\mathcal{P}(S)$ is the powerset of the set of states, $\mathcal{P}(X) = \{Y | Y \subseteq X\}$. In the above example, we have used $\mathfrak{D}_f^{rec}$ as a *variable* over logical forumlae. If a formula has a free variable Z, we indicate this by writing it as $\rho(Z)$. This may be regarded as a function $\rho(Z) : \mathcal{P}S \to \mathcal{P}S$. Now, by the Knaster-Tarski theorem, the function has least and greatest *fixed points*; that is, there are least and greatest sets that are solutions to the equation $Z = \rho(Z)$. These shall be denoted $\mu Z.\rho(Z)$ and $\nu Z.\rho(Z)$, respectively. For reference, the semantics of the fixpoint operators μ and ν is given by:

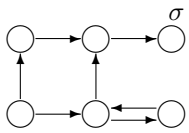
$$\|\mu Z.\rho(Z)\| = \bigcap\{S | S \supseteq \|\rho(S)\|\}$$

and

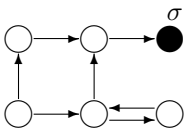
$$\|\nu Z.\rho(Z)\| = \bigcup\{S | S \subseteq \|\rho(S)\|\}$$

[4] We shall see later that we should have used $\subseteq$ rather than $=$

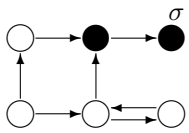
One of the difficulties of writing formulæ in modal-μ is deciding which fixpoint operator to use, and, for the reader, understanding the meaning of this choice. We may gain some intuition if we consider the way in which the fixed points may be constructed. To construct the least fixed point of a formula $X = \sigma \vee [-]X$ for the state-space illustrated through Figure 1.1, we would start with the formula holding nowhere; that is, $\|X\| = \emptyset$. We shall call this the zeroth-approximant, written X^0 (Fig. 1.1(a)). Now, one application of the formula later, we have $X^1 = \|\sigma\| \cup$ the set of states with no transitions from them; this is precisely the set of states where σ holds or every transition from the state leads to a state in $\emptyset$ (Fig. 1.1(b)). This series of approximations continues until a fixed point is reached (Fig. 1.1(d)).



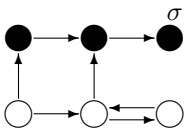
(a) Zeroth approximant



(b) First approximant



(c) Second approximant



(d) Least upper bound

Figure 1.1: Approximants of $X = \sigma \vee [-]X$ to form $\mu X.\sigma \vee [-]X$

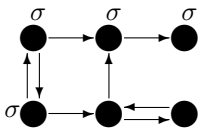
Similarly, the greatest fixed point may be found by iterating from the set of all states, taking the intersection of the iterates. An example of this is given in Figure 1.2.

As an alternative to this interpretation, the authors of [BS01] propose the following maxim:

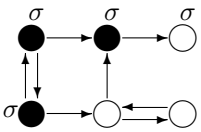
> ν means looping; μ means finite looping.

Consider the formula $Z = \sigma \vee \langle - \rangle Z$, meaning that a state either makes σ hold or some such state is reachable. The maxim gives different meaning to the formulæ $\mu Z.\sigma \vee \langle - \rangle Z$ and $\nu Z.\sigma \vee \langle - \rangle Z$; the former means that there is a finite-length path to a state where σ holds, whereas the latter means that there is a possibly infinite length path to a state where σ holds. Notice that this does not imply that σ holds in any particular state $s \in \mathcal{S}$! Figure 1.3 is an illustration of the semantics of this formula. Further difficulty comes from the nesting of fixpoint operators, which can be extremely difficult to interpret. The reader, again, is referred to [BS01] for a proper treatment of the logic. An exercise for the reader is to decide which fixpoint operator should be used in the definition of deadlock freedom[5].

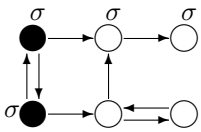
[5] **Answer:** ν, giving $\mathfrak{D}_f^\mu \stackrel{\text{def}}{=} \nu X.\langle A \rangle \top \wedge [A]X$.



(a) Zeroth approximant

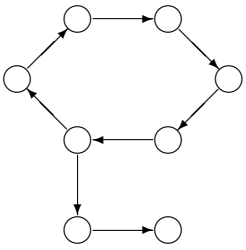


(b) First approximant

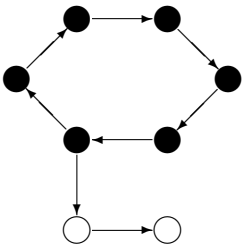


(c) Greatest fixed point

Figure 1.2: Approximants of $X = \sigma \wedge \langle - \rangle X$ to form $\nu X.\sigma \wedge \langle - \rangle X$



(a) $\mu Z.\sigma \vee \langle - \rangle Z$



(b) $\nu Z.\sigma \vee \langle - \rangle Z$

Filled circles represent states in the fixed point solution for Z; hollow circles represent states not in the fixed point solution for Z. σ holds in no state.

Figure 1.3: Differing semantics of least and greatest fixed points for an example state space

It is possible to encode all CTL formulæ quite simply in modal-μ.

- $\exists[\varphi \mathbf{U} \psi] \iff \mu Z.\psi \vee (\varphi \wedge \langle - \rangle Z)$.
- $\forall[\varphi \mathbf{U} \psi] \iff \mu Z.\psi \vee (\varphi \wedge [-]Z \wedge \langle - \rangle Z)$.

The definition of $\forall[-\mathbf{U}-]$ requires that we be able to enter a state in which Z holds because we do not assume that the transition relation is total.

In defining the modalities of CTL$^-$, we make use of the modal-μ equivalence $\neg\mu Z.(\neg\varphi) \equiv \nu Z.\varphi$.

- $\forall \mathbf{G}\varphi \stackrel{\text{def}}{\iff} \nu Z.\varphi \wedge [-]Z$;
- $\exists \mathbf{G}\varphi \stackrel{\text{def}}{\iff} \nu Z.\varphi \wedge (\langle - \rangle Z \vee [-]\bot)$;
- $\forall \mathbf{F}\varphi \stackrel{\text{def}}{\iff} \mu Z.\varphi \vee ([-]Z \wedge \langle - \rangle Z)$;
- $\exists \mathbf{F}\varphi \stackrel{\text{def}}{\iff} \mu Z.\varphi \vee \langle - \rangle Z$.

CTL may not, however, encode all formulæ written in modal-μ; we see that modal-μ is more powerful than CTL (and also, therefore, CTL$^-$).

The Edinburgh Concurrency Workbench [MS] uses this fact — natively, the system checks assertions made in modal-μ, but it is possible to load macros that convert formulæ written in CTL or CTL$^-$ to modal-μ. Users that are not fluent in modal-μ can, therefore, use a simpler language.

1.4 Modal BI

The addition of modalities to **BI** was first suggested in [POY02, Pym02]. Following our pedagogical account of modal calculi, we shall now extend these definitions to define **PBI**[6]. We shall not venture beyond the modalities of HML at this stage, though it would not be hard to extend the logic to allow recursion in the manner of modal-μ, for example. As before, process-resource terms shall be written $\mathbf{R} \lhd [\![P]\!]$, where $\mathbf{R}$ represents the globally available resources and P represents the current state of the process. We define the set of all such states to be $\mathcal{S} = Res \times Proc$.

As is the case for **BI**, the semantics of the logic is defined according to a Kripke resource monoid. This time, though, we shall define the monoid over process-resource pairs:

$$\mathcal{M} = \langle Res \times Proc, (e_R, e_P), \circ, \sqsubseteq \rangle,$$

where $Res \times Proc$ is the set of all process-resource terms $\mathcal{S}$, $\circ$ is a composition operator, (e_P, e_R) is the identity element under $\circ$, and $\sqsubseteq$ is a pre-order on $\mathcal{S}$. As $\mathcal{S}$ is a set of pairs, we shall let $\mathbf{R} \lhd [\![P]\!]$ range over it.

Table 1.3 gives the Kripke resource monoid semantics of the non-modal part of **PBI**. This corresponds to the semantics of **BI**, other than the multiplicative

[6] *Process* **BI**

units, I_P and I_R. These are introduced to allow expression of zero-resource and null-process respectively. So:

$$\|I_R\| = e_R \times Prog$$

$$\|I_P\| = Res \times e_P$$

We may see the normal multiplicative unit of **BI**, I, as $I_R \wedge I_P$. Thus I is extraneous. In the sequel, we shall take modalities to bind more strongly than anything else; between themselves, they shall be of equal precedence.

1.4.1 Additive modalities

As the static logic, **BI**, has both additive and multiplicative operators, it seems natural to introduce both additive and multiplicative modalities into our process logic.

Presented in Table 1.4, the additive modalities describe process evolution without a change in the resource component. For resource formulæ φ_R, such a definition would allow the arbitrary introduction and removal of additive modalities,

$$(\mathrm{DI}_{\mathrm{res}}): \quad \frac{\mathbf{R} \triangleleft [\![P]\!] \models \varphi_R \quad \varphi_R : \mathbf{RForm}}{\mathbf{R} \triangleleft [\![P]\!] \models \langle - \rangle \varphi_R}$$

(and similar for 'box' and elimination rules), where $\varphi_R : \mathbf{RForm}$ inicates that φ_R is a resource formula.

1.4.2 Multiplicative modalities

We shall define four multiplicative modalities, $\langle - \rangle^+_{\mathbf{new}}$, $[-]^+_{\mathbf{new}}$, $\langle - \rangle^-_{\mathbf{new}}$ and $[-]^-_{\mathbf{new}}$, in Table 1.5. In contrast to the additive modalities, we do not have the arbitrary introduction of modalities to resource propositions. Intuitively, they refer to transitions that involve contraction or expansion of the resource component.

The distinction between the diamond and box modalities is conventional: the diamond modalities refer to the existence of a transition satisfying a property, and the box modalities refer to all transitions satisfying a property. The distinction between the '+' and '−' modalities is more interesting. The positive multiplicatives refer to transitions that involve 'releasing' a resource and process component specified by the $\restriction$ relation. The negative modalities refer to transitions that 'acquire' or remove a specified resource and process component (we denote this using $\downharpoonright$, though it could be the same relation).

For the moment, we shall not venture to extend the logic with *e.g.,* the modalities of CTL or modal-μ. Infinitary conjunction and disjunction shall, though, be allowed.

1.4.3 Observable vs. silent actions

Hitherto we have failed to consider how the above modalities may take into account the distinction between observable and silent actions. Two options exist:

$\mathbf{R} \lhd [\![P]\!] \models \mathrm{p}$	iff	$\mathbf{R} \lhd [\![P]\!] \in \|\mathrm{p}\|$
$\mathbf{R} \lhd [\![P]\!] \models \varphi \wedge \psi$	iff	$\mathbf{R} \lhd [\![P]\!] \models \varphi$ and $\mathbf{R} \lhd [\![P]\!] \models \psi$
$\mathbf{R} \lhd [\![P]\!] \models \varphi \vee \psi$	iff	$\mathbf{R} \lhd [\![P]\!] \models \varphi$ or $\mathbf{R} \lhd [\![P]\!] \models \psi$
$\mathbf{R} \lhd [\![P]\!] \models \varphi \to \psi$	iff	for all $\mathbf{R}' \lhd [\![P']\!] \sqsubseteq \mathbf{R} \lhd [\![P]\!]$ $\mathbf{R}' \lhd [\![P']\!] \models \varphi$ implies $\mathbf{R}' \lhd [\![P']\!] \models \psi$
$\mathbf{R} \lhd [\![P]\!] \models \varphi * \psi$	iff	for some $\mathbf{S} \lhd [\![Q]\!], \mathbf{S}' \lhd [\![Q']\!] \in \mathcal{S}$ $[\quad \mathbf{R} \lhd [\![P]\!] \sqsubseteq (\mathbf{S} \lhd [\![Q]\!]) \circ (\mathbf{S}' \lhd [\![Q']\!])$ and $\mathbf{S} \lhd [\![Q]\!] \models \varphi$ and $\mathbf{S}' \lhd [\![Q']\!] \models \psi \quad]$
$\mathbf{R} \lhd [\![P]\!] \models \varphi \mathrel{-\!\!*} \psi$	iff	for all $\mathbf{S} \lhd [\![Q]\!] \in \mathcal{S}$ $\mathbf{S} \lhd [\![Q]\!] \models \varphi$ implies $(\mathbf{R} \lhd [\![P]\!] \circ \mathbf{S} \lhd [\![Q]\!]) \models \psi$
$\mathbf{R} \lhd [\![P]\!] \models \top$	for all	$\mathbf{R} \lhd [\![P]\!] \in \mathcal{S}$
$\mathbf{R} \lhd [\![P]\!] \models \bot$	for any	$\mathbf{R} \lhd [\![P]\!] \in \mathcal{S}$
$\mathbf{R} \lhd [\![P]\!] \models I_P$	iff	for some $\mathbf{R} \in \mathit{Res}.\ \mathbf{R} \lhd [\![P]\!] \sqsubseteq \mathbf{R} \lhd e_P$
$\mathbf{R} \lhd [\![P]\!] \models I_R$	iff	for some $Q \in \mathit{Prog}.\ \mathbf{R} \lhd [\![P]\!] \sqsubseteq e_R \lhd [\![Q]\!]$
$\mathbf{R} \lhd [\![P]\!] \models I$	iff	$\mathbf{R} \lhd [\![P]\!] \sqsubseteq e_r \lhd e_P$

Table 1.3: Kripke semantics of **PBI** (excluding modalities)

$\mathbf{R} \lhd [\![P]\!] \models [A]\varphi$	iff	$\forall \alpha \in A.$ $\mathbf{R} \lhd [\![P]\!] \xrightarrow{\alpha} \mathbf{R} \lhd [\![P']\!]$ implies $\mathbf{R} \lhd [\![P']\!] \models \varphi$
$\mathbf{R} \lhd [\![P]\!] \models \langle A \rangle \varphi$	iff	$\exists \alpha \in A.$ $\mathbf{R} \lhd [\![P]\!] \xrightarrow{\alpha} \mathbf{R} \lhd [\![P']\!]$ and $\mathbf{R} \lhd [\![P']\!] \models \varphi$

Table 1.4: Kripke semantics of additive modalities of **PBI**

$\mathbf{R} \lhd [\![P]\!] \models \langle A \rangle^{+}_{\mathbf{new}} \varphi$	iff	$\exists a \in A.\quad \exists \mathbf{S} \lhd [\![Q]\!] \in \mathcal{P}_{\mathrm{ROC}}.$ $a \restriction \mathbf{S} \lhd [\![Q]\!]$ and $\mathbf{R} \lhd [\![P]\!] \xrightarrow{a} (\mathbf{R} \lhd [\![P']\!]) \circ (\mathbf{S} \lhd [\![Q]\!])$ and $(\mathbf{R} \lhd [\![P']\!]) \circ (\mathbf{S} \lhd [\![Q]\!]) \models \varphi$
$\mathbf{R} \lhd [\![P]\!] \models [A]^{+}_{\mathbf{new}} \varphi$	iff	$\forall a \in A.\quad \forall \mathbf{S} \lhd [\![Q]\!] \in \mathcal{P}_{\mathrm{ROC}}.$ $a \restriction \mathbf{S} \lhd [\![Q]\!]$ and $\mathbf{R} \lhd [\![P]\!] \xrightarrow{a} (\mathbf{R} \lhd [\![P']\!]) \circ (\mathbf{S} \lhd [\![Q]\!])$ implies $(\mathbf{R} \lhd [\![P']\!]) \circ (\mathbf{S} \lhd [\![Q]\!]) \models \varphi$
$\mathbf{R} \lhd [\![P]\!] \models \langle A \rangle^{-}_{\mathbf{new}} \varphi$	iff	$\exists a \in A.\quad \exists \mathbf{S} \lhd [\![Q_1]\!], \mathbf{T} \lhd [\![Q_2]\!] \in \mathcal{P}_{\mathrm{ROC}}.$ $a \restriction \mathbf{S} \lhd [\![Q_1]\!]$ and $\mathbf{R} \lhd [\![P]\!] \sqsubseteq (\mathbf{S} \lhd [\![Q_1]\!]) \circ (\mathbf{T} \lhd [\![Q_2]\!])$ and $\mathbf{R} \lhd [\![P]\!] \xrightarrow{a} \mathbf{T} \lhd [\![Q_2']\!]$ and $\mathbf{T} \lhd [\![Q_2']\!] \models \varphi$
$\mathbf{R} \lhd [\![P]\!] \models [A]^{-}_{\mathbf{new}} \varphi$	iff	$\forall a \in A.\quad \forall \mathbf{S} \lhd [\![Q]\!], \mathbf{T} \lhd [\![Q_2]\!] \in \mathcal{P}_{\mathrm{ROC}}.$ $a \restriction \mathbf{S} \lhd [\![Q_1]\!]$ and $\mathbf{R} \lhd [\![P]\!] \sqsubseteq (\mathbf{S} \lhd [\![Q_1]\!]) \circ (\mathbf{T} \lhd [\![Q_2]\!])$ and $\mathbf{R} \lhd [\![P]\!] \xrightarrow{a} \mathbf{T} \lhd [\![Q_2']\!]$ implies $\mathbf{T} \lhd [\![Q_2']\!] \models \varphi$

Table 1.5: Kripke semantics of multiplicative modalities of **PBI**

- The transition system could be augmented with rules to provide

  $$\xrightarrow{\sigma} = \xrightarrow{\tau}{}^{*} \xrightarrow{\sigma} \xrightarrow{\tau}{}^{*}$$

  for all observable σ.

- Observable transition modalities could be introduced to complement $\overset{\sigma}{\Longrightarrow}$. They would be just as the modalities presented here, other than requiring an observable action $\overset{\sigma}{\Longrightarrow}$ rather than any type.

We shall adopt the second solution: it allows us to define distinct notions of program equivalence, and otherwise results in a clearer notion of activity in the logic.

1.5 Petri nets in PBI

Petri nets can be used to represent evolving systems. State of the system is described by the distribution of *tokens* to *places*, forming a *marking*. State evolves by the pre-conditions (a set of places) of an action being met (having a token), along with all the post-conditions (a set of places) being empty. Following the action, the places forming its post-conditions are then marked. Several introductions to Petri nets exist, *e.g.,* [Mur89] and the introduction to [MBC+95]. Petri nets can be represented graphically: it is conventional to draw

$\mathbf{R} \lhd [\![P]\!] \models \langle\!\langle A \rangle\!\rangle \varphi$	iff	$\exists a \in A.$ $\quad \mathbf{R} \lhd [\![P]\!] \stackrel{a}{\Longrightarrow} \mathbf{R} \lhd [\![P']\!]$ $\wedge \mathbf{R} \lhd [\![P']\!] \models \varphi$
$\mathbf{R} \lhd [\![P]\!] \models \lceil\!\lceil A \rfloor\!\rfloor \varphi$	iff	$\forall a \in A.$ $\quad \mathbf{R} \lhd [\![P]\!] \stackrel{a}{\Longrightarrow} \mathbf{R} \lhd [\![P']\!]$ $\wedge \mathbf{R} \lhd [\![P']\!] \models \varphi$
$\mathbf{R} \lhd [\![P]\!] \models \langle\!\langle A \rangle\!\rangle^{+}_{\mathbf{new}} \varphi$	iff	$\exists a \in A.$ $\quad a \upharpoonright \mathbf{S} \lhd [\![Q]\!]$ and $\mathbf{R} \lhd [\![P]\!] \stackrel{a}{\Longrightarrow} (\mathbf{R} \lhd [\![P']\!]) \circ (\mathbf{S} \lhd [\![Q]\!])$ and $(\mathbf{R}' \lhd [\![P']\!]) \circ (\mathbf{S} \lhd [\![Q]\!]) \models \varphi$
$\mathbf{R} \lhd [\![P]\!] \models \lceil\!\lceil A \rfloor\!\rfloor^{+}_{\mathbf{new}} \varphi$	iff	$\forall a \in A.$ $\quad a \upharpoonright \mathbf{S} \lhd [\![Q]\!]$ and $\mathbf{R} \lhd [\![P]\!] \stackrel{a}{\Longrightarrow} (\mathbf{R} \lhd [\![P']\!]) \circ (\mathbf{S} \lhd [\![Q]\!])$ implies $(\mathbf{R}' \lhd [\![P']\!]) \circ (\mathbf{S} \lhd [\![Q]\!]) \models \varphi$
$\mathbf{R} \lhd [\![P]\!] \models \langle\!\langle A \rangle\!\rangle^{-}_{\mathbf{new}} \varphi$	iff	$\exists a \in A.\ a \upharpoonright \mathbf{S} \lhd [\![Q_1]\!]$ and $\mathbf{R} \lhd [\![P]\!] \sqsubseteq (\mathbf{S} \lhd [\![Q_1]\!]) \circ (\mathbf{T} \lhd [\![Q_2]\!])$ and $\mathbf{R} \lhd [\![P]\!] \stackrel{a}{\Longrightarrow} \mathbf{T} \lhd [\![Q_2']\!]$ and $\mathbf{T} \lhd [\![Q_2']\!] \models \varphi$
$\mathbf{R} \lhd [\![P]\!] \models \lceil\!\lceil A \rfloor\!\rfloor^{-}_{\mathbf{new}} \varphi$	iff	$\forall a \in A.\ a \upharpoonright \mathbf{S} \lhd [\![Q_1]\!]$ and $\mathbf{R} \lhd [\![P]\!] \sqsubseteq (\mathbf{S} \lhd [\![Q_1]\!]) \circ (\mathbf{T} \lhd [\![Q_2]\!])$ and $\mathbf{R} \lhd [\![P]\!] \stackrel{a}{\Longrightarrow} \mathbf{T} \lhd [\![Q_2']\!]$ implies $\mathbf{T} \lhd [\![Q_2']\!] \models \varphi$

Table 1.6: Kripke semantics of observable modalities of **PBI**

places as circles, tokens as dots, and actions as bars. Pre-conditions of actions are drawn as arrows leading into the action, and post-conditions are drawn as arrows leading away. Figure 1.4 gives some examples.

More formally, a net is a 4-tuple, $N = \langle P, T, pre, post \rangle$. A *marking* M is a function giving the number of tokens in a given place. We shall consider $M : P \to \{0, 1\}$. Taking $\mathcal{M}$ to be the set of all markings, P is the set of places (circles), T the set of transitions (bars), and $pre, post : T \to \mathcal{M}$ are functions that return the required markings before and after a given transition (arrows).

As described in [POY02, Pym02], it is possible to encode Petri nets in **BI**. We denote composition of markings M and N, $M + N$, and define:

$$(M + N)p = \begin{cases} 0, & Mp = 0 \wedge Np = 0 \\ 1, & \text{otherwise} \end{cases} .$$

This restricts our attention to the variant of Petri nets where at most one token occurs in any place. $[-]$ shall denote the empty marking (*i.e.,* for all p, $[-]p = 0$). We consider only finite Petri nets, so $\mathsf{dom}(M)$ is finite.

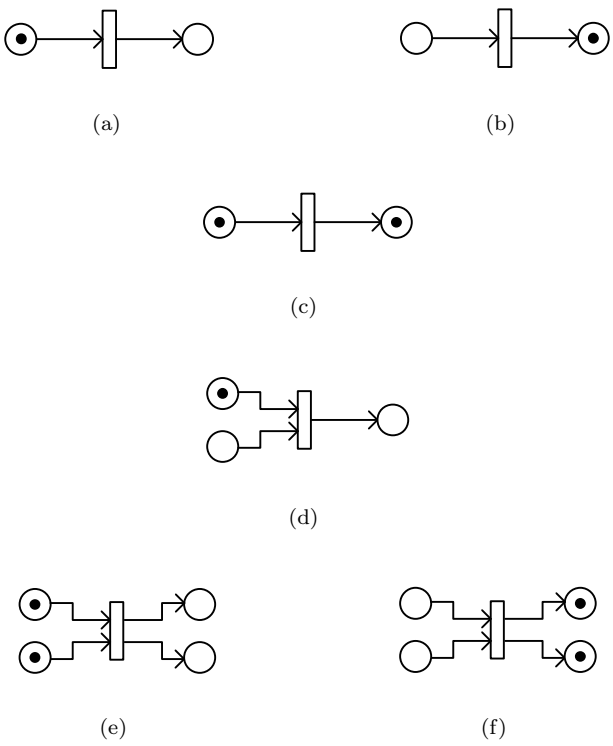


Figure 1.4: 1.4(a) goes to 1.4(b). 1.4(c): no action — blocked by post-condition. 1.4(d): no action — not all pre-conditions met. 1.4(e) goes to 1.4(f)

Thus [POY02] defines a pre-ordered commutative monoid

$$\langle \mathcal{M}, [-], +, \sqsubseteq \rangle.$$

Petri nets, therefore, give semantics to **BI**. We represent action (transitions) between Petri net markings using $\sqsubseteq$. So, for example, $N \sqsubseteq M$ iff from M a number of transitions lead to a marking N. This interpretation makes $\rightarrow$, in a sense, a global temporal operator similar to $\forall\mathbf{G}$ in CTL: $m \models \varphi \rightarrow \psi$ is true if in all reachable states, if φ is true then ψ is true. We also lose the ability to describe only the current marking using multiplicative conjunction, $*$.

As mentioned before, we chose to allow only one token per place in each marking. Depending on the application, it would be quite easy to generalise away from this restriction: we would simply take $M, N : P \rightarrow \mathbb{N}$ and $(M+N)p = Mp+Np$, and relax the requirement that the post-condition of a transition be empty before the transition can take place.

It is also posssible to encode Petri nets in **PBI**. In essence, all we do is take marked places as resources, and the graph formed by $\langle pre, post, P, T \rangle$ as the process term. Graphically, thhe process term would simply be the net with no places marked. Writing $N[X]$ to mean a Petri net N that has a sub-net X and $\mathbf{M}$ for a marking (set of marked places), we can define a transition relation to describe the activity of Petri nets. As an example,

$$(\text{STRAN}) : \frac{}{\mathbf{M}, p \lhd [\![N[p \rightarrow \boxed{\text{t}} \rightarrow m]]\!] \xrightarrow{t} \mathbf{M}, m \lhd [\![N[p \rightarrow \boxed{\text{t}} \rightarrow m]]\!]} (m \notin \mathbf{M})$$

for an arbitrary net $N[p \rightarrow \boxed{\text{t}} \rightarrow m]$ with a transition t such that $pre(t) = \{p\}$ and $post(t) = \{m\}$; the pre- and post-conditions in this specific case each comprise of only one place.

With such a transition relation, we can ask questions relating to the consequences of additional places being marked and tokens being removed from places. For example, the semantics gives us that

$$\wr \int \lhd [\![N]\!] \models m \mathrel{-\!\!*} \varphi$$

asks if φ holds of a state in which only m is marked. We may use the modalities to ask of the markings formed by transitions. The correspondence between questions posed of Demos states and Petri nets is so great that the reader is referred to §4.1 for a discussion of the logic presented.

The invariance in the process term need not necessarily be so: we could have a Petri net that changes. Such a system would describe a *dynamic* Petri net [AB96]. Unlike our presentation of additive modalities here, we could perhaps consider additive modalities as referring to transitions in which only the marking changes, and multiplicative modalities as referring to transitions in which the net may change. Further consideration of this, though, is beyond our current scope.

Chapter 2

Demos

2.1 Introduction

Demos [BT01] is a simulation language for modelling processes, executing in parallel, that interact through taking, and then relinquishing, control of resources.

As indicated by the title of the manual [BT01], the value of the language is that its operational semantics is clearly defined [TB01]. It is possible, therefore, to define translations between Demos programs and other systems and then prove the translation correct. For example, Demos programs can be translated to CCS processes [TB94, TB95, BTa, BTb] and to Petri nets [Tof01] (again, if we fail to model time). Because the translation is demonstrably correct, we can have confidence that any results obtained of a Demos model by, say, translating to CCS and then using the Edinburgh Concurrency Workbench [MS], are correct.

We refer the reader to [BT01] for an introduction to the language.

This document, in order to maintain a concise notation, shall consider only a subclass of Demos programs, the syntax of which is given in Backus-Naur form in Figure 2.1. Notably, this does not include variables, or bins or synchronisations with values.

Returning to our intelligence agency example, we may model the system in Demos by the program of Figure 2.2. The trace (omitted) indicates no deadlock.

We shall use the abbreviation ε for the *null* process, a process incapable of any action. Letting $\mathcal{P}_{\mathrm{ROG}}$ be the set of correct processes derived from the BNF description, the set of processes is $Code = \{\varepsilon\} \cup \mathcal{P}_{\mathrm{ROG}}$. Writing $\cdot \| \cdot$ for the parallel composition of processes and $\cdot ; \cdot$ for the sequential composition of processes (see later), the following structural equivalences should be taken to

$$
\begin{array}{rl}
Prog ::= & \texttt{Res}(r,n) \mid \texttt{Bin}(r,n) \mid \\
& \texttt{Entity}(class,t) \mid \texttt{Class}\ name = \{Decl\} \mid \\
& \texttt{close} \mid Prog; Prog \mid \texttt{hold}(t) \\
\\
Decl ::= & \texttt{getR}(r,n) \mid \texttt{putR}(r,n) \mid \\
& \texttt{getB}(r,n) \mid \texttt{putB}(r,n) \mid \\
& \texttt{sync}(x) \mid \texttt{getS}(x,n) \mid \texttt{putS}(x,n) \mid \\
& \texttt{try}\ [cond]\ \texttt{then}\ Decl\ \texttt{etry}\ [cond]\ \texttt{then}\ Decl \mid \\
& \texttt{while}\ [cond]\ \texttt{do}\ Decl \mid \texttt{do}\ n\ Decl \mid \texttt{req}[cond] \mid \\
& \texttt{Entity}(class,t) \mid \texttt{hold}(t) \mid \\
& Decl\,;Decl \\
\\
cond ::= & \texttt{getR}(r,n) \mid \texttt{getB}(r,n) \mid \texttt{getS}(x,n) \mid cond, cond
\end{array}
$$

Figure 2.1: BNF syntax of restriced Demos considered

```
cons ACT_TIME6 4;
cons ACT_TIMECIA 4;

class MI6 = {
  getR(MI6_line, 1); hold(2);
  getR(MI5_line, 1); hold(2);
  getR(FBI_line, 1); hold(2);

  hold(ACT_TIME6);

  putR(FBI_line, 1); putR(MI5_line, 1); putR(MI6_line, 1);
}

class CIA = {
  getR(CIA_line, 1); hold(2);
  getR(FBI_line, 1); hold(2);
  getR(MI5_line, 1); hold(2);

  hold(ACT_TIMECIA);

  putR(MI5_line, 1); putR(FBI_line, 1); putR(CIA_line, 1);
}

Res(CIA_line, 1); Res(FBI_line, 1);
Res(MI6_line, 1); Res(FBI_line, 1);

Entity(CIA, 12*60 + 40); %12:40
Entity(MI6, 12*60 + 43); %12:43
```

Figure 2.2: Demos program for the intelligence agency example

exist:

$$\begin{aligned}
[\![P \| Q]\!] &\equiv [\![Q \| P]\!] \\
[\![P \| (Q_1 \| Q_2)]\!] &\equiv [\![(P \| Q_1) \| Q_2]\!] \\
[\![P \| \varepsilon]\!] &\equiv [\![P]\!] \\
[\![\varepsilon; P]\!] &\equiv [\![\varepsilon]\!] \\
[\![P; \varepsilon]\!] &\equiv [\![P]\!]
\end{aligned}$$

We shall, though, define the transition system so as to make these equivalences unnecessary in the derivation of transitions.

The reader shall recall that in our definition of **PBI** we required a transition relation to describe the activity of processes. This is presented in the following section, and shall be called σDemos.

2.2 Transition rules

Notation Let R denote the set of actual resources; $r_0, r_1, \ldots$ shall range over R. In the sequel, we shall find it convenient to think of processes as resources. We can define the set of resources (actual and processes)

$$\mathcal{R} \stackrel{\text{def}}{=} \mathcal{P}_+(R) \uplus \mathcal{P}(\mathcal{P}_{\text{ROC}})$$

where $\mathcal{P}_+(R)$ denotes the set of all sub-multisets of R. $\mathbf{R} \in \mathcal{R}$ shall denote a multiset of resources; multiset operations shall be as set operations tagged with a '+'. $\mathbf{R}, r$ shall denote the multiset $\mathbf{R}$ with one unit of resource r added. That is,

$$\mathbf{R}, r \stackrel{\text{def}}{=} \mathbf{R} \uplus \wr r \int .$$

We shall write r^n for $\uplus_1^n \wr r \int$. A *state* in the transition system shall be represented by a pair, $s \in \mathcal{R} \times \mathcal{P}_{\text{ROG}}$. We let the set of all states be $\Omega \stackrel{\text{def}}{=} \mathcal{R} \times \mathcal{P}_{\text{ROC}}$.

Let the set Act be the set of actions[1] that any Demos program may make, ranged over by σ. We can define a transition relation, $\longrightarrow \subseteq \Omega \times Act \times \Omega$, between programs and the resources they hold; we shall use infix notation for the relation. It is defined, inductively, according to the following set of rules, which essentially give the (exclusively) resource-sensitive, *non-temporal* structural operational semantics of Demos programs.

As well as observable actions, we shall allow *silent* (non-observable) transitions; such transitions shall be written $\mathbf{R} \triangleleft [\![P]\!] \xrightarrow{\tau} \mathbf{R}' \triangleleft [\![P']\!]$[2]. Recall that this should decrease the level of non-determinism within the proof system compared to an equivalence relation.

It may be useful later to define a new transition relation,

$$\stackrel{\sigma}{\Longrightarrow} \stackrel{\text{def}}{=} \xrightarrow{\tau}{}^* \xrightarrow{\sigma} \xrightarrow{\tau}{}^*,$$

[1] Silent and observable

[2] The inspiration for 'τ', obviously, is CCS

where S^* denotes the reflexive, transitive closuse of the relation S. Alternatively, we may define $\Longrightarrow$ inductively:

$$(\text{OBSER}_0): \quad \frac{\mathbf{R} \lhd [\![P]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P']\!]}{\mathbf{R} \lhd [\![P]\!] \overset{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![P']\!]}$$

$$(\text{OBSER}_1): \quad \frac{\mathbf{R} \lhd [\![P]\!] \xrightarrow{\tau} \mathbf{R}'' \lhd [\![P'']\!] \qquad \mathbf{R}'' \lhd [\![P'']\!] \overset{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![P']\!]}{\mathbf{R} \lhd [\![P]\!] \overset{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![P']\!]}$$

$$(\text{OBSER}_2): \quad \frac{\mathbf{R} \lhd [\![P]\!] \overset{\sigma}{\Longrightarrow} \mathbf{R}'' \lhd [\![P'']\!] \qquad \mathbf{R}'' \lhd [\![P'']\!] \xrightarrow{\tau} \mathbf{R}' \lhd [\![P']\!]}{\mathbf{R} \lhd [\![P]\!] \overset{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![P']\!]}$$

The symbol P (and $P_1, P_2, \ldots, Q, \ldots$) is used frequently throughout this document. In general, it ranges over processes, including parallel processes. It is also used to range over non-parallel processes — that is, bodies of code. The reader should extract the appropriate choice between these from the syntax of Demos code.

2.3 Transition system σDemos

We shall write $\mathbf{R} \lhd [\![\varepsilon]\!]$ for cleanly terminated states. The restricted subset of Demos considered here has the useful property that one can deduce what resources are held by a given process by examining the code of the process, under the assumption that the process code is *well formed* (only `puts` resources that it holds) — for our restricted language, this property can be checked syntactically. Consequently, we do not need to store this information, *e.g.,* in the process term.

The declaration of each process (the `Entity` command) attaches to each non-parallel process a label $\ell \in \mathit{Label}$, where Label is the set of all process labels. A process P labelled ℓ shall be written $P_{[\ell]}$. For example, if the jobbers in the jobshop example are called Bob and Alice, we would write $[\![J_{[\text{Bob}]} \| J_{[\text{Alice}]}]\!]$. The process label shall be included in the action label of any transition derived by a non-parallel process so that in the process logic we will be able to determine which process is acting. We shall illustrate this in the first rule defined below and in the definition of `Entity`; otherwise, for clarity, it shall be omitted. Moreover, in the sequel, except where this label is of particular interest to us, we shall omit the process labels. We shall assume that σ allows labels to be carried through from sub-derivations.

2.3.1 Resource actions

We commence by defining how processes acquire and release resources in σDemos. To `get` a resource, a process simply removes the resource from the collection of available resources if it is available. If it is not, the process will not be able to pass the command, thus simulating the process waiting for the resource. Releasing a resource is just a case of replacing the resource in the collection. The rules for bins and actual resources are the same in σDemos; in Demos itself, however, bins are used to model resources that may be generated or consumed

by processes. For example, a process that must withdraw £50 from a bank account b before executing P should be modelled $\texttt{getB}(b, 50); P$.

$$(\mathrm{GET}^{\mathrm{n}}_{\mathrm{R}}): \quad \frac{}{\mathbf{R}, r^n \lhd [\![\texttt{getR}(r,n)_{[\ell]}]\!] \xrightarrow{\ell:\mathrm{getR}^n_r} \mathbf{R} \lhd [\![\varepsilon]\!]}$$

$$(\mathrm{PUT}^{\mathrm{n}}_{\mathrm{R}}): \quad \frac{}{\mathbf{R} \lhd [\![\texttt{putR}(r,n)]\!] \xrightarrow{\mathrm{putR}^n_r} \mathbf{R}, r^n \lhd [\![\varepsilon]\!]}$$

$$(\mathrm{GET}^{\mathrm{n}}_{\mathrm{B}}): \quad \frac{}{\mathbf{R}, r^n \lhd [\![\texttt{getB}(r,n)]\!] \xrightarrow{\mathrm{getB}^n_r} \mathbf{R} \lhd [\![\varepsilon]\!]}$$

$$(\mathrm{PUT}^{\mathrm{n}}_{\mathrm{B}}): \quad \frac{}{\mathbf{R} \lhd [\![\texttt{putB}(r,n)]\!] \xrightarrow{\mathrm{putB}^n_r} \mathbf{R}, r^n \lhd [\![\varepsilon]\!]}$$

Note that in the rules (PUT), we have assumed that processes are well formed in that we have not checked that they own the resources that they are returning.

2.3.2 Hold

As we have already stated, our model of Demos programs shall not consider time. Thus the transition for $\texttt{hold}(t)$ shall be:

$$(\mathrm{NOTIME}): \quad \frac{}{\mathbf{R} \lhd [\![\texttt{hold}(t)]\!] \xrightarrow{\tau} \mathbf{R} \lhd [\![\varepsilon]\!]},$$

effectively skipping the command. Notice that we have used a τ-transition to indicate that this is not observable. [3]

2.3.3 Initialisation

It is also possible to define initialisation of the program (allowing a little vagueness by assuming that we are passed the code of *class* rather than just its name. This could be resolved by introducing a function $\Sigma : \mathit{classname} \to \mathit{code}$). Again, notice that we have defined a τ-transition.

$$(\mathrm{ENTITY}): \quad \frac{}{\mathbf{R} \lhd [\![\texttt{Entity}(class, name, t); P]\!] \xrightarrow{\tau^{class,name}_{\mathrm{Entity}}} \mathbf{R} \lhd [\![class_{[name]} \| P]\!]}$$

$$(\mathrm{RES_n}): \quad \frac{}{\mathbf{R} \lhd [\![\texttt{Res}(r,n)]\!] \xrightarrow{\tau^{r,n}_{\mathrm{Res}}} \mathbf{R}, r^n \lhd [\![\varepsilon]\!]}$$

$$(\mathrm{BIN_n}): \quad \frac{}{\mathbf{R} \lhd [\![\texttt{Bin}(b,n)]\!] \xrightarrow{\tau^{b,n}_{\mathrm{Bin}}} \mathbf{R}, b^n \lhd [\![\varepsilon]\!]}$$

[3] It may be more useful to split the `hold` into two transitions, $\mathbf{R} \lhd [\![\texttt{hold}(t)]\!] \xrightarrow{\mathrm{startstop}} \longrightarrow$, for model-checking some properties *e.g.,* mutual exclusion over a critical region.

2.3.4 Sequential composition

The sequential composition of program code is relatively straightforward to define: one command is executed, which changes the state of the system, with the rest of the program remaining. c shall range over single commands which may be compound *e.g.,* `while` loops. There are two cases to consider. Firstly, the first command executes completely:

$$(\mathrm{COMP_{Te}}): \quad \frac{\mathbf{R} \lhd [\![c]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![\varepsilon]\!]}{\mathbf{R} \lhd [\![c; P_2]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P_2]\!]}$$

If the command at the head executes to give an intermediate step, we place this back in front:

$$(\mathrm{COMP_{I}}): \quad \frac{\mathbf{R} \lhd [\![c]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P_1']\!]}{\mathbf{R} \lhd [\![c; P_2]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P_1'; P_2]\!]} \, (P_1' \not\equiv \varepsilon)$$

2.3.5 Parallel composition

We say that processes are acting in *parallel* or *concurrently* if more than one process is 'running' at the same time. We shall use $\cdot \| \cdot$ to represent parallel composition of processes. Each parallel process, providing it is not blocked (waiting for a resource not free in $\mathbf{R}$), may be executed. At this stage, we choose not to define a notion of *fairness* in execution — see §6.1.

One approach that may be adopted is to only allow one process to execute one step in order to form a transition of the parallel system, forming a so-called *asynchronous* transition system. The choice of which process to execute is arbitrary, or non-deterministic.

$$(\mathrm{PAR_{I,1}}): \quad \frac{\mathbf{R} \lhd [\![P_1]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P_1']\!]}{\mathbf{R} \lhd [\![P_1 \| P_2]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P_1' \| P_2]\!]}$$

$$(\mathrm{PAR_{I,2}}): \quad \frac{\mathbf{R} \lhd [\![P_2]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P_2']\!]}{\mathbf{R} \lhd [\![P_1 \| P_2]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P_1 \| P_2']\!]}$$

We must also consider the cases where one of the parallel processes terminates.

$$(\mathrm{PAR_{Te,1}}): \quad \frac{\mathbf{R} \lhd [\![P_1]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![\varepsilon]\!]}{\mathbf{R} \lhd [\![P_1 \| P_2]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P_2]\!]}$$

$$(\mathrm{PAR_{Te,2}}): \quad \frac{\mathbf{R} \lhd [\![P_2]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![\varepsilon]\!]}{\mathbf{R} \lhd [\![P_1 \| P_2]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P_1]\!]}$$

Alternatively, we may define a rule schema to allow easy access to parallel processes. One may consider this to be a derived set of rules, or consider the above to be specific instances of this; this is similar to the Expansion Law of CCS.

$$\left(\mathrm{PAR_I^i}\right): \quad \frac{\mathbf{R} \lhd [\![P_i]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P_i']\!]}{\mathbf{R} \lhd \left[\!\!\left[\|_{j \in I} P_j \right]\!\!\right] \xrightarrow{\sigma} \mathbf{R}' \lhd \left[\!\!\left[P_i' \|_{j \in I \setminus \{i\}} P_j \right]\!\!\right]} \begin{pmatrix} I = \{1, \ldots, n\}, \\ n > 1, \\ i \in I \end{pmatrix}$$

with a similar rule ($\mathrm{PAR}_{\mathrm{Te}}$).

See §5 for consideration of a rule for synchronous parallel execution.

2.3.6 Process synchronisation

A powerful feature of Demos is that it is able to model *synchronisation* between processes. For example, a process representing a train may synchronise with a process representing a station to represent loading/unloading. The train process will, once it has placed itself in the *sync pool* of arrived trains, not be able to perform any action until it is released by the station — after it has been allocated a platform and all the passengers have boarded.

To define synchronisation, we must have a notation for representing processes waiting in a pool and processes holding other processes.

Notation $\overset{x}{\overline{(\!|P|\!)}}$ shall represent a process P that has synchonized on a pool x. This shall lie in the resource term $\mathbf{R}$. When this process is claimed by another process, it shall be written $(\!|P|\!)_x$.
A process Q holding a multiset of processes $S = \wr (\!|P_1|\!)_{x_1}, \ldots, (\!|P_n|\!)_{x_n} \int$ that synchronised on pools $x_1, \ldots, x_n$ shall be written $\left[\!\left[\frac{Q}{S} \right]\!\right]$.

We see, then, that a process is actually a pair consisting of the remaining code and the processes synchronised on it[4]. Where the collection of processes that synchronised on a process is not affected by a rule (for example, sequential composition does not itself modify the collection of synchronized processes, though a sub-derivation may do so), in general we shall not specify this in the rule — for $\mathbf{R} \lhd [\![P]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P']\!]$ it shall be implicit in $[\![P]\!]$ and $[\![P']\!]$. Otherwise, where the collection is explicitly modified, we shall write rules of the form

$$\mathbf{R} \lhd \left[\!\left[\frac{P}{S} \right]\!\right] \xrightarrow{\sigma} \mathbf{R}' \lhd \left[\!\left[\frac{P'}{S, s} \right]\!\right].$$

The upper-case letter S denotes a multiset; the lower-case letter s denotes a single item. The above definition extends in the obvious manner to parallel processes.

$$(\mathrm{SYNC}^{\mathrm{x}}): \quad \frac{}{\mathbf{R} \lhd [\![\mathtt{sync}(x); P]\!] \overset{\mathrm{sync}_x}{\longrightarrow} \mathbf{R}, \overset{x}{\overline{(\!|P|\!)}} \lhd [\![\varepsilon]\!]}$$

$$(\mathrm{GETS}^{\mathrm{x}}): \quad \frac{}{\mathbf{R}, \overset{x}{\overline{(\!|P_i|\!)}}^{\,n} \lhd \left[\!\left[\frac{\mathtt{getS}(x,n)}{S} \right]\!\right] \overset{\mathrm{getS}^n_x}{\longrightarrow} \mathbf{R} \lhd \left[\!\left[\frac{\varepsilon}{S, (\!|P_i|\!)_x{}^n} \right]\!\right]}$$

$$(\mathrm{PUTS}^{\mathrm{x}}): \quad \frac{}{\mathbf{R} \lhd \left[\!\left[\frac{\mathtt{putS}(x,n);Q}{S, (\!|P_i|\!)^n_x} \right]\!\right] \overset{\mathrm{putS}^n_x}{\longrightarrow} \mathbf{R} \lhd \left[\!\left[\frac{Q}{S} \| _n P_i \right]\!\right]}$$

[4] $\mathcal{P}_{\mathrm{ROC}} = \mathit{Label} \times \mathit{Code} \times \mathcal{P}_+(\mathcal{P}_{\mathrm{ROC}})$, where $\mathcal{P}_+$ is the multiset version of powerset, *Code* is the set of syntactically correct Demos programs and *Label* is the set of process labels. The reader will, of course, recognise that this type, $\mathcal{P}_{\mathrm{ROC}}$, represents the solution of a recursive domain equation.

Though we shall not write implicity modified synchronized process collections, it is important that the reader recognizes that they exist. So, for example, sequential composition is actually:

$$(\text{COMP}_\text{I}): \quad \frac{\mathbf{R} \lhd [\![\frac{P}{S}]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![\frac{P'}{S'}]\!]}{\mathbf{R} \lhd [\![\frac{P;Q}{S}]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![\frac{P';Q}{S'}]\!]}$$

$$(\text{COMP}_\text{Te}): \quad \frac{\mathbf{R} \lhd [\![\frac{P}{S}]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![\frac{\varepsilon}{S'}]\!]}{\mathbf{R} \lhd [\![\frac{P;Q}{S}]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![\frac{Q}{S'}]\!]}$$

2.3.7 Non-blocking requests

In order to define valid transition rules for `try` and `while`, we shall need a non-blocking multi-request (a request where the list of actions is atomic — if any one of the requests fails, the whole list fails — and that we can detect failure of); `req` will be similar. We shall use the non-program label $g[\cdot]$ to denote such a request.

When we fail to complete a non-blocking request, we shall indicate this using the symbol $\mathcal{FF}$.

$$(\text{NBLOCK}^\text{T}_\text{P}): \quad \frac{\mathbf{R} \lhd [\![\texttt{getS}(x,1)]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![\varepsilon]\!]}{\mathbf{R} \lhd [\![g[\texttt{getS}(x,1)]]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![\varepsilon]\!]}$$

$$(\text{NBLOCK}^\text{F}_\text{P}): \quad \frac{}{\mathcal{FF}(\mathbf{R} \lhd [\![g[\texttt{getS}(x,1)]]\!])} \left(\neg \exists P. \ \overline{\overset{x}{\langle P \rangle}} \in \mathbf{R} \right)$$

$$(\text{NBLOCK}^\text{T}_\text{R}): \quad \frac{\mathbf{R} \lhd [\![\texttt{getR}(r,n)]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![\varepsilon]\!]}{\mathbf{R} \lhd [\![g[\texttt{getR}(r,n)]]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![\varepsilon]\!]}$$

$$(\text{NBLOCK}^\text{F}_\text{R}): \quad \frac{}{\mathcal{FF}(\mathbf{R} \lhd [\![g[\texttt{getR}(r,n)]]\!])} \ (r^n \notin \mathbf{R})$$

$$(\text{NBLOCK}^\text{T}): \quad \frac{}{\mathbf{R} \lhd [\![g[]]\!] \xrightarrow{\tau} \mathbf{R} \lhd [\![\varepsilon]\!]}$$

$$(\text{NBLOCK}^\text{T}_{1,2}): \quad \frac{\mathbf{R} \lhd [\![g[e_i]]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![\varepsilon]\!] \qquad \mathbf{S} \lhd [\![g[e_j]]\!] \xrightarrow{\sigma} \mathbf{S} \lhd [\![\varepsilon]\!]}{\mathbf{R} \uplus \mathbf{S} \lhd [\![g[e_1, e_2]]\!] \xrightarrow{\sigma} \mathbf{R}' \uplus \mathbf{S}' \lhd [\![\varepsilon]\!]}$$

$$(\text{NBLOCK}^\text{F}): \quad \frac{\mathcal{FF}(\mathbf{R} \lhd [\![g[r_i]]\!])}{\mathcal{FF}(\mathbf{R} \lhd [\![g[r_1, r_2]]\!])} \ (i \in \{1,2\})$$

where $e ::= e, e \mid \texttt{getR}(r,n) \mid \texttt{getB}(r,n) \mid \texttt{getS}(x,1), \quad r \in R, n \in \mathbb{Z}^+$.

These definitions are relatively straightforward; their value is that they deal with comma-separated lists and define $\mathcal{FF}$.

2.3.8 Try, while and req

The `try` command introduces conditional choice into Demos. The command `try` $[e_1]$ `then` P_1 `etry` $[e_2]$ `then` P_2 attempts the atomic list of actions e_1. If this succeeds, P_1 shall be executed. Otherwise, e_2 is tested to see if P_2 should be executed. If none of the e_i pass, the process waits (no derivable transition) until one does. If more than one pass at the same time, the earliest P_i shall be selected.

Hence we can define the rules for `try`:

$$(\mathrm{TRY}^{\mathrm{T}}_{\mathrm{Te}}): \quad \frac{\mathbf{R} \lhd [\![g[e_1]]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![\varepsilon]\!]}{\mathbf{R} \lhd [\![\texttt{try } [e_1] \texttt{ then } P \texttt{ etry } [e_2] \texttt{ then } Q]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P]\!]}$$

$$(\mathrm{TRY}^{\mathrm{F}}_{\mathrm{Te}}): \quad \frac{\mathcal{FF}(\mathbf{R}, g[e_1]) \qquad \mathbf{R} \lhd [\![g[e_2]]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![\varepsilon]\!]}{\mathbf{R} \lhd [\![\texttt{try } [e_1] \texttt{ then } P \texttt{ etry } [e_2] \texttt{ then } Q]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![Q]\!]}$$

The `while` construct defines a loop: while an expression is true, a body of code will be executed. Should the expression be false at the beginning of the loop, iteration will stop. The `req` command simply defines an atomic series of actions.

$$(\mathrm{WHILE}^{\mathrm{T}}_{\mathrm{Te}}): \quad \frac{\mathbf{R} \lhd [\![g[e]]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![\varepsilon]\!]}{\mathbf{R} \lhd [\![\texttt{while } [e] \texttt{ do } Q]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![Q; \texttt{while } [e] \texttt{ do } Q]\!]}$$

$$(\mathrm{WHILE}^{\mathrm{F}}): \quad \frac{\mathcal{FF}(g[e])}{\mathbf{R} \lhd [\![\texttt{while } [e] \texttt{ do } Q]\!] \xrightarrow{\sigma} \mathbf{R} \lhd [\![\varepsilon]\!]}$$

$$(\mathrm{REQ}_{\mathrm{Te}}): \quad \frac{\mathbf{R} \lhd [\![g[e]]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![\varepsilon]\!]}{\mathbf{R} \lhd [\![\texttt{req}[e]]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![\varepsilon]\!]}$$

2.3.9 Repeat n-times

Assume that n is an integer constant. We may then define the repetition command, `do`, that repeats a body n times.

$$(\mathrm{DO}_{\mathrm{n}}): \quad \frac{}{\mathbf{R} \lhd [\![\texttt{do } n \ P]\!] \xrightarrow{\tau} \mathbf{R} \lhd [\![P; \texttt{do } (n-1) \ P]\!]} \ (n > 0)$$

$$(\mathrm{DO}_0): \quad \frac{}{\mathbf{R} \lhd [\![\texttt{do } n \ P]\!] \xrightarrow{\tau} \mathbf{R} \lhd [\![\varepsilon]\!]} \ (n \leqslant 0)$$

2.4 Example derivations

We now give some examples to show how transitions may be derived by the presented system.

Example 1: Jobshop

This example refers to the '`jobshop.d2000`' program supplied with Demos. The code is given in full in Figure 2.3. We only show one of the numerous possible derivation sequences, and omit the derivations that go through the instantiation of the entities, resources and bins [see Example 3]. Hammers shall be written h and mallets m.

We shall first introduce some derived rules to make our derivations slightly more succinct. The first allows us to skip some explanation of how we execute the parallel composition of sequential processes.

$$\dfrac{\dfrac{\mathbf{R} \lhd [\![P]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P']\!]}{\mathbf{R} \lhd [\![P; Q]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P'; Q]\!]}\,(\mathrm{COMP}_{\mathrm{I},1})}{\mathbf{R} \lhd [\![P; Q \| P_2]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P'; Q \| P_2]\!]}\,(\mathrm{PAR}_{\mathrm{I},1})$$

$$\equiv$$

$$\dfrac{\mathbf{R} \lhd [\![P]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P']\!]}{\mathbf{R} \lhd [\![P; Q \| P_2]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P'; Q \| P_2]\!]}\,(\mathrm{PCOMP}^2_1)$$

etc.

The second simplifies how we can write a simple 'get' from a bin.

$$\dfrac{\dfrac{}{\mathbf{R}, b \lhd [\![\mathtt{getB}(b, 1)]\!] \xrightarrow{\sigma} \mathbf{R} \lhd [\![\varepsilon]\!]}\,(\mathrm{GET}^{\mathrm{B}}_1)}{\mathbf{R}, b \lhd [\![g[\mathtt{getB}(b, 1)]]\!] \xrightarrow{\sigma} \mathbf{R} \lhd [\![\varepsilon]\!]}\,(\mathrm{NBLOCK}^{\mathrm{T}}_{\mathrm{B}})$$

$$\equiv$$

$$\dfrac{}{\mathbf{R}, b \lhd [\![g[\mathtt{getB}(b, 1)]]\!] \xrightarrow{\sigma} \mathbf{R} \lhd [\![\varepsilon]\!]}\,(\mathrm{NBLOCK}^{1}_{\mathrm{B}})$$

etc.

We commence by showing how the while-loop of the worker process is expanded, and how a bin item is placed in the available resource collection.

$$\dfrac{\dfrac{\dfrac{}{\wr m, h \int \lhd [\![g[]]\!] \xrightarrow{\tau} \wr m, h \int \lhd [\![\varepsilon]\!]}\,(\mathrm{NBLOCK}^{\mathrm{T}})}{\wr m, h \int \lhd [\![W]\!] \xrightarrow{\tau} \wr m, h \int \lhd [\![\mathtt{putB}(\mathit{easy}, 1); \ldots; W]\!]}\,(\mathrm{WHILE}^{\mathrm{T}}_{\mathrm{Te}})}{\wr m, h \int \lhd [\![J \| J \| W]\!] \xrightarrow{\tau} \wr m, h \int \lhd [\![J \| J \| \mathtt{putB}(\mathit{easy}, 1); \ldots; W]\!]}\,(\mathrm{PAR}_{\mathrm{I},3})$$

$$\dfrac{\dfrac{}{\wr m, h \int \lhd [\![\mathtt{putB}(\mathit{easy}, 1)]\!] \xrightarrow{\mathrm{putB}} \wr m, h \int \lhd [\![\varepsilon]\!]}\,(\mathrm{GET}^{1}_{\mathrm{B}})}{\wr m, h \int \lhd [\![J \| J \| \mathtt{putB}(\mathit{easy}, 1); \ldots; W]\!] \xrightarrow{\mathrm{putB}} \wr m, h, \mathit{easy} \int \lhd [\![J \| J \| \mathtt{hold}(1); \ldots; W]\!]}\,(\mathrm{PCOMP}^3_3)$$

where J is the code of the `Jobber` class, W is the code of the `Worker` class, and

$$K \equiv \mathtt{try}\ [\mathtt{getB}(\mathit{easy}, 1)]\ \mathtt{then\ trace('Easy')\ etry}\ [\ldots]\ \mathtt{then}\ \ldots; J.$$

```
Class Jobber=
{While []
 {try [getB(easy,1)] then {Trace("Doing easy");hold(5);}
  etry [getB(mid,1)]
   then {try [getR(mallet,1)]
           then {hold(20);
                 trace("Done mid");
                 putR(mallet,1);
                 }
         etry [getR(hammer,1)]
           then {hold(10);
                 trace("Done mid");
                 putR(hammer,1);
                 }
        }
 etry [getB(hard,1)]
  then {getR(hammer,1);
        hold(20);
        trace("Done hard");
        putR(hammer,1);
       }
  }
}

Class Work=
{while []
 {putB(easy,1);
  hold(1);
  putB(hard,1);
  hold(1);
  putB(mid,1);
  hold(25);
 }
}

Res(mallet,1);
Res(hammer,1);
Bin(easy,0);
Bin(mid,0);
Bin(hard,0);
Entity(J1,Jobber,0);
Entity(J2,Jobber,0);
Entity(W,Work,0);
Hold(200);
Hold(0);
Close;
```

Figure 2.3: Jobber soruce code

We can now silently remove the `hold` from the head of the worker process, and expand one of the Jobber's while loops.[5]

$$\dfrac{\dfrac{\dfrac{}{m,h,easy \lhd [\![\texttt{hold}(1);\ldots]\!] \xrightarrow{\tau} m,h,easy \lhd [\![\texttt{putB}(hard,1);\ldots]\!]}\,(\text{NOTIME})}{m,h,easy \lhd [\![J \| \texttt{hold}(1);\ldots]\!] \xrightarrow{\tau} m,h,easy \lhd [\![J \| \texttt{putB}(hard,1);\ldots]\!]}\,(\text{PAR}_{\text{I},2})}{m,h,easy \lhd [\![J \| J \| \texttt{hold}(1);\ldots]\!] \xrightarrow{\tau} m,h,easy \lhd [\![J \| J \| \texttt{putB}(hard,1);\ldots]\!]}\,(\text{PAR}_{\text{I},2})$$

$$\dfrac{\dfrac{\dfrac{}{m,h,easy \lhd [\![g[]]\!] \xrightarrow{\tau} m,h,easy \lhd [\![]\!]}\,(\text{NBLOCK}^{\text{T}})}{m,h,easy \lhd [\![J]\!] \xrightarrow{\tau} m,h,easy \lhd [\![K]\!]}\,(\text{WHILE}^{\text{T}}_{\text{Te}})}{m,h,easy \lhd [\![J \| J \| \texttt{putB}(hard,1);\ldots]\!] \xrightarrow{\tau} m,h,easy \lhd [\![K \| J \| \texttt{putB}(hard,1);\ldots]\!]}\,(\text{PAR}_{\text{I},1})$$

Thus we could write

$$\wr m,h \int \lhd [\![J \| J \| W]\!] \overset{\text{putB}}{\Longrightarrow} \wr m,h,easy \int \lhd [\![K \| J \| \texttt{putB}(hard,1);\ldots;W]\!]$$

[5] We shall adopt a syntactic equivalence that allows us to drop the brackets denoting multiset.

The Jobber process with its `while` loop expanded is now in a position to start a job. The only type of job that it is able to get is an easy one.

$$\dfrac{\dfrac{}{m, h, easy \triangleleft [\![g[\mathtt{getB}(easy, 1)]]\!] \xrightarrow{\mathrm{getB}} m, h \triangleleft [\![]\!]}\,(\mathrm{NBLOCK}^1_{\mathrm{B}})}{\dfrac{m, h, easy \triangleleft [\![\mathtt{try}\ [\mathtt{getB}(easy, 1)]\ \mathtt{then\ trace('Easy')\ etry}\ [\ldots]\ \mathtt{then}\ \ldots]\!] \xrightarrow{\mathrm{getB}} m, h \triangleleft [\![\mathtt{trace('Easy')}]\!]}\,(\mathrm{PCOMP}^1_3)}{m, h, easy \triangleleft [\![K \| J \| \mathtt{putB}(hard, 1); \ldots]\!] \xrightarrow{\mathrm{getB}} m, h \triangleleft [\![\mathtt{trace('Easy')}; J \| J \| \mathtt{putB}(hard, 1); \ldots]\!]}\,(\mathrm{TRY}^{\mathrm{T}}_{\mathrm{Te}})$$

We now execute the `trace` command — it performs no visible resource action, so it is silent.

$$\dfrac{\dfrac{}{m, h \triangleleft [\![\mathtt{trace('Easy')}]\!] \xrightarrow{\tau} m, h \triangleleft [\![]\!]}\,(\mathrm{TRACE})}{m, h \triangleleft [\![\mathtt{trace('Easy')}; J \| J \| \mathtt{putB}(hard, 1); \ldots]\!] \xrightarrow{\tau} m, h \triangleleft [\![J \| J \| \mathtt{putB}(hard, 1); \ldots]\!]}\,(\mathrm{PCOMP}^1_3)$$

Another Jobber's `while`-loop is expanded.

$$D_4 : \dfrac{\dfrac{\dfrac{}{m, h \triangleleft [\![g[]]\!] \xrightarrow{\tau} m, h \triangleleft [\![]\!]}\,(\mathrm{NBLOCK}^{\mathrm{T}})}{m, h \triangleleft [\![J]\!] \xrightarrow{\tau} m, h \triangleleft [\![K]\!]}\,(\mathrm{WHILE}^{\mathrm{T}}_{\mathrm{Te}})}{m, h \triangleleft [\![J \| J \| \mathtt{putB}(hard, 1); \ldots]\!] \xrightarrow{\tau} m, h \triangleleft [\![K \| J \| \mathtt{putB}(hard, 1); \ldots]\!]}\,(\mathrm{PAR}^3_1)$$

The Worker process releases a hard job.

$$\dfrac{\dfrac{\dfrac{}{m, h \triangleleft [\![\mathtt{putB}(hard, 1)]\!] \xrightarrow{\mathrm{putB}} m, h, hard \triangleleft [\![]\!]}\,(\mathrm{PUT}^1_{\mathrm{B}})}{m, h \triangleleft [\![\mathtt{putB}(hard, 1); \ldots]\!] \xrightarrow{\mathrm{putB}} m, h, hard \triangleleft [\![\mathtt{hold}(1); \ldots]\!]}\,(\mathrm{COMP}_{\mathrm{Te}})}{m, h \triangleleft [\![K \| J \| \mathtt{putB}(hard, 1); \ldots]\!] \xrightarrow{\mathrm{putB}} m, h, hard \triangleleft [\![K \| J \| \mathtt{hold}(1); \ldots]\!]}\,(\mathrm{PAR}^3_3)$$

The `hold`(1) at the head of the Worker process is now removed.

$$\dfrac{\dfrac{\dfrac{}{m,h,hard \triangleleft [\![\texttt{hold}(1)]\!] \xrightarrow{\tau} m,h,hard \triangleleft [\![]\!]}\,(\text{NOTIME})}{m,h,hard \triangleleft [\![\texttt{hold}(1);\ldots]\!] \xrightarrow{\tau} m,h,hard \triangleleft [\![\texttt{putB}(mid,1);\ldots]\!]}\,(\text{COMP}_{\text{Te}})}{m,h,hard \triangleleft [\![K \| J \| \texttt{hold}(1);\ldots]\!] \xrightarrow{\tau} m,h,hard \triangleleft [\![K \| J \| \texttt{putB}(mid,1);\ldots]\!]}\,(\text{PAR}^3_3)$$

The Worker process now releases a medium-difficulty job.

$$\dfrac{\dfrac{}{m,h,hard \triangleleft [\![\texttt{putB}(mid,1)]\!] \xrightarrow{\text{putB}} m,h,hard,mid \triangleleft [\![]\!]}\,(\text{PUT}^1_{\text{B}})}{m,h,hard \triangleleft [\![K \| J \| \texttt{putB}(mid,1);\ldots]\!] \xrightarrow{\text{putB}} m,h,hard,mid \triangleleft [\![K \| J \| \texttt{hold}(25);\ldots]\!]}\,(\text{PCOMP}^3_3)$$

By a derivation similar to D_4,

$$m,h,hard,mid \triangleleft [\![K \| J \| \texttt{hold}(25);\ldots]\!] \xrightarrow{\tau} m,h,hard,mid \triangleleft [\![K \| K \| \texttt{hold}(25);\ldots]\!]$$

A process now tries to get an easy job, fails, and so gets a medium-difficulty job.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathit{FF}(m,h,hard,mid \triangleleft [\![g[\texttt{getB}(easy,1)]]\!])}\;\wr easy \int \notin \wr m,h,hard,mid \int \qquad \dfrac{}{m,h,hard,mid \triangleleft [\![g[\texttt{getB}(mid,1)]]\!] \xrightarrow{\text{getB}} m,h,hard \triangleleft [\![]\!]}\,(\text{NBLOCK}^1_{\text{B}})}{m,h,hard,mid \triangleleft [\![\texttt{try}\ [\texttt{getB}(easy,1)]\ \texttt{then}\ \ldots\ \texttt{etry}\ [\texttt{getB}(mid,1)]\ \texttt{then}\ \ldots]\!] \xrightarrow{\text{getB}} m,h,hard \triangleleft [\![\texttt{try}\ [\texttt{getR}(m,1)]\ \texttt{then}\ \ldots\ \texttt{etry}\ [\texttt{getR}(h,1)]\ \texttt{then}\ \ldots]\!]}\,(\text{TRY}^{\text{F}}_{\text{Te}})}{m,h,hard,mid \triangleleft [\![K \| K \| \texttt{hold}(25);\ldots]\!] \xrightarrow{\text{getB}} m,h,hard \triangleleft [\![\texttt{try}\ [\texttt{getR}(m,1)]\ \texttt{then}\ \{\texttt{hold}(20);\texttt{trace('Mid')}\}\ \texttt{etry}\ [\texttt{getR}(h,1)]\ \texttt{then}\ \ldots;J \| K \| \texttt{hold}(25);\ldots]\!]}\,(\text{PCOMP}^1_3)$$

We build a derivation tree similar to the one above (other than there is no failure) to show the same Jobber process getting the mallet it needs to complete its medium-difficulty job.

$$\begin{array}{l} m, h, hard \triangleleft [\![\texttt{try } [\texttt{getR}(m, 1)] \texttt{ then } \{\texttt{hold}(20); \texttt{trace('Mid')}\} \texttt{ etry } [\texttt{getR}(h, 1)] \texttt{ then } \ldots; J \| K \| \texttt{hold}(25); \ldots]\!] \\ \xrightarrow{\text{getR}} h, hard \triangleleft [\![\texttt{hold}(20); \texttt{trace('Mid')}; J \| K \| \texttt{hold}(25); \ldots]\!] \end{array},$$

from which we obtain

$$h, hard \triangleleft [\![\texttt{hold}(20); \texttt{trace('Mid')}; J \| K \| \texttt{hold}(25); \ldots]\!] \xrightarrow{\tau} \xrightarrow{\tau} h, hard \triangleleft [\![J \| K \| \texttt{hold}(25); \ldots]\!]$$

Example 2: Semaphore

This simple example considers a synchronisation-based semaphore and may be found in the Demos distribution as `me4.d2000`, though, for conciseness, we ignore the third user. The code is given again in Figure 2.4. Again, we omit the initial declations of entities *etc.*

We first define, for convenience:

$$\begin{array}{rcl} A & \equiv & \texttt{getS}(semS,1);\texttt{hold}(5);\texttt{trace('have')};\texttt{putS}(semS,1) \\ S & \equiv & \texttt{sync}(semS);\underbrace{\texttt{trace('back')}}_{Tr} \\ U & \equiv & \text{the code of } \texttt{User} \\ & \equiv & \texttt{while [] do } A \\ Sem & \equiv & \text{the code of } \texttt{Sem} \\ & \equiv & \texttt{while [] do } S \end{array}$$

We shall expand the while loops that control the processes.

$$D_1 : \quad \dfrac{\dfrac{\dfrac{}{\wr\int\triangleleft[\![g[]]\!] \xrightarrow{\tau} \wr\int\triangleleft[\![\varepsilon]\!]}(\text{NBLOCK}_\text{T})}{\wr\int\triangleleft[\![U]\!] \xrightarrow{\tau} \wr\int\triangleleft[\![A;U]\!]}(\text{WHILE}^\text{T}_\text{Te})}{\wr\int\triangleleft[\![U\|U\|Sem]\!] \xrightarrow{\tau} \wr\int\triangleleft[\![A;U\|U\|Sem]\!]}(\text{PAR}^3_1)$$

Very similar derivations justify

$$\wr\int\triangleleft[\![A;U\|U\|Sem]\!] \xrightarrow{\tau} \wr\int\triangleleft[\![A;U\|A;U\|Sem]\!]$$

and

$$\wr\int\triangleleft[\![A;U\|A;U\|Sem]\!] \xrightarrow{\tau} \wr\int\triangleleft[\![A;U\|A;U\|S;Sem]\!]\,.$$

$$D_2 : \quad \dfrac{\dfrac{}{\wr\int\triangleleft[\![S;Sem]\!] \xrightarrow{\text{sync}} \wr\overline{(\!|Tr;Sem|\!)}^{semS}\int\triangleleft[\![\varepsilon]\!]}(\text{SYNC}^\text{x})}{\wr\int\triangleleft[\![A;U\|A;U\|S;Sem]\!] \xrightarrow{\text{sync}} \wr\overline{(\!|Tr;Sem|\!)}^{semS}\int\triangleleft[\![A;U\|A;U]\!]}(\text{PAR}^2_3)$$

We shall now allow the left user process to take the semaphore:

$$D_3 : \quad \dfrac{\dfrac{}{\wr\overline{(\!|Tr;Sem|\!)}^{semS}\int\triangleleft[\![A;U]\!] \xrightarrow{\text{getS}} \wr\int\triangleleft\left[\!\!\left[\dfrac{\texttt{hold}(5);\ldots;U}{(\!|Tr;Sem|\!)_{semS}}\right]\!\!\right]}(\text{GETS}^\text{x})}{\wr\overline{(\!|Tr;Sem|\!)}^{semS}\int\triangleleft[\![A;U\|A;U]\!] \xrightarrow{\text{getS}} \wr\int\triangleleft\left[\!\!\left[\dfrac{\texttt{hold}(5);\ldots;U}{(\!|Tr;Sem|\!)_{semS}}\|A;U\right]\!\!\right]}(\text{PAR}^1_1)$$

As there are now no processes waiting in $semS$, the right user process cannot proceed, and so the critical region is not entered.

```
class sem=
{while []
 {sync(semS);
  Trace("Im back %n");
 }
}

Entity(theSem,sem,0);

class User=
{while []
 {getS(semS,1);
 hold(5);
 Trace("Have %s %n");
 putS(semS,1);
 }
}

Entity(U1,User,0);
Entity(U2,User,0);
Hold(50);
Entity(U3,User,0);
Hold(50);
HOLD(0);
CLOSE;
```

Figure 2.4: Semaphore source code

Following two τ-transitions to skip the `hold`(5); `trace('have')` part of the active user process, we may re-activate the semaphore process:

$$D_4 : \quad \dfrac{\dfrac{}{\wr\int\lhd\left[\!\!\left[\dfrac{\texttt{putS}(semS,1);U}{(\!|Tr;Sem|\!)_{semS}}\right]\!\!\right] \xrightarrow{\text{putS}} \wr\int\lhd[\![U\|Tr;Sem]\!]}(\text{PUTS}^{\text{x}})}{\wr\int\lhd\left[\!\!\left[\dfrac{\texttt{putS}(semS,1);U}{(\!|Tr;Sem|\!)_{semS}}\|A;U\right]\!\!\right] \xrightarrow{\text{putS}} \wr\int\lhd[\![U\|Tr;Sem\|A;U]\!]}(\text{PAR}^1_{\text{I}})$$

A τ-transition will remove the Tr part of the semaphore process (which is just a `trace` command).

We may derive all following states using similar derivation trees. So, starting again, we could obtain this sequence:

$$\begin{array}{lll}
\wr\int\lhd[\![U\|U\|Sem]\!] & \xrightarrow{\tau} & \wr\int\lhd[\![A;U\|U\|Sem]\!] \\
& \xrightarrow{\tau} & \wr\int\lhd[\![A;U\|A;U\|Sem]\!] \\
& \xrightarrow{\text{sync}} & \wr\int\lhd[\![A;U\|A;U\|S;Sem]\!] \\
& \xrightarrow{\text{getS}} & \wr\dfrac{semS}{(\!|Tr;Sem|\!)}\int\lhd[\![A;U\|A;U]\!] \\
& \xrightarrow{\text{putS}} & \wr\int\lhd\left[\!\!\left[\dfrac{\texttt{hold}(5);\ldots;U}{(\!|Tr;Sem|\!)_{semS}}\|A;U\right]\!\!\right] \\
& \xrightarrow{\tau,\tau} & \wr\int\lhd\left[\!\!\left[\dfrac{\texttt{putS}(semS,1);U}{(\!|Tr;Sem|\!)_{semS}}\|A;U\right]\!\!\right] \\
& \xrightarrow{\tau} & \wr\int\lhd[\![U\|A;U\|Tr;Sem]\!] \\
& \xrightarrow{\tau} & \wr\int\lhd[\![U\|A;U\|Sem]\!] \\
& \xrightarrow{\tau} & \wr\int\lhd[\![U\|A;U\|S;Sem]\!] \\
& \xrightarrow{\text{sync}} & \wr\dfrac{semS}{(\!|Tr;Sem|\!)}\int\lhd[\![U\|A;U]\!] \\
& \xrightarrow{\text{getS}} & \wr\int\lhd\left[\!\!\left[U\|\dfrac{\texttt{hold}(5);\ldots;U}{(\!|Tr;Sem|\!)_{semS}}\right]\!\!\right] \\
& \xrightarrow{\text{putS}} & \wr\int\lhd\left[\!\!\left[A;U\|\dfrac{\texttt{hold}(5);\ldots;U}{(\!|Tr;Sem|\!)_{semS}}\right]\!\!\right] \\
& \xrightarrow{\tau,\tau} & \wr\int\lhd\left[\!\!\left[A;U\|\dfrac{\texttt{putS}(semS,1));U}{(\!|Tr;Sem|\!)_{semS}}\right]\!\!\right] \\
& \vdots &
\end{array}$$

Example 3: Deadlock

This example shows the transition system becoming deadlocked and gives a suggestion of how this may be expressed in the process logic. This time, we shall show processes and resources being created.

The code for this (simple) example may be found in Figure 2.5. In the following derivations, we shall abbreviate the complete code of `P1` by $P_1 \equiv \texttt{getR}(a,1);Q_1$ and of `P2` by $P_2 \equiv \texttt{getR}(b,1);Q_2$.

```
Res(a,1);
Res(b,1);

Class P1={
  getR(a,1);
  getR(b,1);
  hold(5);
  putR(a,1);
  putR(b,1);
}

class P2={
  getR(b,1);
  getR(a,1);
  hold(6);
  putR(a,1);
  putR(b,1);
}

Entity(P1,P1,0); Entity(P2,P2,0);
```

Figure 2.5: Deadlock source code

$$D_1 : \quad \frac{\overline{\wr \int \lhd [\![\texttt{Res}(a,1)]\!] \xrightarrow{\tau} \wr a \int \lhd [\![]\!]}\,(\mathrm{RES}_1)}{\wr \int \lhd [\![\texttt{Res}(a,1); \texttt{Res}(b,1); \ldots]\!] \xrightarrow{\tau} \wr a \int \lhd [\![\texttt{Res}(b,1); \ldots]\!]}\,(\mathrm{COMP}_\mathrm{T})$$

Similarly,

$$\wr a \int \lhd [\![\texttt{Res}(b,1); \texttt{Entity}(P1,0); \ldots]\!] \xrightarrow{\tau} \wr a, b \int \lhd [\![\texttt{Entity}(P1,0); \ldots]\!]$$

$$D_2 : \quad \frac{}{\wr a, b \int \lhd [\![\texttt{Entity}(P1,0); \texttt{Entity}(P2,0)]\!] \xrightarrow{\tau} \wr a, b \int \lhd [\![P_1 \| \texttt{Entity}(P2,0)]\!]}\,(\mathrm{ENTITY}_\mathrm{I})$$

$$D_3 : \quad \frac{\overline{\wr a, b \int \lhd [\![\texttt{Entity}(P2,0)]\!] \xrightarrow{\tau} \wr a, b \int \lhd [\![P_2]\!]}\,(\mathrm{ENTITY}_\mathrm{T})}{\wr a, b \int \lhd [\![P_1 \| \texttt{Entity}(P2,0)]\!] \xrightarrow{\tau} \wr a, b \int \lhd [\![P_1 \| P_2]\!]}\,(\mathrm{PAR}_\mathrm{I})$$

$$D_4 : \quad \frac{\overline{\wr a, b \int \lhd [\![P_1]\!] \xrightarrow{\mathrm{getR}} \wr b \int \lhd [\![Q_1]\!]}\,(\mathrm{GETR}_1)}{\wr a, b \int \lhd [\![P_1 \| P_2]\!] \xrightarrow{\mathrm{getR}} \wr, b \int \lhd [\![Q_1 \| P_2]\!]}\,(\mathrm{COMP}_\mathrm{I})$$

$$D_5 : \quad \frac{\overline{\wr b \int \lhd [\![P_2]\!] \xrightarrow{\mathrm{getR}} \wr \int \lhd [\![Q_2]\!]}\,(\mathrm{GETR}_1)}{\wr b \int \lhd [\![Q_1 \| P_2]\!] \xrightarrow{\mathrm{getR}} \wr \int \lhd [\![Q_1 \| Q_2]\!]}\,(\mathrm{COMP}_\mathrm{I})$$

At this stage, we would not be able to derive any σ-transition. As we still have code left to execute (formally, $[\![Q_1 \| Q_2]\!] \neq [\![\varepsilon]\!]$ — we have not terminated cleanly), we can deduce that the system has deadlocked.

Chapter 3

Correctness of transition system

3.1 Introduction

We now consider various properties of the transition system presented above. These properties relate to the correctness of the system with respect to the operational semantics of Demos [TB01].

The latest version of Demos, Demos 2000, has a large command set. Pragmatism leads us to consider a small subclass of Demos 2000 programs — as we did above by not considering variables or value-resources — namely, those written in πDemos with the Demos 2000 delaration format. πDemos is defined, with its operational semantics, in [BT93]. Specifically, this avoids the notational burden of considering synchronization between processes and increases the brevity of our proofs.

The properties considered shall relate to the validity (correctness) of our transition system. As shall be discussed, we shall not be able to demonstrate the soundness of the transition system with respect to the operational semantics in the familiar sense. This is to be expected: we intentionally made the transition system non-time sensitive, and so transitions derived may not be possible according to the temporal (time-based) constraints actually in place.

πDemos allows us to consider the essential aspect of Demos programs: the interaction of processes through shared resources. In essence, there are three main commands:

getR The command `getR`$(r, 1)$ acquires one item of r from the globally available resources, thus making it unavailable to other processes. If r is not available at the time of request, the process will wait until it is available.

putR The command `getR`$(r, 1)$ releases one item of r, making it available to all other processes. If a process is waiting for the release of r, it will be activated. Otherwise, it will be placed in the resource pool.

hold The command `hold`(t) signifies that the process waits T time units before resuming execution.

We shall assume that processes have been initialised correctly; failing to make this assumption adds little to the presentation. Therefore, we will not consider `Entity` *etc.*

State in the operational semantics is represented by a 3-tuple:

$$S = \langle \mathcal{EL}, \mathcal{RL}, \Sigma \rangle.$$

- Σ is a function that records the various syntactic definitions found within the system. It will contain, for example, class definitions and resource labels. As we are not going to consider any commands that manipulate Σ (*e.g.,* `class` and `cons`), we shall frequently omit it when discussing state.
- $\mathcal{RL}$ is the resource list, a list of 3-tuples $\langle id, avail, \mathcal{WL} \rangle$. id is the resource name or label; $avail \in \{\mathtt{tt}, \mathtt{ff}\}$ is a boolean indicating whether the resource is taken; and $\mathcal{WL}$ is a list of processes waiting on the resource, non-empty only if not $avail$.
- $\mathcal{EL}$ is the event list, $[(P, \mathrm{PD}(code, attrs, t))]$. It is a list of processes not waiting on any resource. Processes are represented by a pair comprising a label and the process descriptor. The process descriptor is another 3-tuple, comprising the code remaining to be executed, the resources held by the process, and the simulation time at which the next command is to be executed.

The reader is referred to [BT93] for the formal presentation of the operational semantics.

In the sequel, to match the account presented in [BT93], we shall only consider one-item get / put, with each identifier in the resource list being unique (that is, only one of any resource shall exist).

3.2 Simulation

We shall represent the globally *available* resources in the transition system by $\mathbf{R}$. Process terms shall be bracketed $[\![P]\!]$. The state of a program in the transition system is of the form $\mathbf{R} \triangleleft [\![P]\!]$; an equivalent representation in the operational semantics shall be written $f(\mathbf{R} \triangleleft [\![P]\!])$. Transitions derived according to the operational semantics shall be denoted $\Longrightarrow$ (note that this is unlabelled, to distinguish it from observable transitions made in σDemos); as usual, single-step transitions in the transition system shall be denoted $\xrightarrow{\sigma}$. Explicitly observable actions shall be denoted $\overset{\sigma}{\Longrightarrow}$.

Our transition system is non-temporal — it does not take into account any times inside the Demos program. So, for example, it will completely ignore synchronisation by any `hold` statement. Consequently, it will consider the execution of commands on resources in any and every order, including those that are impossible according to the operational semantics of the program, which does take into account time.

We can see, therefore, that for the full Demos system we do not have the following:

$$\text{If } \mathbf{R} \triangleleft [\![P]\!] \xrightarrow{\sigma} \mathbf{R}' \triangleleft [\![P']\!] \text{ then } f(\mathbf{R} \triangleleft [\![P]\!]) \Longrightarrow f(\mathbf{R} \triangleleft [\![P']\!])$$

for some (justified — *i.e.,* not something vacuous that makes the above trivially true) function f mapping transition system states to operational semantics states.

We may, however, consider the converse: does our transition system *simulate* the operational semantics? That is, do we have the property:

$$\text{If } S \Longrightarrow S' \text{ then } g(S) \xrightarrow{\sigma}{}^{*} g(S'),$$

where g maps states in the operational semantics to states in the transition system and $\xrightarrow{\sigma}{}^{*}$ is the reflexive, transitive closure of $\xrightarrow{\sigma}$ (the effect of running an arbitrary number of steps, including τ-transitions), $\sigma \in Act$.

In order to demonstrate such a relationship, we must define the map g between states in the operational semantics and states in the transition system. Firstly, though, to aid clarity of expression, we shall introduce a new syntactic abbreviation for the parallelisation operator that operates outside the semantic brackets:

$$\begin{array}{rcl} [\![P_1]\!] \mid [\![P_2]\!] & \equiv & [\![P_1 \| P_2]\!] \\ [\![P_1]\!] \mid [\![\varepsilon]\!] & \equiv & [\![P_1]\!] \end{array}$$

We shall define g through three auxilliary functions.

- g_E shall convert processes in the event list to transition system processes.

$$\begin{array}{rcl} g_E([]) & \stackrel{\text{def}}{=} & [\![\varepsilon]\!] \\ g_E((P, \mathrm{PD}([], [], t)) : \mathcal{EL}) & \stackrel{\text{def}}{=} & g_E(\mathcal{EL}) \\ g_E((P, \mathrm{PD}([], attrs, t)) : \mathcal{EL}) & \stackrel{\text{def}}{=} & [\![\varepsilon]\!] \\ g_E((P, \mathrm{PD}(b : body, attrs, t)) : \mathcal{EL}) & \stackrel{\text{def}}{=} & \begin{cases} [\![\varepsilon]\!], & \text{if } b = \texttt{close} \\ [\![b; body_{[P]}]\!] \mid g_E(\mathcal{EL}), & \text{otherwise} \end{cases} \end{array}$$

- g_W shall convert lists of waiting (blocked) processes to processes of the transition system.

$$\begin{array}{rcl} g_W(r, []) & \stackrel{\text{def}}{=} & [\![\varepsilon]\!] \\ g_W((r, (P, PD(body, attr, t))) : \mathcal{WL}) & \stackrel{\text{def}}{=} & [\![\texttt{getR}(r, 1); body_{[P]}]\!] \mid g_W(\mathcal{WL}) \end{array}$$

- To convert the resource list, $[\langle ident, avail, waiting]\rangle] : [\langle R, \mathbb{B}, [Proc]\rangle]$, we shall define $g_R : [\langle R, \mathbb{B}, [Proc]\rangle] \to \langle \mathcal{R}, [\![\mathcal{P}_{\mathrm{ROG}}]\!]\rangle$; the operator $\oplus$ shall combine these pairs:

$$\langle \mathbf{R}, [\![P_{[\ell_1]}]\!]\rangle \oplus \langle \mathbf{S}, [\![Q_{[\ell_2]}]\!]\rangle \stackrel{\text{def}}{=} \langle \mathbf{R} \uplus \mathbf{S}, [\![P_{[\ell_1]}]\!] \mid [\![Q_{[\ell_2]}]\!]\rangle$$

$$
\begin{aligned}
g_R([]) &\stackrel{\text{def}}{=} \langle \wr\wr, [\![\varepsilon]\!] \rangle \\
g_R(\langle r, \mathtt{tt}, [] \rangle : \mathcal{RL}) &\stackrel{\text{def}}{=} \langle \wr r \wr, [\![\varepsilon]\!] \rangle \oplus g_R(\mathcal{RL}) \\
g_R(\langle r, \mathtt{ff}, W \rangle : \mathcal{RL}) &\stackrel{\text{def}}{=} \langle \wr\wr\ , g_W(W) \rangle \oplus g_R(\mathcal{RL})
\end{aligned}
$$

We may now define our map between states in the two systems:

$$g(\mathcal{EL}, \mathcal{RL}, \Sigma) = \mathbf{R} \triangleleft g_E(\mathcal{EL})|W,$$

where $g_R(\mathcal{RL}) = \langle \mathbf{R}, W \rangle$.

Theorem 3.2.1 Let $S = \langle \mathcal{EL}, \mathcal{RL} \rangle$ be a reachable state in a πDemos program which evolves in one step of the operational semantics to $S' = \langle \mathcal{EL}', \mathcal{RL}' \rangle$. Then this is simulated in the transition system. That is, for some series of σ,

$$\text{If } S \Longrightarrow S' \text{ then } g(S) \xrightarrow{\sigma}{}^{*} g(S')$$

PROOF *By analysis of the cases of next command to be executed (the command at the head of the event list).*
We may assume, without loss of generality, that the choice of the transition to make is decided solely by the command at the head of the event list. Generality is not lost as this provides an arbitrary selection from the processes with equal (but still the least) event time, assuming insertion order is arbitrary where equal event times occur.

The event list is empty:

We have $S \equiv \langle [], \mathcal{RL} \rangle$. According to the operational semantics, the program halts with an error. As there is no next state, the statement is vacuously true.

Lead object has no actions left:

We consider two cases:

$attrs \neq []$:

Then the program halts with an error, so, again, the statement is vacuously true.

$attrs = []$:

Then $S \equiv ((P, \mathrm{PD}([], [], t)) : \mathcal{EL}, \mathcal{RL})$. The operational semantics gives $S' \equiv (\mathcal{EL}, \mathcal{RL})$. By the definition of g_E, $g(S) = g(S')$, and, as $\xrightarrow{\sigma}{}^{*}$ is reflexive, we have $g(S) \xrightarrow{\sigma}{}^{*} g(S')$ as required.

`close`:

Then $S \equiv ((P, \mathrm{PD}(\texttt{close} : body, attrs, t) : \mathcal{EL}, \mathcal{RL}))$. The operational semantics tells us that the program halts and that there is no next state, so the statement is vacuously true.

`hold`(t):

Then we have $S \equiv ((P, \mathrm{PD}(\texttt{hold}(t) : body, attrs, evt)) : \mathcal{EL}, \mathcal{RL})$. By applying the rule for `hold`, we obtain $S' \equiv (\mathcal{EL}', \mathcal{RL})$ where $\mathcal{EL}'$ is the same

as the original event list with t added to evt (and, consequently, causing a re-order of the event list) and $\texttt{hold}(t)$ removed.

Now, $g(S) = \mathbf{R} \lhd [\![\texttt{hold}(t); body]\!] \mid g_E(\mathcal{EL}) \mid W$ where $g_R(\mathcal{RL}) = \langle \mathbf{R}, W \rangle$. Similarly, we have $g(S') = \mathbf{R} \lhd g_E(\mathcal{EL}) \mid [\![body]\!] \mid W$, where $g_R(\mathcal{RL}) = \langle \mathbf{R}, W \rangle$. By the associativity and commutativity of parallelisation, this is just the same as the state derived by executing the hold in the transition system,

$$\mathbf{R} \lhd [\![\texttt{hold}(t); body]\!] \xrightarrow{\tau} \mathbf{R} \lhd [\![body]\!] \mid g_E(\mathcal{EL}) \mid W.$$

putR(r,1):

There are two subcases for this instruction:

Nothing is waiting on the resource:

Then $S \equiv ((P, \mathrm{PD}(\texttt{putR}(r,1); body, attrs, t)) : \mathcal{EL}, \mathcal{RL})$. After we have executed putR, we have state $S' \equiv ((P, \mathrm{PD}(body, attrs, t)) : \mathcal{EL}, \mathcal{RL}')$. We know that r was previously unavailable, so (without loss of generality thanks to the order of $\mathcal{RL}$ being arbitrary), $\mathcal{RL} = (r, \mathtt{ff}, []) : \mathcal{RL}''$, and $\mathcal{RL}' = (r, \mathtt{tt}, []) : \mathcal{RL}''$.

Now, $g(S) = \mathbf{R} \lhd [\![\texttt{putR}(r,1); body]\!] \mid g_E(\mathcal{EL}) \mid W$ where $g_R(\mathcal{RL}) = \langle W, \mathbf{R} \rangle$. With appropriate selection of parallelisation and sequential composition rules, we can derive the transition

$$\mathbf{R} \lhd [\![\texttt{putR}(r,1); body]\!] \mid g_E(\mathcal{EL}) \mid W \xrightarrow{\sigma} \mathbf{R}, r \lhd [\![body]\!] \mid g_E(\mathcal{EL}) \mid W;$$

the actual label, $\ell_1 : \mathrm{putR}^1_r$, is unimportant, so shall just call it σ. Consider $g(S') = \mathbf{R}' \lhd [\![body]\!] \mid g_E(\mathcal{EL}) \mid W'$: by the definition of g_R, because no processes were waiting on r, $W = W'$ and $\mathbf{R}' = \mathbf{R}, r$, so we have the the state obtained by the transition system.

There are processes waiting on the resource:

S is as before. After putR, we have $S' \equiv ((P, \mathrm{PD}(body, attrs', t)) : \mathcal{EL}', \mathcal{RL}')$. Assume, without loss of generality, that $\mathcal{RL} = (r, \mathtt{ff}, (P_1, \mathrm{PD}(body_1, attrs_1, t_1)) : W) : \mathcal{RL}''$. Then $\mathcal{RL}' = (r, \mathtt{ff}, W) : \mathcal{RL}''$. The operational semantics tells us that $\mathcal{EL}' = \mathcal{EL}$ with $(P_1, \mathrm{PD}(body_1, attrs_1, t_1))$ inserted at some point.

Now, $g(S) = \mathbf{R} \lhd [\![\texttt{putR}(r,1); body]\!] \mid g_E(\mathcal{EL}) \mid W$, where $g_R(\mathcal{RL}) = \langle \mathbf{R}, W \rangle$. By consideration of the definition of g_R, we see that $W = [\![\texttt{getR}(r,1); body_1]\!] \mid W'$, and so we have $g(S) = \mathbf{R} \lhd [\![\texttt{putR}(r,1); body]\!] \mid g_E(\mathcal{EL}) \mid [\![\texttt{getR}(r,1); body_1]\!] \mid W'$.

$$\begin{aligned} g(S) &\xrightarrow{\sigma} \mathbf{R}, r \lhd [\![body]\!] \mid g_E(\mathcal{EL}) \mid [\![\texttt{getR}(r,1); body_1]\!] \mid W' \\ &\xrightarrow{\sigma} \mathbf{R} \lhd [\![body]\!] \mid g_E(\mathcal{EL}) \mid [\![body_1]\!] \mid W' \end{aligned}$$

which is exactly (up to the associativity and commutativity of parallel process composition) $g(S')$.

getR(r,1) — **resource available**:

We have $S \equiv ((P, \mathrm{PD}(\texttt{getR}(r,1); body, attrs, t)) : \mathcal{EL}, \mathcal{RL})$. The resource is available, so $S' \equiv ((P, \mathrm{PD}(body, attrs', t)) : \mathcal{EL}, \mathcal{RL}')$. Without loss of generality (thanks to the arbitrary order or $\mathcal{RL}$ — it is a multiset rather

than a list), let $\mathcal{RL} = (r, \mathtt{tt}, []) : \mathcal{RL}''$, and so $\mathcal{RL}' = (r, \mathtt{ff}, []) : \mathcal{RL}''$ by the operational semantics.

Now, $g(S) = \mathbf{R} \triangleleft [\![body]\!] \mid g_E(\mathcal{EL}) \mid W$, where $g_R(\mathcal{RL}) = \langle \mathbf{R}, W \rangle$. Considering $g_R(\mathcal{RL})$, we notice that $\mathbf{R} = \mathbf{R}'', r$, and so $g(S) = \mathbf{R}'', r \triangleleft [\![body]\!] \mid g_E(\mathcal{EL}) \mid W$. Finally,

$$\mathbf{R}'', r \triangleleft [\![\mathtt{getR}(r, 1); body]\!] \mid g_E(\mathcal{EL}) \mid W \xrightarrow{\sigma} \mathbf{R}'' \triangleleft [\![body]\!] \mid g_E(\mathcal{EL}) \mid W,$$

which is exactly $g(S')$.

getR(r,1) — the resource is unavailable:

We have $S \equiv ((P, \mathrm{PD}(\mathtt{getR}(r, 1); body, attrs, t)) : \mathcal{EL}, \mathcal{RL})$. After executing $\mathtt{getR}(r, 1)$, we have $S' \equiv \langle \mathcal{EL}, \mathcal{RL}' \rangle$. Without loss of generality, let $\mathcal{RL} = (r, \mathtt{ff}, W) : \mathcal{RL}''$; hence, $\mathcal{RL}' = (r, \mathtt{ff}, W') : \mathcal{RL}''$ where W' is W with $(P, \mathrm{PD}(body, attrs, t))$ inserted at the appropriate point.

$g(S) = \mathbf{R} \triangleleft [\![\mathtt{getR}(r, 1); body]\!] \mid g_E(\mathcal{EL}) \mid V$, where $g_R(\mathcal{RL}) = \langle \mathbf{R}, V \rangle$. Notice that, by the definition of g_R, $g_R(\mathcal{RL}') = \langle \mathbf{R}, V \mid [\![\mathtt{getR}(r, 1); body]\!] \rangle$. Consequently, $g(S') = \mathbf{R} \triangleleft g_E(\mathcal{EL}) \mid V \mid [\![getRr1; body]\!]$, which, thanks to the associativity and commutativity of the parallel composition operator, is $g(S)$. Now, as $\xrightarrow{\sigma}{}^*$ by definition is reflexive, we have $g(S) \xrightarrow{\sigma} {}^* g(S)$, or, equivalently, $g(S) \xrightarrow{\sigma}{}^* g(S')$, as required.

□

As we stated earlier, we must also show that g faithfully translates states belonging to the operational semantics. A number of preliminaries must be covered first, so this is presented in §3.7.1.

3.3 Soundness of transitions

As we have already stated, we do not have the obvious soundness property stating that any transition derived by our transition system could have been derived according to the operational semantics.

The transition system was designed to take no heed of synchronisation by time, nor does it take into account the priority of processes. We should, though, be able to prove soundess with respect to the operational semantics with the following assumptions:

1. All event times are ignored; *i.e.,* $t = 0$, or some other arbitrary constant, for all PD entries.
2. All priorities are ignored.
3. The orders of the event list and waiting lists, consequently, are arbitrary.
4. When a resource is released, it is not necessarily a waiting process that acquires the resource — it could be either a process in a waiting list or a process about to enter a waiting list (*i.e.,* with `getR` at its head).

In order to prove the property, we firstly need to define the map from the transition system state to state in the operational semantics, $\bar{f}$. We use a number of auxiliary functions:

- h_P defines the resources held by a non-parallel process (note here that we assume that the programs passed are *well-formed*; that is, they only put resources that they own *etc.*). '\' denotes subtraction from a multiset.

$$\begin{aligned} h_P([\![\varepsilon]\!]) &\stackrel{\text{def}}{=} \{\!\} \\ h_P([\![\texttt{getR}(r,1);P]\!]) &\stackrel{\text{def}}{=} h_P([\![P]\!]) \setminus \{\!| r |\!\} \\ h_P([\![\texttt{putR}(r,1);P]\!]) &\stackrel{\text{def}}{=} h_P([\![P]\!]) \uplus \{\!| r |\!\} \end{aligned}$$

- h shall be the multiset of all held resources:

$$\begin{aligned} h([\![P_1 \| P_2]\!]) &\stackrel{\text{def}}{=} h([\![P_1]\!]) \uplus h([\![P_2]\!]) \\ h([\![P]\!]) &\stackrel{\text{def}}{=} h_P([\![P]\!]), P \text{ not parallel} \end{aligned}$$

- w_R defines the multiset of $\langle$process, resource$\rangle$ pairs where the process is waiting on the resource

$$\begin{aligned} w_R([\![\varepsilon]\!], \mathbf{R}) &\stackrel{\text{def}}{=} \{\!\} \\ w_R([\![P \| Q]\!], \mathbf{R}) &\stackrel{\text{def}}{=} w_R(P) \uplus w_R(Q) \\ w_R([\![c;P]\!], \mathbf{R}) &\stackrel{\text{def}}{=} \begin{cases} \{\!| \langle [\![P]\!], r \rangle |\!\}, & \text{if } c = \texttt{getR}(r,1) \text{ and } r \notin \mathbf{R} \\ \{\!\}, & \text{otherwise} \end{cases} . \end{aligned}$$

- w is just the multiset of waiting processes (without the resource that they are waiting on):

$$w([\![P]\!], \mathbf{R}) \stackrel{\text{def}}{=} \{\!| [\![P]\!] \text{ s.t. } \langle \_, [\![P]\!] \rangle \in w_R([\![P]\!], \mathbf{R}) |\!\}$$

- *mset* simply splits parallel processes into a multiset of non-parallel processes:

$$\begin{aligned} mset([\![\varepsilon]\!]) &\stackrel{\text{def}}{=} \{\!\} \\ mset([\![P]\!]) &\stackrel{\text{def}}{=} \{\!| [\![P]\!] |\!\}, \text{where } [\![P]\!] \text{ not parallel} \\ mset([\![P_1 \| P_2]\!]) &\stackrel{\text{def}}{=} mset(P_1) \uplus mset(P_2) \end{aligned}$$

- The multiset of active processes is just the multiset of all processes minus the multiset of waiting processes.

$$a([\![P]\!], \mathbf{R}) \stackrel{\text{def}}{=} mset([\![P]\!]) \setminus w([\![P]\!], \mathbf{R})$$

- The set of processes waiting on a resource r is given by:

$$w_P(r) \stackrel{\text{def}}{=} \{\!| [\![P]\!] | \langle [\![P]\!], r \rangle \in w_R([\![P]\!], \mathbf{R}) |\!\}$$

We are now in a position to define our translation $\bar{f}$:

$$\bar{f}(\mathbf{R} \triangleleft [\![P]\!]) \stackrel{\text{def}}{=} \langle \mathcal{EL}, \mathcal{RL} \rangle$$

where

$$\begin{array}{rcl}
\mathcal{EL} & \stackrel{\text{def}}{=} & \left[(\ell, \mathrm{PD}(body, attrs, t)) \;\middle|\; \begin{array}{c} [\![body_{[\ell]}]\!] \Subset a([\![P]\!], \mathbf{R}) \wedge \\ attrs = h_P([\![body]\!]) \wedge t = 0 \end{array} \right] \\
\mathcal{RL} & \stackrel{\text{def}}{=} & \wr \langle r, \mathtt{tt}, [] \rangle \text{ s.t. } r \in \mathbf{R} \int \uplus \wr \langle r, \mathtt{ff}, \mathcal{W}_r \rangle \text{ s.t. } r \Subset h([\![P]\!]) \int \\
\mathcal{W}_r & = & [(\ell, PD([\![body]\!], h_P[\![body]\!], 0)) \text{ s.t. } [\![body_{[\ell]}]\!] \Subset w_P(r)]
\end{array}$$

We shall now proceed with the body of the proof, which shall be by induction on the structure of σDemos processes.

Theorem 3.3.1 The transition system is sound with respect to a non-time and non-priority sensitive operational semantics. That is,

$$\text{If } \mathbf{R} \lhd [\![P]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P']\!] \text{ then } \bar{f}(\mathbf{R} \lhd [\![P]\!]) \Longrightarrow^* \bar{f}(\mathbf{R}' \lhd [\![P']\!]).$$

PROOF *By induction on the structure of σDemos processes P.*

$\underline{P \equiv \mathtt{hold}(t)}$:

TS	We may only derive the transition $\mathbf{R} \lhd [\![\mathtt{hold}(t)]\!] \xrightarrow{\tau} \mathbf{R} \lhd [\![\varepsilon]\!]$
Op sem	$\bar{f}(\mathbf{R} \lhd [\![P]\!]) = \langle [P, \mathrm{PD}([\mathtt{hold}(t)], attrs, 0)], \mathcal{RL} \rangle$. Executing the hold yields $\mathcal{EL}' = [(P, \mathrm{PD}([], attrs, 0))]$. Given the well-formedness of states, $attrs = []$, so another execution gives $\mathcal{EL}'' = []$, $\mathcal{RL}'' = \mathcal{RL}$. Notice that $\bar{f}(\mathbf{R} \lhd [\![\varepsilon]\!]) = \langle [], \mathcal{RL} \rangle = \langle \mathcal{EL}'', \mathcal{RL}'' \rangle$.

$\underline{P \equiv \mathtt{getR}(r, 1)}$:

TS	Assuming that we are able to make a transition, the resource must be available, so $\mathbf{R} \equiv \mathbf{R}'', r$, for some $\mathbf{R}''$. We have the transition $\mathbf{R}'', r \lhd [\![\mathtt{getR}(r, 1)]\!] \xrightarrow{\sigma} \mathbf{R}'' \lhd [\![\varepsilon]\!]$.
Op sem	Without loss of generatlity thanks to the arbitrary order of the resource list, because r is available we may assume that $\bar{f}(\mathbf{R} \lhd [\![P]\!]) = \langle [(P, \mathrm{PD}([\mathtt{getR}(r, 1)], attrs, 0))], [(r, \mathtt{tt}, [])] : \mathcal{RL}'' \rangle$. The operational semantics allows us to take the resource, yielding $\langle [(P, \mathrm{PD}([], [], 0))], [(r, \mathtt{ff}, [])] : \mathcal{RL}'' \rangle$. The 'exec' command allows us to take another step to $\langle [], [(r, \mathtt{ff}, [])] : \mathcal{RL}'' \rangle$. Assuming the well-formedness of the program (this atomic command in itself is not well-formed — we could, alternatively, have made this a special case in the COMP case below), this is $\bar{f}(\mathbf{R}'' \lhd [\![P'']\!])$.

$\underline{P \equiv \mathtt{putR}(r, 1)}$:

Similar — but we use assumption (4) to eliminate the possibility of a waiting process taking the resource straight away.

$\underline{P \equiv c; body}$:

c is a command and *body* some non-parallel process. Consider the cases for c.

$\underline{c \equiv \mathtt{hold}(t)}$:

TS	$P \equiv \texttt{hold}(t); body$. We may only derive the transition $\mathbf{R} \triangleleft [\![\texttt{hold}(t); body]\!] \xrightarrow{\tau} \mathbf{R} \triangleleft [\![body]\!]$.
Op sem	$\bar{f}(\mathbf{R} \triangleleft [\![\texttt{hold}(t); body]\!]) = \langle[(P, \mathrm{PD}([\texttt{hold}(t); body], attrs, 0))], \mathbf{R}\rangle$. The operational semantics yields, upon execution of the hold, $\langle[(P, \mathrm{PD}([body], attrs, 0))], \mathbf{R}\rangle$, which is exactly $\bar{f}(\mathbf{R} \triangleleft [\![body]\!])$

$c \equiv \texttt{getR}(r, 1)$ **or** $\texttt{putR}(r, 1)$:

As before, but we will not need to use the 'exec' rule to reduce down to an empty event list. (The consideration of getR above could be placed here instead of above — it was placed there to show how a proof that considered bins would be constructed)

$P \equiv P_1 \| P_2$:

Assume, without loss of generality, that $\mathbf{R} \triangleleft [\![P_1 \| P_2]\!] \xrightarrow{\sigma} \mathbf{R}' \triangleleft [\![P_1' \| Q]\!]$. We therefore have $\mathbf{R} \triangleleft [\![P_1]\!] \xrightarrow{\sigma} \mathbf{R}' \triangleleft [\![P_1']\!]$, so, by induction, $\bar{f}(\mathbf{R} \triangleleft [\![P_1]\!]) \Longrightarrow^* \bar{f}(\mathbf{R}' \triangleleft [\![P_1']\!])$. We must show that this execution matches the execution of $\bar{f}(\mathbf{R} \triangleleft [\![P_1 \| P_2]\!])$. P_2 is either waiting on a resource or it is not: if it is, it will be placed in the waiting list of some resource. It, therefore, will not affect the execution of this command. If P_2 is placed in the event list, thanks to the arbitrary order of the event list, it is possible that P_1's execution may be followed again. Moreover, this property is maintained through the execution of any intermediate steps made by P due to all times being equal.

□

3.4 Weak bisimulation

To consider some further properties, it will be useful to have a notion of equivalence between states; we shall adopt *observational equivalence* similar to that developed in [Mil89]. Observational equivalence between states, informally, tells us that any *visible* (in our case, ignoring τ-transitions) series of transitions from one of the states can be made if, and only if, it can be made from the other.

More formally, writing Act for the set of all actions and Act^* for the set of observable actions (*i.e.,* $Act^* = Act \setminus \{\tau\}$):

Definition (Weak bisimulation)
Let $\star$ represent a relation between states in the transition system, *i.e.,* $\star \subseteq \Omega^2$. We shall use infix notation, writing $\mathbf{R} \triangleleft [\![P]\!] \star \mathbf{R} \triangleleft [\![Q]\!]$[1] rather than $(\mathbf{R} \triangleleft [\![P]\!], \mathbf{R} \triangleleft [\![Q]\!]) \in \star$. $\star$ is a *weak bisimulation* between states if:

1. for any transition $\mathbf{R} \triangleleft [\![P]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \triangleleft [\![P']\!]$ where $\sigma \in Act^*$, we have $\mathbf{R} \triangleleft [\![Q]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \triangleleft [\![Q']\!]$ and $\mathbf{R}' \triangleleft [\![P']\!] \star \mathbf{R}' \triangleleft [\![Q']\!]$; and
2. for any transition $\mathbf{R} \triangleleft [\![Q]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \triangleleft [\![Q']\!]$ where $\sigma \in Act^*$, we have $\mathbf{R} \triangleleft [\![P]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \triangleleft [\![P']\!]$ and $\mathbf{R}' \triangleleft [\![Q']\!] \star \mathbf{R}' \triangleleft [\![Q']\!]$.

[1] We shall not allow bisimulation between states with different resource sets at this stage, so if $\mathbf{R} \neq \mathbf{S}$, $\mathbf{R} \triangleleft [\![P]\!] \not\approx \mathbf{S} \triangleleft [\![Q]\!]$ for any P, Q.

Definition (Observational equivalence relation)
We shall use $\approx$ to denote observable equivalence. We write $\mathbf{R} \lhd [\![P]\!] \approx \mathbf{R} \lhd [\![Q]\!]$ if there is a weak bisimulation between these states. That is,

$$\approx \stackrel{\text{def}}{=} \bigcup \{\star \mid \star \text{is a bisimulation}\}$$

It may be that $\approx$ is a *congruence relation*. That is, for any process $\Psi[P]$ containing P as a sub-term, if $P \approx Q$ then $\Psi[P] \approx \Psi[Q]$ (where $\Psi[Q]$ is $\Psi[P]$ with P substituted for Q). We do not need to prove this property, but do require that if two states are observationally equivalent and we create the parallel composition of each of them with a third state, observational equivalence is preserved.

Lemma 3.4.1 If $\mathbf{R} \lhd [\![P]\!] \approx \mathbf{R} \lhd [\![P']\!]$, then $\mathbf{R} \lhd [\![P \| Q]\!] \approx \mathbf{R} \lhd [\![P' \| Q]\!]$ for any process Q.

PROOF *Very similar to CCS, [Mil89] pp. 98*
We shall demonstrate that $\star = \{(\mathbf{R} \lhd [\![P_1 \| Q]\!], \mathbf{R} \lhd [\![P_2 \| Q]\!]) \mid \mathbf{R} \lhd [\![P_1]\!] \approx \mathbf{R} \lhd [\![P_2]\!]\}$ is a bisimulation.

To prove this we can show that, for $\sigma \in Act^*$ and supposing $\mathbf{R} \lhd [\![P_1 \| Q]\!] \star \mathbf{R} \lhd [\![P_2 \| Q]\!]$:

1. (a) $\mathbf{R} \lhd [\![P_1 \| Q]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![P_1' \| Q]\!]$ implies $\mathbf{R} \lhd [\![P_2 \| Q]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![P_2' \| Q]\!]$ and $\mathbf{R}' \lhd [\![P_1' \| Q]\!] \star \mathbf{R}' \lhd [\![P_2' \| Q]\!]$.
   (b) $\mathbf{R} \lhd [\![P_1 \| Q]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![P_1 \| Q']\!]$ implies $\mathbf{R} \lhd [\![P_2 \| Q]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![P_2 \| Q']\!]$ and $\mathbf{R}' \lhd [\![P_1 \| Q']\!] \star \mathbf{R}' \lhd [\![P_2 \| Q']\!]$.
2. The same, swapping P_2 and P_1. This shall follow immediately from the symmetry of the parallelisation operator.

1. (a) $\mathbf{R} \lhd [\![P_1 \| Q]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![P_1' \| Q]\!]$, so must be able to derive the transition $\mathbf{R} \lhd [\![P_1]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![P_1']\!]$. As $\mathbf{R} \lhd [\![P_1]\!] \approx \mathbf{R} \lhd [\![P_2]\!]$, we can also derive $\mathbf{R} \lhd [\![P_2]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![P_2']\!]$ with $\mathbf{R}' \lhd [\![P_1']\!] \approx \mathbf{R}' \lhd [\![P_2']\!]$. We may then derive $\mathbf{R} \lhd [\![P_2 \| Q]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![P_2' \| Q]\!]$, and $\mathbf{R}' \lhd [\![P_1' \| Q]\!] \star \mathbf{R}' \lhd [\![P_2' \| Q]\!]$.
   (b) $\mathbf{R} \lhd [\![P_1 \| Q]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![P_1 \| Q']\!]$, so $\mathbf{R} \lhd [\![Q]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![Q']\!]$. Hence, $\mathbf{R} \lhd [\![P_2 \| Q]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![P_2 \| Q']\!]$, and $\mathbf{R}' \lhd [\![P_1 \| Q']\!] \star \mathbf{R}' \lhd [\![P_2 \| Q']\!]$.

□

Another useful property of observational equivalence is that it is an equivalence relation: it is *reflexive*, *symmetric* and *transitive*.

Theorem 3.4.2 Observational equivalence is an equivalence relation.

PROOF We split the proof into proving the three sub-properties. That is, for all $\mathbf{R}_1 \lhd [\![P_1]\!]$, $\mathbf{R}_2 \lhd [\![P_2]\!]$ and $\mathbf{R}_3 \lhd [\![P_3]\!]$,

1. Reflexivity: $\mathbf{R}_1 \lhd [\![P_1]\!] \approx \mathbf{R}_1 \lhd [\![P_1]\!]$.
2. Symmetry: $\mathbf{R}_1 \lhd [\![P_1]\!] \approx \mathbf{R}_2 \lhd [\![P_2]\!]$ implies $\mathbf{R}_2 \lhd [\![P_2]\!] \approx \mathbf{R}_1 \lhd [\![P_1]\!]$.

3. Transitivity: $\mathbf{R}_1 \lhd [\![P_1]\!] \approx \mathbf{R}_2 \lhd [\![P_2]\!]$ and $\mathbf{R}_2 \lhd [\![P_2]\!] \approx \mathbf{R}_3 \lhd [\![P_3]\!]$ implies $\mathbf{R}_1 \lhd [\![P_1]\!] \approx \mathbf{R}_3 \lhd [\![P_3]\!]$.

The proofs of (1) and (2) are immediate from the definitions of $\approx$ and bisimulation. (3) follows easily, too. □

3.5 Kernel of πDemos states

Let Π be the set of states in the operational semantics of πDemos, ranged over by tuples $D = \langle \mathcal{EL}, \mathcal{RL}, \Sigma \rangle$. The equivalence class of a state D with respect to g,

$$\|D\| \stackrel{\text{def}}{=} \{E \in \Pi \mid g(D) \approx g(E)\},$$

is the set of states (containing programs) that are "equal", in some sense relating to the execution of the states, to the state under consideration. In our case this shall be observational equivalence in the transition system.

Suppose we want to find a member of the equivalence class for a particular state, D. We cannot simply construct $\|D\|$ by checking every Demos state for equivalence: the set of states Π is infinite. Instead, we must obtain such a program (which we will say belongs to the *kernel* of Π, Π_K) from its syntax.

Definition The πDemos state $D_k \in \Pi_K$ shall be the πDemos state $D = (\mathcal{EL}, \mathcal{RL})$ with all the $\texttt{hold}(t)$ commands removed and all times (both in the event list and the resource lists) set to zero. (If priorities existed in πDemos — and they do not — we would equalise these too.) Where the first command of a process in the event list is $\texttt{getR}(r, 1)$ and r is unavailable in $\mathcal{RL}$, the remaining body of code of the process is placed (with the other kernelised process information) in the waiting list of r. The resource list of the kernel state has no extraneous resource entries; that is, there are no entries $(r, \texttt{ff}, [])$ where r is not referred to by any process. We shall assume that the operational semantics are extended to maintain this property.

The following lemma asserts that the the corresponding kernel state for a non-parallel process belongs to the equivalence class. We use the notation $\mathcal{RL}[(r, b, W)]$ to mean the resource list $\mathcal{RL}$ with the entry for r being (r, b, W). We use $\mathcal{EL}[P/Q]$ to mean the event list $\mathcal{EL}$ having P in place of Q (thus possibly causing the order to be different if the times of P and Q are different). The reader should be cautious as we use the subscript $_k$ to indicate kernelisation following other subscripts.

Lemma 3.5.1 For all Π states $(\mathcal{EL}, \mathcal{RL})$, if $g(\mathcal{EL}, \mathcal{RL}) = \mathbf{R} \lhd [\![P]\!]$ for some $\mathbf{R} \lhd [\![P]\!]$ where P is non-parallel, then $g(\mathcal{EL}, \mathcal{RL}) = \mathbf{R} \lhd [\![P]\!] \approx \mathbf{R}' \lhd [\![P_k]\!] = g((\mathcal{EL}, \mathcal{RL})_k)$

Proof By the definition of g, we have $g(\mathcal{EL}, \mathcal{RL}) = \mathbf{R} \lhd g_E(\mathcal{EL}) \mid W$ where $g_R(\mathcal{RL}) = \langle \mathbf{R}, W \rangle$. Since the result is not the parallel composition of processes, there is only one body of code in $(\mathcal{RL}, \mathcal{EL})$. Irrespective of where it came from, we may induce over the body of code (*i.e.,* $P \in \Pi$) found in some process descriptor.

No code in body:

Notice that this would imply that the process came from the event list; otherwise, if it had been a process waiting on a resource, we would have `getR` at the head of the body.

Kernelisation yields exactly the same process (albeit possibly with a different event time). Consequently, $g(D) = g(D_k)$. As $\approx$ is reflexive, $g(D) = g(D_k)$

$P \equiv \texttt{putR}(r, 1)$ **or** `close`:

Again, [2] $D = D_k$ implies that $g(D) = g(D_k)$, so $g(D) \approx g(D_k)$.

$P \equiv \texttt{getR}(r, 1)$:

There are two cases to consider: either $(r, \mathtt{tt}, [])$ is in $\mathcal{RL}$ or $(r, \mathtt{ff}, W)$ is in $\mathcal{RL}$.

If $(r, \mathtt{tt}, [])$ is in $\mathcal{RL}$, we have $D = D_k$ which implies $g(D) = g(D_k)$, so $g(D) \approx g(D_k)$.

Otherwise, r is not in $\mathbf{R}$, the available resource collection of $g(D)$ or in $\mathbf{R}'$, the available resource collection of $g(D_k)$. Consequently, irrespective of whether the process descriptor was in $\mathcal{EL}$ or waiting on r in $\mathcal{RL}$, we can derive no transition from $g(D)$, nor can we from $g(D_k)$. Thus $g(D) \approx g(D_k)$.

$P \equiv \texttt{hold}(t)$:

Then $g(D) = \mathbf{R} \lhd [\![\texttt{hold}(t)]\!]$; the corresponding kernel state, D_k, gives $g(D_k) = \mathbf{R} \lhd [\![\varepsilon]\!]$. Neither of these processes may make any observable transition — $[\![\texttt{hold}(t)]\!]$ may only make a silent transition.

$P \equiv P_1; P_2$:

Without loss of generality, we may assume that $P \equiv a; body$, where a is a single command. Consider the cases for a:

$a \equiv \texttt{getR}(r, 1)$:

Then $P \equiv \texttt{getR}(r, 1); body$, and $g(D) = \mathbf{R} \lhd [\![\texttt{getR}(r, 1); body]\!]$. There are two cases to consider: either $r \in\!\!\!\!\cdot\ \mathbf{R}$ iff $\langle r, \mathtt{tt}, [] \rangle \in \mathcal{RL}$ or $r \notin \mathbf{R}$ iff $\langle r, \mathtt{ff}, W \rangle \in \mathcal{RL}$.

- ($r \notin \mathbf{R}$): We can derive no transition from $\mathbf{R} \lhd [\![\texttt{getR}(r, 1); body]\!]$. In the kernel, $g(D_k) = \mathbf{R} \lhd [\![\texttt{getR}(r, 1); body_k]\!]$, irrespective of whether $body$ was in the waiting list of r or $\texttt{getR}(r, 1); body$ was in the event list, which we can derive no transition from.

- ($r \in\!\!\!\!\cdot\ \mathbf{R}$): Then $\mathbf{R} \equiv \mathbf{R}', r$, and we may only derive the transition $\mathbf{R}', r \lhd [\![\texttt{getR}(r, 1); body]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![body]\!]$. Kernelisation yields D_k where the process is in the event list (because r is available), so $g(D_k) = \mathbf{R}', r \lhd [\![\texttt{getR}(r, 1); body_k]\!]$, from which we may only derive the transition $\mathbf{R}', r \lhd [\![\texttt{getR}(r, 1); body_k]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![body_k]\!]$.

  Now, let D' be the πDemos state $D[body/\texttt{getR}(r, 1); body][\langle r, \mathtt{ff}, [] \rangle / \langle r, \mathtt{tt}, [] \rangle]$. Then $g(D') = \mathbf{R}' \lhd [\![body]\!]$, and, by induction, $g(D'_k) = \mathbf{R}' \lhd [\![body_k]\!] \approx \mathbf{R}' \lhd [\![body]\!]$. Consequently, by the definition of $\approx$, $g(D) = \mathbf{R}', r \lhd [\![\texttt{getR}(r, 1); body]\!] \approx \mathbf{R}', r \lhd [\![\texttt{getR}(r, 1); body_k]\!] = g(D_k)$.

[2] We shall write '=' in the sense of two states in Π being exactly the same other than having possibly different event times. Their translations into the transition system will thus be the same.

$\underline{a \equiv \texttt{putR}(r,1)}$:

Similar to ($r \Subset \mathbf{R}$) case.

$\underline{a \equiv \texttt{close}}$:

Similar to ($r \notin \mathbf{R}$) case.

$\underline{a \equiv \texttt{hold}(t)}$:

Then $P \equiv \texttt{hold}(t); body$ and $g(D) = \mathbf{R} \triangleleft [\![\texttt{hold}(t); body]\!]$. Kernelisation yields $g(D_k) = \mathbf{R} \triangleleft [\![body_k]\!]$. By induction, $g(D[body/\texttt{hold}(t); body]) = \mathbf{R} \triangleleft [\![body]\!] \approx \mathbf{R} \triangleleft [\![body_k]\!] = g(D_k)$. Notice that the hold generates a non-observable τ-transition, so $g(D) \approx g(D_k)$ by the definition of $\approx$.

□

We now generalise, showing that the translation of any πDemos state into the kernel generates an element of the equivalence class.

Theorem 3.5.2 Translation of a πDemos program into the kernel generates a member of the equivalence class. That is, $D_k \in \|D\|$; or $g(D) \approx g(D_k)$.

PROOF *By induction on* $n = |\mathcal{EL}| + |\mathcal{WL}|$ [3] where $|\mathcal{WL}|$ = the number of processes waiting on resources.

We shall say that two $\pi Demos$ states are equal even if they have differing event times or the order of lists differs if that order is arbitrary as their intepretation in the transition system will be the same. For convenience, we shall have two base cases.

$\underline{n = 0}$:

There are no waiting or active processes. Consequently, $D \equiv D_k$ so $g(D) = g(D_k)$. $\approx$ is an equivalence relation and therefore reflexive, so $g(D) \approx g(D_k)$.

$\underline{n = 1}$:

Immediate by Lemma 3.5.1.

$\underline{n > 1}$:

The event and resource list translated into σDemos states will be of the form $g(D) = \mathbf{R} \triangleleft [\![P_1 \| P_2]\!]$.

Let $D_1 = D[P_1/(P_1 \| P_2)]$ and $D_2 = D[P_2/(P_1 \| P_2)]$ — essentially, deleting from the event and waiting lists a non-zero number of processes. Then $g(D_1) = \mathbf{R} \triangleleft [\![P_1]\!]$ and $g(D_2) = \mathbf{R} \triangleleft [\![P_2]\!]$. By induction, $\mathbf{R} \triangleleft [\![P_1]\!] \approx \mathbf{R} \triangleleft [\![P_{1k}]\!] = g(D_{1k})$ and $\mathbf{R} \triangleleft [\![P_2]\!] \approx \mathbf{R} \triangleleft [\![P_{2k}]\!] = g(D_{2k})$.

Now, by Lemma 3.4.1, $\mathbf{R} \triangleleft [\![P_1 \| P_2]\!] \approx \mathbf{R} \triangleleft [\![P_{1k} \| P_2]\!] \approx \mathbf{R} \triangleleft [\![P_{1k} \| P_{2k}]\!] = g(D_k)$. Therefore, because $\approx$ is an equivalence relation, $g(D) \approx g(D_k)$

□

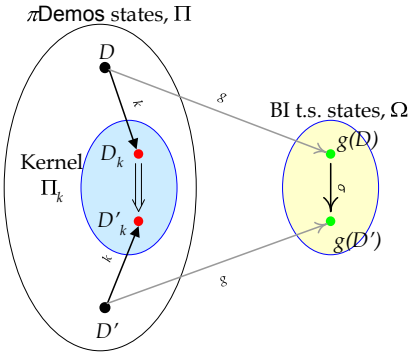


Figure 3.1: The correspodence between states of the transition system and the of operational semantics.

3.6 Correspondence between Π_k and σDemos states

As we used Σ in the previous section to mean the signature of a state in πDemos, to avoid confusion we shall use the upper-case Greek letter Ω to denote the set of σDemos (transition system) states. So,

$$g : \Pi \to \Omega$$

and

$$\bar{f} : \Omega \to \Pi.$$

The range of g *when applied to states in the kernel*, Π_k, shall be written Ω_k. That is,

$$\Omega_k \stackrel{\text{def}}{=} \{\mathbf{R} \triangleleft [\![P]\!] \mid \exists D_k \in \Pi_k.\ g(D_k) = \mathbf{R} \triangleleft [\![P]\!]\}$$

It is interesting to consider how our translations between Π and σDemos states work when we restrict our attention to the kernel sets Π_k and Ω_k. Looking at the definitions of g and $\bar{f}$, they do not seem to 'lose' any information about the state. For some πDemos state in the kernel, $D_k \in \Pi_k$, given its representation in σDemos, $g(D_k) \in \Omega_k$, we would expect to be able to determine D_k. Similarly, given $\bar{f}(\mathbf{R} \triangleleft [\![P]\!])$, we would expect to be able to determine $\mathbf{R} \triangleleft [\![P]\!]$. We ask, then, the question: does $\bar{f}$ 'undo' g, and g 'undo' $\bar{f}$?

Theorem 3.6.1 For any state $D_k \in \Pi_k$,

$$\bar{f}(g(D_k)) = D_k$$

PROOF OUTLINE *By induction on* $|\mathcal{EL}|+|\mathcal{WL}|$ where $\mathcal{WL}$ is the list of processes waiting on resources.

[3] $|\cdot|$ may be read as "list length". We shall refrain from giving the normal recursive definition.

First consider the resource list, and note that the collection of resource labels is preserved, as is the availability of each resource. Also notice that the each non-parallel (sub)process label is preserved.

The interesting case is where $|\mathcal{EL}| + |\mathcal{WL}| = 1$ and the only process is in the waiting list of some unavailable resource r, $\mathcal{WL}_r$. Applying g gives a process with $\texttt{getR}(r, 1)$ at its head, and application of $\bar{f}$ removes this, putting the process back into the waiting list of r.

Notice that we do *not* consider the case where $\texttt{getR}(r, 1)$ is the command at the head of the event list and r is unavailable, as this is not a kernel state. Therefore, we do not have a problem deciding from $g(D_k)$ whether a process has or has not yet 'executed' the $\texttt{getR}(r, 1)$ command. □

Theorem 3.6.2 For any state $\mathbf{R} \triangleleft [\![P]\!] \in \Omega$ — note that we do *not* require that $\mathbf{R} \triangleleft [\![P]\!]$ be in Ω_k —

$$g(\bar{f}(\mathbf{R} \triangleleft [\![P_{[\ell]}]\!])) = \mathbf{R} \triangleleft [\![P_{[\ell]}]\!]$$

PROOF OUTLINE *By induction on the structure of* $\mathbf{R} \triangleleft [\![P]\!]$. We first note that the action of g after $\bar{f}$ on an arbitrary state $\mathbf{R} \triangleleft [\![P]\!]$ results in the same resource collection, $\mathbf{R}$. We also note that the labels of non-parallel processes are preserved.

Again, there is essentially one interesting case: If $P \equiv \texttt{getR}(r, 1); body$ and $r \notin \mathbf{R}$, *body* will be placed in the waiting list of r in $\bar{f}(\mathbf{R} \triangleleft [\![P]\!])$. g will restore the $\texttt{getR}(r, 1)$ command. Otherwise, for a non-parallel process, the code will be placed in the event list of $\bar{f}(\mathbf{R} \triangleleft [\![P]\!])$. g will not add anything to the front, thus restoring $\mathbf{R} \triangleleft [\![P]\!]$. □

A *bijection* between two sets A and B is a function that maps every element of A to exactly one element of B, with every element of B being mapped onto exactly once.

Corollary 3.6.3 $\bar{f}$ is the *inverse function* of g (or, equivalently, g is the inverse of $\bar{f}$) when we consider the sets of kernel states. That is, for $D_k \in \Pi_k$ and $(\mathbf{R} \triangleleft [\![P]\!])_k \in \Omega_k$,

$$g(\bar{f}((\mathbf{R} \triangleleft [\![P]\!]))_k) = (\mathbf{R} \triangleleft [\![P]\!])_k$$

and

$$\bar{f}(g(D_k)) = D_k.$$

Consequently, g is a *bijection* from Ω_k to Π_k and $\bar{f}$ is a bijection from Π_k to Ω_k.

PROOF The proof is immediate from the above theorems and the definitions of inverse functions and bijections. □

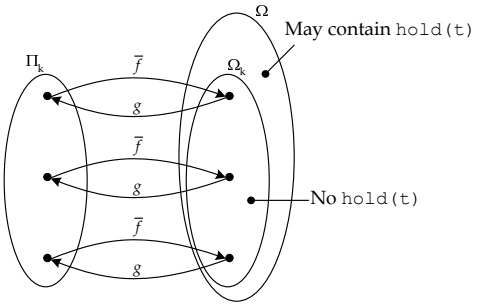


Figure 3.2: Bijection between kernel and transition system

3.7 Correctness of translations

3.7.1 Correctness of g

As mentioned before, in order to demonstrate that Theorem 3.2.1, that the transition system simulates πDemos, is not vacuously true through choosing an artificial definition of g, we must show the following property, which relates to the soundness of the transition system with respect to the operational semantics of πDemos. This is illustrated in Figure 3.1. We make use of the soundness property of the transition system (Theorem 3.3.1) and Theorem 3.6.1.

Theorem 3.7.1 Let $\Longrightarrow^*$ represent the reflexive, transitive closure of the transitions derivable according to the operational semantics. Then $\forall D, D' \in \Pi, \sigma \in Act^*$

$$g(D) \stackrel{\sigma}{\Longrightarrow} g(D') \text{ implies } D_k \Longrightarrow^* D'_k.$$

Proof By Theorem 3.3.1,

$$\mathbf{R} \lhd [\![P]\!] \stackrel{\sigma}{\longrightarrow} \mathbf{R}' \lhd [\![P']\!] \text{ implies } \bar{f}(\mathbf{R} \lhd [\![P]\!]) \Longrightarrow^* \bar{f}(\mathbf{R}' \lhd [\![P']\!]).$$

The property applies for every sub-transition to derive $\mathbf{R} \lhd [\![P]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![P']\!]$, so

$$\mathbf{R} \lhd [\![P]\!] \stackrel{\sigma}{\Longrightarrow} \mathbf{R}' \lhd [\![P']\!].$$

Now, $\mathbf{R} \lhd [\![P]\!]$ ranges over $g(D_k)$, so

$$g(D_k) \stackrel{\sigma}{\Longrightarrow} g(D'_k) \text{ implies } \bar{f}(g(D_k)) \Longrightarrow^* \bar{f}(g(D'_k)).$$

By Theorem 3.6.1,

$$g(D_k) \stackrel{\sigma}{\Longrightarrow} g(D'_k) \text{ implies } D_k \Longrightarrow^* D'_k.$$

Finally, because $g(D) \approx g(D_k)$,

$$g(D) \stackrel{\sigma}{\Longrightarrow} g(D') \text{ implies } D_k \Longrightarrow^* D'_k,$$

as required. □

3.7.2 Correctness of $\bar{f}$

Simularly, to demonstrate that we have not chosen an artificial $\bar{f}$ to make Theorem 3.3.1, the soundness of the transition system, vacuously true, we require the following theorem, which relies on the simulation theorem (Theorem 3.2.1) and on Theorem 3.6.2.

Theorem 3.7.2 Let $\xrightarrow{\sigma}{}^{*}$ represent the reflexive, transitive closure of the transition relation $\xrightarrow{\sigma}$. Then for all $\mathbf{R} \triangleleft [\![P]\!] \in \Omega$,

$$\bar{f}(\mathbf{R} \triangleleft [\![P]\!]) \Longrightarrow \bar{f}(\mathbf{R}' \triangleleft [\![P']\!]) \textit{ implies } \mathbf{R} \triangleleft [\![P]\!] \xrightarrow{\sigma}{}^{*} \mathbf{R}' \triangleleft [\![P']\!]$$

PROOF By Theorem 3.2.1, for all $D, D' \in \Pi$,

$$D \Longrightarrow D' \text{ implies } g(D) \xrightarrow{\sigma}{}^{*} g(D').$$

Now, $\bar{f}(\mathbf{R} \triangleleft [\![P]\!]), \bar{f}(\mathbf{R}' \triangleleft [\![P']\!]) \in \Pi$, so

$$\bar{f}(\mathbf{R} \triangleleft [\![P]\!]) \Longrightarrow \bar{f}(\mathbf{R}' \triangleleft [\![P']\!]) \text{ implies } g(\bar{f}(\mathbf{R} \triangleleft [\![P]\!])) \xrightarrow{\sigma}{}^{*} g(\bar{f}(\mathbf{R}' \triangleleft [\![P']\!])).$$

Theorem 3.6.2 gives us

$$\bar{f}(\mathbf{R} \triangleleft [\![P]\!]) \Longrightarrow \bar{f}(\mathbf{R}' \triangleleft [\![P']\!]) \text{ implies } \mathbf{R} \triangleleft [\![P]\!] \xrightarrow{\sigma}{}^{*} \mathbf{R}' \triangleleft [\![P']\!],$$

as required. □

Chapter 4

Demos and PBI

We have now defined, in abstract terms, the logic that we shall use, **PBI**, and a transition system for Demos programs. It remains to consider how we integrate the two and what we are able to express in the resulting system.

4.1 PBI and σDemos

Recall that the definition of the semantics of **PBI** given in §1.4 was by a monoid,

$$\mathcal{M} = \langle Res \times Prog, e, \cdot, \sqsubseteq \rangle.$$

The monoid that we shall use to define the logic for Demos models shall be denoted M_D,

$$\mathcal{M}_D \stackrel{\text{def}}{=} \langle \mathcal{R} \times \mathcal{P}_{\text{ROC}},\ \wr\!\wr \triangleleft [\![\varepsilon]\!] , \circ, =_\alpha \rangle.$$

The composition relation $\circ$ combines states in $\Omega = \mathcal{R} \times \mathcal{P}_{\text{ROC}}$. It simply forms the multiset union of the resource component and the parallel composition of the process component.

$$(\mathbf{R} \triangleleft [\![P]\!]) \circ (\mathbf{S} \triangleleft [\![Q]\!]) \stackrel{\text{def}}{=} \mathbf{R} \uplus S \triangleleft [\![P \| Q]\!]$$

The comparison pre-order $=_\alpha$ is a simple equivalence,

$$\mathbf{R} \triangleleft [\![P]\!] =_\alpha \mathbf{S} \triangleleft [\![Q]\!] \iff^{\text{def}} \mathbf{R} \equiv \mathbf{S} \wedge P \equiv Q.$$

$=_\alpha$ considers pairs to be equal iff they have the same resource multisets and their process terms are the same, up to the commutativity and associativity of parallelisation, with the inclusion of an arbitrary number of $[\![\varepsilon]\!]$ processes. Thus $\wr\!\wr \triangleleft [\![\varepsilon]\!]$ is the identity element under $\circ$.

The semantics of **PBI** following from these definitions is presented in Table 4.1.

Finally, we must define the relations $\restriction$ and $\downharpoonright$ for the multiplicative modalities. These define, respectively, the resources acquired and released by particular

$\mathbf{R} \lhd [\![P]\!] \models r$	iff	$\mathbf{R} \lhd [\![P]\!] =_\alpha \wr r \int \lhd [\![\varepsilon]\!]$
$\mathbf{R} \lhd [\![P]\!] \models Q$	iff	$\mathbf{R} \lhd [\![P]\!] =_\alpha \wr \int \lhd [\![Q]\!]$
$\mathbf{R} \lhd [\![P]\!] \models \varphi \wedge \psi$	iff	$\mathbf{R} \lhd [\![P]\!] \models \varphi$ and $\mathbf{R} \lhd [\![P]\!] \models \psi$
$\mathbf{R} \lhd [\![P]\!] \models \varphi \vee \psi$	iff	$\mathbf{R} \lhd [\![P]\!] \models \varphi$ or $\mathbf{R} \lhd [\![P]\!] \models \psi$
$\mathbf{R} \lhd [\![P]\!] \models \varphi \to \psi$	iff	$\mathbf{R} \lhd [\![P]\!] \models \varphi$ implies $\mathbf{R} \lhd [\![P]\!] \models \psi$
$\mathbf{R} \lhd [\![P]\!] \models \varphi * \psi$	iff	for some $\mathbf{S} \lhd [\![Q]\!], \mathbf{S}' \lhd [\![Q']\!] \in \Omega$ [$\mathbf{R} \lhd [\![P]\!] = \mathbf{S} \uplus \mathbf{S}' \lhd [\![Q \| Q']\!]$) and $\mathbf{S} \lhd [\![Q]\!] \models \varphi$ and $\mathbf{S}' \lhd [\![Q']\!] \models \psi$]
$\mathbf{R} \lhd [\![P]\!] \models \varphi \mathrel{-\!\!*} \psi$	iff	for all $\mathbf{S} \lhd [\![Q]\!] \in \Omega$ $\mathbf{S} \lhd [\![Q]\!] \models \varphi$ implies $\mathbf{R} \uplus \mathbf{S} \lhd [\![P \| Q]\!] \models \psi$
$\mathbf{R} \lhd [\![P]\!] \models \langle A \rangle \varphi$	iff	for some $\alpha \in A$ $\mathbf{R} \lhd [\![P]\!] \xrightarrow{\alpha} \mathbf{R} \lhd [\![P']\!]$ and $\mathbf{R} \lhd [\![P']\!] \models \varphi$
$\mathbf{R} \lhd [\![P]\!] \models [A] \varphi$	iff	for all $\alpha \in A$ $\mathbf{R} \lhd [\![P]\!] \xrightarrow{\alpha} \mathbf{R} \lhd [\![P']\!]$ implies $\mathbf{R} \lhd [\![P']\!] \models \varphi$
$\mathbf{R} \lhd [\![P]\!] \models \langle A \rangle^+_{\mathbf{new}} \varphi$	iff	$\exists a \in A.$ $a \restriction \mathbf{S} \lhd [\![Q]\!]$ and $\mathbf{R} \lhd [\![P]\!] \xrightarrow{a} \mathbf{R} \uplus \mathbf{S} \lhd [\![P' \| Q]\!]$ and $\mathbf{R} \uplus \mathbf{S} \lhd [\![P' \| Q]\!] \models \varphi$
$\mathbf{R} \lhd [\![P]\!] \models [A]^+_{\mathbf{new}} \varphi$	iff	$\forall a \in A.$ $a \restriction \mathbf{S} \lhd [\![Q]\!]$ and $\mathbf{R} \lhd [\![P]\!] \xrightarrow{a} \mathbf{R} \uplus \mathbf{S} \lhd [\![P' \| Q]\!]$ implies $\mathbf{R} \uplus \mathbf{S} \lhd [\![P' \| Q]\!] \models \varphi$
$\mathbf{R} \lhd [\![P]\!] \models \langle A \rangle^-_{\mathbf{new}} \varphi$	iff	$\exists a \in A.\ a \restriction \mathbf{S} \lhd [\![Q_1]\!]$ and $\mathbf{R} \lhd [\![P]\!] = \mathbf{S} \uplus \mathbf{T} \lhd [\![Q_1 \| Q_2]\!]$ and $\mathbf{S} \uplus \mathbf{T} \lhd [\![Q_1 \| Q_2]\!] \xrightarrow{a} \mathbf{T} \lhd [\![Q_2']\!]$ and $\mathbf{T} \lhd [\![Q_2']\!] \models \varphi$
$\mathbf{R} \lhd [\![P]\!] \models [A]^-_{\mathbf{new}} \varphi$	iff	$\forall a \in A.\ a \restriction \mathbf{S} \lhd [\![Q_1]\!]$ and $\mathbf{R} \lhd [\![P]\!] = \mathbf{S} \uplus \mathbf{T} \lhd [\![Q_1 \| Q_2]\!]$ and $\mathbf{S} \uplus \mathbf{T} \lhd [\![Q_1 \| Q_2]\!] \xrightarrow{a} \mathbf{T} \lhd [\![Q_2']\!]$ implies $\mathbf{T} \lhd [\![Q_2']\!] \models \varphi$
$\mathbf{R} \lhd [\![P]\!] \models \neg \varphi$	iff	$\mathbf{R} \lhd [\![P]\!] \not\models \varphi$
$\mathbf{R} \lhd [\![P]\!] \models \top$	always	
$\mathbf{R} \lhd [\![P]\!] \not\models \bot$	for any	$\mathbf{R} \lhd [\![P]\!]$
$\mathbf{R} \lhd [\![P]\!] \models I_R$	iff	$\exists P \in \mathcal{P}_{\mathrm{ROC}}.\ \ \mathbf{R} \lhd [\![P]\!] =_\alpha \wr\!\int \lhd [\![P]\!]$
$\mathbf{R} \lhd [\![P]\!] \models I_P$	iff	$\exists \mathbf{S} \in \mathcal{R}.\ \ \mathbf{R} \lhd [\![P]\!] =_\alpha \mathbf{S} \lhd [\![\varepsilon]\!]$
$\mathbf{R} \lhd [\![P]\!] \models I$	iff	$\mathbf{R} \lhd [\![P]\!] =_\alpha \wr\!\int \lhd [\![\varepsilon]\!]$

Table 4.1: Demos semantics of **PBI**

actions.

$$\begin{array}{rcl} \mathrm{getR}_r^n & \upharpoonright & \wr r^n \int \triangleleft [\![\varepsilon]\!] \\ \mathrm{getB}_b^n & \upharpoonright & \wr b^n \int \triangleleft [\![\varepsilon]\!] \\ \mathrm{getS}_x^n & \upharpoonright & \wr x^n \int \triangleleft [\![\varepsilon]\!] \end{array}$$

$$\begin{array}{rcl} \mathrm{putR}_r^n & \uparrow & \wr r^n \int \triangleleft [\![\varepsilon]\!] \\ \mathrm{putB}_b^n & \uparrow & \wr b^n \int \triangleleft [\![\varepsilon]\!] \\ \mathrm{putS}_x^n & \uparrow & \wr x^n \int \triangleleft [\![\varepsilon]\!] \end{array}$$

4.1.1 Intuitive account

- Resource propositions r hold in states with no process component where r is the only resource available.
- Process propositions P hold in states with null resource component with P being the only process.
- The additive unit of $\wedge$ is $\top$, which holds in any state.
- The additive unit of $\vee$ is $\bot$, which holds in no state.
- The multiplicative resource unit I_R holds in states with no resource component, but allows an arbitrary (possibly parallel) process.
- The multiplicative resource unit I_P holds in states with only a null process component, but allows an arbitrary resource component.
- The (extraneous) multiplicative unit I holds in states with only a null process component and no available resources.
- The additive logical operators $\wedge$,$\vee$ and $\rightarrow$ are all as in classical (boolean) logic. $\wedge$ allows us to assert that two propositions hold in a state, $\vee$ allows us to assert that at least one of two propositions holds in a state, and $\rightarrow$ allows us to assert that if one proposition holds in a state, so does another.
- The multiplicative conjunction of two propositions will hold in a state, $\mathbf{R} \triangleleft [\![P]\!] \models \varphi * \psi$, if the state can be partitioned into two parts, the partition being of resources and processes, with one part making φ hold and the other making ψ hold.
- Multiplicative implication relates to the addition of resources or processes: $\mathbf{R} \triangleleft [\![P]\!] \models \varphi \mathrel{-\!\!*} \psi$ holds if all states in which φ holds when composed with the state $\mathbf{R} \triangleleft [\![P]\!]$ give states in which ψ holds.
- $\langle A \rangle \varphi$ will hold if a state can evolve to a new state by a single action $\alpha \in A$ in which φ holds given no change in the resource component of the state. Similarly, $[A]\varphi$ holds in a state if all transitions labelled $\alpha \in A$ lead to states in which φ holds, again with no change in the resource component.

- $\mathbf{R} \triangleleft [\![P]\!] \models \langle A \rangle^+ \varphi$ will hold if by some action $\sigma \in A$, there is a transition to a new state with a *larger* resource collection, $\mathbf{R} \triangleleft [\![P]\!] \xrightarrow{\sigma} \mathbf{R} \uplus \mathbf{S} \triangleleft [\![P']\!]$, with $\sigma \upharpoonright \mathbf{S}$, and $\mathbf{R} \uplus \mathbf{S} \triangleleft [\![P']\!] \models \varphi$. Similarly, $[A]^+$ relates to all transitions with labels in A leading to states in which a proposition holds.
- Dually, the negative multiplicative modalities relate to transitions leading to states with *smaller* resource collections. They relate to transitions of the form $\mathbf{R} \uplus \mathbf{S} \triangleleft [\![P]\!] \xrightarrow{\sigma} \mathbf{R} \triangleleft [\![P']\!]$, with the difference in resource specified by $\sigma \upharpoonright \mathbf{S}$.
- Finally, because our pre-order is an equality, our logic is classical so we have negation, $\neg$. This allows us to assert that a proposition does not hold in a given state. The inclusion of negation makes $\rightarrow$ redundant: we could use the equivalence $\varphi \rightarrow \psi \Leftrightarrow \neg\varphi \vee \psi$.

If we want to reason about an arbitrary type of action occuring, we would form the disjunction of diamond modalities or the conjunction of the box modalities. We shall introduce the following abbreviations:

$$\mathbf{R} \triangleleft [\![P]\!] \models [A]_{\mathbf{any}} \varphi \quad \text{iff} \quad \mathbf{R} \triangleleft [\![P]\!] \models [A]^-_{\mathbf{new}} \varphi \wedge [A]^+_{\mathbf{new}} \varphi \wedge [A] \varphi$$

$$\mathbf{R} \triangleleft [\![P]\!] \models \langle A \rangle_{\mathbf{any}} \varphi \quad \text{iff} \quad \mathbf{R} \triangleleft [\![P]\!] \models \langle A \rangle^+_{\mathbf{new}} \varphi \vee \langle A \rangle^-_{\mathbf{new}} \varphi \vee \langle A \rangle \varphi.$$

Recalling the definition of I_R, the formula $r * I_R$ holds in states where r is the only available resource with an arbitrary process term. We shall soon see a formula representing r being *one* of the free resources. Recalling the definition of I_P, the formula $P * I_P$ holds in states where P is the only process in the state with an arbitrary collection of resources.

4.2 Deadlock in PBI

As presented in §1.3.2, deadlock freedom in Hennessy-Milner logic is defined as:

$$\mathfrak{D}_f^{hml} = \bigwedge_{i \in \mathbb{N} \cup \{\omega\}} [A]^i \langle A \rangle \top$$

We must make a few adaptations to this to make it suitable for us. Firstly, we must select an appropriate modality: we shall choose the $_{\mathbf{any}}$ modalities. Secondly, we wish to relax the requirement that the process never terminates to requiring that if it is in a non-terminated state, it can take an action. We do this by requiring I_P, the proposition indicating clean termination, or non-termination in every state. Consequently, the **PBI** formula for deadlock freedom in Demos is:

$$\mathfrak{D}_f^{D} = \bigwedge_{i \in \mathbb{N} \cup \{\omega\}} [-]^i_{\mathbf{any}} (\langle A \rangle_{\mathbf{any}} \top \vee I_P)$$

4.3 Examples

We shall start by considering the modalities. The additive modalities, as we have said, relate to transitions in which the resource component of state does not change. We have, then,

$$\mathbf{R} \lhd [\![\mathtt{hold}(5); \mathtt{putR}(r,1)]\!] \models \langle - \rangle \top,$$

which expresses that the process can act without changing the available resource collection. We also have

$$\wr r \int \lhd [\![\mathtt{hold}(5); \mathtt{putR}(r,1) \| \mathtt{getR}(r,1); \mathtt{putR}(r,1)]\!] \models [-](r * I_R).$$

which expresses that every action that does not change the available resource collection gives a state in which r is the only available resource. Notice that we do not have $\models \langle - \rangle \neg (r * I_R)$: the `getR` transition, the only transition leading to a state in which r is unavailable, does not match the semantics of the additive modality.

The additive modality does not refer to actions that will change the available resource collection, so

$$\mathbf{R} \lhd [\![\mathtt{putR}(r,1)]\!] \models [-] \bot,$$

which expresses that the process can make no action that does not change the available resource collection.

The multiplicative release modality is used for `put`X transitions. So, for example,

$$\wr r, s \int \lhd [\![\mathtt{putB}(b,1)]\!] \models \langle - \rangle^{+}_{\mathbf{new}} (b * r * s)$$

by the (only) transition

$$\wr s, r \int \lhd [\![\mathtt{putB}(b,1)]\!] \stackrel{\mathrm{putB}^1_b}{\longrightarrow} \wr b, r, s \int \lhd [\![\varepsilon]\!]$$

and that $\mathrm{putB}^1_b \upharpoonright \wr b \int \lhd [\![\varepsilon]\!]$.

The multiplicative acquisition modality is used for `get`X transitions:

$$\wr b, r, s \int \lhd [\![\mathtt{putB}(b,1) \| \mathtt{getB}(b,1)]\!] \models [-]^{-}_{\mathbf{new}} (r * s * I_R)$$

by the only `get` transition

$$\wr b, r, s \int \lhd [\![\mathtt{putB}(b,1) \| \mathtt{getB}(b,1)]\!] \stackrel{\mathrm{getB}^1_b}{\longrightarrow} \wr r, s \int \lhd [\![\mathtt{putB}(b,1) \| \varepsilon]\!]$$

In general, we will have the following:

$$\mathbf{R} \lhd [\![P]\!] \models \begin{array}{r} [-]^{+}_{\mathbf{new}} \varphi_+ \\ \wedge [-]^{-}_{\mathbf{new}} \varphi_- \\ \wedge [-] \varphi \end{array}$$

where φ_+ is some formula true in all states following a transition by any process with `put` at its head, φ_- following a transition by any process with `get` at its head, and φ following a transition by any other process.

$*$ relates to *partitions* of the state, both of processes and resources. Returning to the intelligence agency example, recalling that $\top$ holds for *any* state, we see that the following holds:

$$\wr C, F, 6, 5 \int \lhd [\![P]\!] \models F * \top$$

where C denotes the CIA line, 6 denotes the MI6 line *etc.* by $\wr F \int \lhd [\![\varepsilon]\!] \models F$ and $\wr C, 6, 5 \int \lhd [\![P]\!] \models \top$. So in general, just as we can in propositional **BI**, we may indicate that a resource r is available by the formula $r * \top$.

Returning to the jobshop example, we may wish to express that h and m are the only available resources. Clearly, $\wr h, m \int \lhd [\![P]\!] \models h * m * \top$, but also $\wr h, m, x \int \lhd [\![P]\!] \models h*m*\top$. Rather than using $\top$, we should have used I_R: $\wr h, m \int \lhd [\![P]\!] \models h*m*I_R$. This expresses that there are three partitions in the state, one having only the resource h, another having the resource m and the final one being an arbitrary process with no resource.

Suppose that we have a proposition ε_C which denotes termination of the CIA process. As has been illustrated, it is not the case that in every possible execution the CIA process terminates.[1]

$$\wr C, F, 6, 5 \int \lhd [\![P]\!] \not\models \bigwedge_{i \in \mathbb{N}_\omega} [-]^i_{\mathbf{any}} (\langle - \rangle_{\mathbf{any}} \top \vee \varepsilon_C)$$

This formula expresses that there is a state from which the system can make no transition and the CIA process has not terminated cleanly. We might like to consider the effect of removing MI6's line: would this allow the CIA to complete its task in all conditions? We may express this in the logic as:

$$\wr C, F, 6, 5 \int \lhd [\![P]\!] \mathrel{?\!\models} 6 * \bigwedge_{i \in \mathbb{N}_\omega} [-]^i_{\mathbf{any}} (\langle - \rangle_{\mathbf{any}} \top \vee \varepsilon_C)$$

Now, $\wr 6 \int \lhd [\![\varepsilon]\!] \models 6$. For the above formula to be true, then, all we have to show is that $\wr C, F, 5 \int \lhd [\![P]\!] \models \bigwedge_{i \in \mathbb{N}_\omega} [-]^i_{\mathbf{any}} (\langle - \rangle_{\mathbf{any}} \top \vee \varepsilon_C)$. (Notice that the choice of partition here is entirely deterministic.) Because MI6's line is removed, the MI6 process will not be able to proceed to take MI5's line or the FBI's. Consequently, the CIA process will terminate properly, so we do have[2]:

$$\wr C, F, 6, 5 \int \lhd [\![P]\!] \models 6 * \bigwedge_{i \in \mathbb{N}_\omega} [-]^i_{\mathbf{any}} (\langle - \rangle_{\mathbf{any}} \top \vee \varepsilon_C).$$

This illustrates how $*$ may be used to reason about *subtraction* of resources from a system.

It is a simple matter to ask if a partition exists that is non-deadlocking:

$$\mathbf{R} \lhd [\![P]\!] \mathrel{?\!\models} \mathfrak{D}^D_f * \mathfrak{D}^D_f$$

This will hold only if there are deadlock-free partitions $\mathbf{S}_1 \lhd [\![P_1]\!]$ and $\mathbf{S}_2 \lhd [\![P_2]\!]$ such that $\mathbf{R} \lhd [\![P]\!] = \mathbf{S_1} \uplus \mathbf{S_2} \lhd [\![P_1 \| P_2]\!]$. If we require a non-trivial partition (*i.e.,* without one partition being null-process), we would ask:

$$\mathbf{R} \lhd [\![P]\!] \mathrel{?\!\models} (\mathfrak{D}^D_f \wedge \neg I_P) * (\mathfrak{D}^D_f \wedge \neg I_P)$$

[1] The following formula comes from the definition of deadlock — see §4.2

[2] But not overall deadlock freedom

Now let us consider $-\!\!*$. Suppose that we want to consider the effect of installing a new line from the hub to the FBI. Would this result in deadlock freedom? The question would be posed in the logic as:

$$\wr C, F, 6, 5 \int \triangleleft [\![P]\!] \stackrel{?}{\models} F \mathrel{-\!\!*} \mathfrak{D}_f^D$$

The only state (model) satisfying $\models F$ is $\wr F \int \triangleleft [\![\varepsilon]\!]$. Thus to answer the above question, we must resolve satisfaction of $\wr C, F, F, 6, 5 \int \triangleleft [\![P]\!] \stackrel{?}{\models} \mathfrak{D}_f^D$. It turns out that this is true: either the CIA process or the MI6 process may end up blocked, but the other will eventually release the resource that it is waiting on.

The multiplicative modalities are also quite interesting: we could now use the positive (given the possible inclusion of processes, a better term might be *composition*) modalities $\langle A \rangle^+_{\mathbf{new}}$ and $[A]^+_{\mathbf{new}}$ to indicate a new process being generated. This occurs with an *Entity* command, and would require $\restriction$ to be extended by

$$\tau^{class}_{\text{Entity}} \restriction \wr \int \triangleleft [\![class]\!] .$$

Similarly, we could use the positive (a better term now might be *decomposition*) modalities to indicate processes being removed. This could be through the termination case of parallel composition or through an explicit *kill* command (which does not exist in Demos). With such rich multiplicative modalities, it may be desirable to exlude the case where the number of processes changes in this manner from the additive modality. Further consideration of this, though, is beyond our current scope. We shall, therefore, use the additive modalities for Entity declarations *etc.*, and leave $\restriction$ and $\downharpoonright$ defined only where necessary: on the resource actions.

It should be noted, though, that the total composition relation described here makes proof-search very non-deterministic. Take, for example, the formula $\langle - \rangle^+_{\mathbf{new}} \top \mathrel{-\!\!*} \varphi$. Not allowed if we define composition *partially*, $\mathbf{R} \triangleleft [\![P]\!] \circ \mathbf{S} \triangleleft [\![\varepsilon]\!] = \mathbf{R} \uplus \mathbf{S} \triangleleft [\![P]\!]$, to evaluate this we may have to consider adding every σDemos process that can release a resource!

4.3.1 Sub-deadlock

Consider again the jobshop example of §2.4. The worker process (the input stream of jobs to do) can, as it is, never deadlock. Consequently, if we consider the whole process we see that it is always possible to derive a transition using the worker process[3], so the whole system may never deadlock. Now, this is not of very much use to us: what we really want to know is whether or not the jobbers deadlock.

Deadlock of a process, whether it be a sub-process or not, is defined to occur where no transition is derivable at any reachable future state. For a sub-process $P_{k[\ell]}$, then, a state is deadlock-free with respect to $P_{k[\ell]}$ iff there is no reachable state from which $P_{k[\ell]}$ does not act in any transition from any future state. We can write this in **PBI** as:

$$\mathfrak{D}_f^{P,k} \quad \stackrel{\text{def}}{\iff} \quad \neg \bigvee \langle - \rangle_{\mathbf{any}} \left(\neg \varepsilon_k \wedge \bigwedge [-]^j_{\mathbf{any}} [K]_{\mathbf{any}} \bot \right)$$

[3]This is not the case if we consider a *fair* transition system, §6.1

Note that we have used a set of action labels indicating action by $P_{k[\ell]}$, K. This set can be defined by pattern-matching,

$$K = \{a \mid a \in \mathit{Labels} \text{ and } a = \ell : \_\_\}$$

Chapter 5

Synchronous parallel rule

5.1 Motivation

Consider a kitchen in which there is a knife, a pair of scissors, a mother and her child. The mother uses the knife and then the scissors before returning them. Adopting our Demos representation for processes, and letting k represent the knife and s the pair of scissors, her activity is modelled by the following sequence of actions:

$$P_m = \texttt{getR}(k,1);\texttt{getR}(s,1);\texttt{hold}(5);\texttt{putR}(s,1);\texttt{putR}(k,1).$$

The child only wants to use the scissors for a short period of time, and they are not allowed to use the knife. Their Demos representation is:

$$P_c = \texttt{getR}(s,1);\texttt{hold}(1);\texttt{putR}(s,1),$$

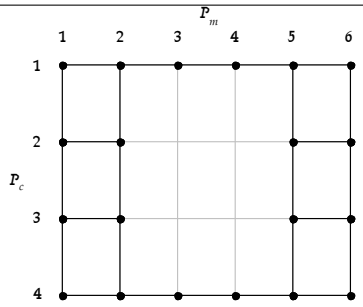
We want to ask if having the mother and the child in the kitchen at the same time will cause deadlock. Letting putative consequence (asking whether a formula is satisfied in a particular model) be written $?\!\models$, the question is represented formally by:

$$\wr s, k \int \triangleleft [\![P_m \| P_c]\!] \; ?\!\models \mathfrak{D}_f^D.$$

In the logics presented above, HML, CTL and modal-μ, the complexity of model-checking such questions increases as the state space generated by the transition system increases. Figure 5.1 shows the program fragment of the states that may be generated using the transition system with the parallel composition rule presented above.

Branching occurs where more than one transition is derivable from a particular state — there is *non-deterministic choice* of the next state to enter into. For example, at state $(1,1)$, either the child may take the scissors or the mother may take the knife. For the transition system presented above, the only time we may have such non-deterministic choice is where we have the parallel composition of processes where more than one is active (*i.e.,* not waiting on a resource).

The astute reader may have noticed that, when considering deadlock, it was not essential to distinguish whether the first action was or was not the mother



The axes are labelled by numbers representing the line of code being executed:

	P_m	P_c
1	$\texttt{getR}(k,1)$	$\texttt{getR}(s,1)$
2	$\texttt{getR}(s,1)$	$\texttt{hold}(1)$
3	$\texttt{hold}(5)$	$\texttt{putR}(s,1)$
4	$\texttt{putR}(s,1)$	ε
5	$\texttt{putR}(k,1)$	—
6	ε	—

So, for example, $(4,3) = [\![\texttt{putR}(s,1); \texttt{putR}(k,1) \| \texttt{putR}(s,1)]\!]$.
Arrowed lines indicate which states may be derived from each state.

Figure 5.1: Reachable state space for kitchen example derived by asynchronous transition system

picking up the knife, and then considering the real interaction. Why, then, did we need to generate a whole collection of states to distinguish this possibility? In terms of process interaction, it would have been more efficient to force the first action to be the mother taking the knife. Of course, the reader could argue that it is the rôle of the modeller to weed-out such irrelevances. Unfortunately, though, the modeller may not know in advance which processes we wish to model interaction between, so this cannot be expected.

5.2 Soundness: `try` and resource splitting

Let us consider how we may formulate a rule that allows synchronous execution of processes. If we include the simple, intuitive rule

$$\text{(SPAR)}: \quad \frac{\mathbf{R} \lhd [\![P]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P']\!] \quad \mathbf{S} \lhd [\![Q]\!] \xrightarrow{\sigma} \mathbf{S}' [\![Q']\!]}{\mathbf{R} \uplus \mathbf{S} \lhd [\![P \| Q]\!] \xrightarrow{\sigma} \mathbf{R}' \uplus \mathbf{S}' \lhd [\![P' \| Q]\!]},$$

the semantics of the transition system become unsound with respect to the operational semantics of Demos. The unsoundness comes, in part, from the fact that we have considered only a simple split of resources: in deducing a transition, we may allocate one processes resources it does not need, thereby depriving the other process of resources. This error may manifest itself in a `try` command choosing the wrong branch.

To give a concrete example of this, consider the jobshop example (§2.4) with one extra hammer. Suppose that one worker has taken a medium-difficulty job to do, and the other has taken a hard job to do. The state in Ω, then, is:

$$\wr m, h, h \int \triangleleft [\![MID; J \| HARD; J]\!]$$

where MID is the procedure for a medium-difficulty process

$$\begin{array}{l} \texttt{try } [\texttt{getR}(m,1)] \texttt{ then } \{\texttt{hold}(20); \texttt{trace('Mid')}; \texttt{putR}(m,1)\} \\ \texttt{ etry } [\texttt{getR}(h,1)] \texttt{ then } \{\texttt{hold}(10); \texttt{trace('Mid')}; \texttt{putR}(h,1)\} \text{ '} \end{array}$$

and $HARD$ denotes

$$\texttt{getR}(h,1); \texttt{hold}(20); \texttt{trace('Hard')}; \texttt{putR}(h,1)$$

Define

$$\begin{array}{rcl} MID_h & \equiv & \texttt{hold}(10); \texttt{trace('Mid')}; \texttt{putR}(h,1) \\ HARD_h & \equiv & \texttt{hold}(20); \texttt{trace('Hard')}; \texttt{putR}(h,1), \end{array}$$

representing the code to be executed if we have a middle- or hard-difficulty job, respectively, and have picked up a hammer.

The derivation from this state, using the unsound transition rule, is presented in Figure 5.2.

$$\dfrac{\dfrac{\dfrac{}{\mathcal{FF}(h \lhd [\![g[\texttt{getR}(m,1)]]\!])}\ \lbag m \rbag \notin \lbag h \rbag \qquad \dfrac{\dfrac{}{\mathbf{R}h \lhd [\![\texttt{getR}(h,1)]\!] \xrightarrow{\sigma} \mathbf{R} \lhd [\![]\!]}\ (\mathrm{GET}^1_{\mathrm{R}})}{\mathbf{R}h \lhd [\![g[\texttt{getR}(h,1)]]\!] \xrightarrow{\sigma} \mathbf{R} \lhd [\![]\!]}\ (\mathrm{NBLOCK}^{\mathrm{T}}_{\mathrm{R}})}{\mathbf{R}h \lhd [\![MID]\!] \xrightarrow{\sigma} \mathbf{R} \lhd [\![MID_h]\!]}\ (\mathrm{TRY}^{\mathrm{F}}_{\mathrm{Te}}) \qquad \dfrac{\dfrac{}{\mathbf{R}m, h \lhd [\![\texttt{getR}(h,1)]\!] \xrightarrow{\sigma} \mathbf{R}m \lhd [\![]\!]}\ (\mathrm{GET}^1_{\mathrm{R}})}{\mathbf{R}m, h \lhd [\![HARD]\!] \xrightarrow{\sigma} \mathbf{R}m \lhd [\![HARD_h]\!]}\ (\mathrm{COMP}_{\mathrm{I}})}{\mathbf{R}h \lhd [\![MID; J]\!] \xrightarrow{\sigma} \mathbf{R} \lhd [\![MID_h; J]\!]}\ (\mathrm{COMP}_{\mathrm{I}}) \qquad \dfrac{}{\mathbf{R}m, h \lhd [\![HARD; J]\!] \xrightarrow{\sigma} \mathbf{R}m \lhd [\![HARD_h; J]\!]}\ (\mathrm{COMP}_{\mathrm{I}})}{\mathbf{R}m, h, h \lhd [\![MID; J \| HARD; J]\!] \xrightarrow{\sigma} \mathbf{R}m \lhd [\![MID_h; J \| HARD_h; J]\!]}\ (\mathrm{UNSOUND})$$

Figure 5.2: Derivation made with the unsound parallelisation rule

Clearly this is wrong — we did not use the mallet on the easy job as we should have done. The error came from the fact that it was possible for us to 'give' the mallet to the process executing $HARD$, which did not need it, whilst MID did not know that the mallet existed.

For this reason, a logical resource weakening rule for transitions,

$$(\text{WEAK}): \quad \frac{\mathbf{R} \lhd [\![P]\!] \xrightarrow{\sigma} \mathbf{R}' \lhd [\![P']\!]}{\mathbf{R} \uplus \mathbf{S} \lhd [\![P]\!] \xrightarrow{\sigma} \mathbf{R}' \uplus \mathbf{S} \lhd [\![P']\!]},$$

would also be unsound.

Two solutions exist to overcome this particular problem. One is to adopt the rule:

$$(\text{ SPAR }): \quad \frac{\mathbf{R} \uplus \mathbf{H} \lhd [\![P]\!] \xrightarrow{\sigma_1} \mathbf{H} \lhd [\![P']\!] \quad \mathbf{R}' \uplus \mathbf{H} \lhd [\![Q]\!] \xrightarrow{\sigma_2} \mathbf{H} \lhd [\![Q']\!]}{\mathbf{R} \uplus \mathbf{R}' \uplus \mathbf{H} \lhd [\![P \| Q]\!] \xrightarrow{\sigma_1 | \sigma_2} \mathbf{H} \lhd [\![P' \| Q']\!]}.$$

This allows both derivations to 'see' all the unused resources in the system.

The second solution is to disallow the command at the head of any process being '`try`' when considering the synchronous parallel rule. We shall adopt the first solution.

5.3 Soundness: Synchrony

The presentation so far has failed to take into one thing into account: we do not necessarily want to consider processes executing synchronously. Take, for example, the following processes:

$$\begin{aligned} P_1 &\stackrel{\text{def}}{=} \texttt{getR}(r,1); \texttt{getR}(s,1); \texttt{putR}(r,1); \texttt{putR}(s,1) \\ P_2 &\stackrel{\text{def}}{=} \texttt{do}(2)\{\texttt{putB}(b,1);\}\texttt{getR}(s,1); \texttt{getR}(r,1); \ldots \end{aligned}$$

The only path derivable from here using the synchronous rule is:

$$\begin{array}{rl} & \wr r, s \int \lhd [\![P_1 \| P_2]\!] \\ \stackrel{\text{getR|putB}}{\Longrightarrow} & \wr s, b \int \lhd \left[\!\!\left[\begin{array}{ll} & \texttt{getR}(s,1); \texttt{putR}(r,1); \texttt{putR}(s,1) \\ \| & \texttt{do}(1)\{\texttt{putB}(b,1);\}\texttt{getR}(s,1); \texttt{getR}(r,1); \ldots \end{array} \right]\!\!\right] \\ \stackrel{\text{getR|putB}}{\Longrightarrow} & \wr b, b \int \lhd \left[\!\!\left[\begin{array}{ll} & \texttt{putR}(r,1); \texttt{putR}(s,1) \\ \| & \texttt{getR}(s,1); \texttt{getR}(r,1); \ldots \end{array} \right]\!\!\right] \\ \Longrightarrow & \ldots \end{array}$$

By executing synchronously, we missed the deadlock that would have occurred if the two `putB`s had executed first. In effect, the transition system that we have described so far in this section could be simulated in the operational semantics by removing all `hold` commands and inserting $\texttt{hold}(k)$, for some constant k, between all the other commands.

How, then, can we reconcile our desire for a state space reduction with the necessity of considering all interactions between processes? The answer lies in

identifying which commands could affect the execution of other processes — we shall say that they *interfere* with other processes. We can choose to branch only on these commands, whilst executing non-interfering commands synchronously. So, for example, if a process has a series of commands on a resource that is not referenced by any other process, we will not introduce any new paths to be examined when model checking.

We shall, therefore, require a condition, $(*)$, that dictates when to use the synchronous parallel rule and when to use the asynchronous rule(s). To capture intereference some auxilliary functions are required. Firstly, the multiset of resources accessed by a non-parallel process:

$$\begin{aligned}
res(\texttt{getR}(r,n)) &\stackrel{\text{def}}{=} \lbag r^n \rbag \\
res(\texttt{putR}(r,n)) &\stackrel{\text{def}}{=} \lbag r \rbag \\
res(\texttt{getB}(b,n)) &\stackrel{\text{def}}{=} \lbag b^n \rbag \\
res(\texttt{putB}(b,n)) &\stackrel{\text{def}}{=} \lbag b \rbag \\
res(\texttt{sync}(x)) &\stackrel{\text{def}}{=} \lbag \texttt{sync_}x \rbag \\
res(\texttt{getS}(x,n)) &\stackrel{\text{def}}{=} \lbag \texttt{sync_}x^n \rbag \\
res(c;P) &\stackrel{\text{def}}{=} res(c) \uplus res(P) \\
res(c,C) &\stackrel{\text{def}}{=} res(c) \uplus res(C) \\
res\left(\begin{array}{l}\texttt{try } [e_1] \texttt{ then } P_1 \\ \texttt{etry } [e_2] \texttt{ then } P_2\end{array}\right) &\stackrel{\text{def}}{=} res(e_1) \uplus res(e_2) \uplus res(P_1) \uplus res(P_2) \\
res(\text{anything else}) &\stackrel{\text{def}}{=} \lbag\!\rbag
\end{aligned}$$

where $\texttt{sync\_}x$ is a token indicating synchronisation on a pool x.

The command that may next be executed by a non-parallel process is its head, defined as follows:

$$head(c) \stackrel{\text{def}}{=} c \qquad head(c;P) \stackrel{\text{def}}{=} c$$

The function $\overrightarrow{res}$ shall return the multiset of resources accessed by parallel processes:

$$\overrightarrow{res}(\llbracket \varepsilon \rrbracket) \stackrel{\text{def}}{=} \lbag\!\rbag$$

$$\overrightarrow{res}(\llbracket P \| Q \rrbracket) \stackrel{\text{def}}{=} res(P) \uplus \overrightarrow{res}(Q), \quad P \text{ not parallel}$$

Take the set of indices of a parallel process P to be In_P; that is, if $P = \llbracket P_1 \| \ldots \| P_n \rrbracket$ then $In_P = \{1, \ldots, n\}$. The set of indices of non-interfering processes (processes with non-interfering commands at their head) shall be written C'_P, defined as:

$$C'_P \stackrel{\text{def}}{=} \left\{ i \;\middle|\; res(head(P_i)) \pitchfork \overrightarrow{res}\left(\Big\| _{k \in I_P \setminus \{i\}} P_k \right) = \emptyset \right\}$$

To ensure that the synchronous parallel rule does not prevent us entering the true deadlocked state, we shall also require that it only be applied to processes that are able to act. This set of processes is:

$$A_P^{\mathbf{R}} \stackrel{\text{def}}{=} \{ i | i \in In_P \wedge (head(P_i) = \texttt{getR}(r,n) \implies res(head(P_i)) \trianglelefteq \mathbf{R}) \}$$

We shall execute all non-interfering processes synchronously until there are none left, at which point we shall use the non-deterministic asynchronous parallel rules. Thus the condition is:

$$C'_P \cap A^{\mathbf{R}}_P \neq \emptyset \qquad (*)$$

The synchronous parallel execution rule, then, is:

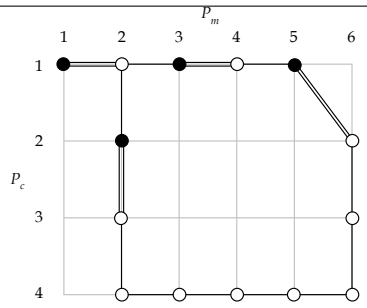
$$\text{(SPAR)}: \quad \frac{\mathbf{R}_i \uplus \mathbf{H} \lhd [\![P_i]\!] \xrightarrow{\sigma_i} \mathbf{H} \lhd [\![P'_i]\!]}{\mathbf{H}(\biguplus_{i \in I'_P} \mathbf{R}_i) \lhd [\![P]\!] \xrightarrow{\sigma_i | i \in I} \mathbf{H} \lhd \left[\!\!\left[\|_{k \notin I'_P} P_k \|_{i \in I'_P} P'_i \right]\!\!\right]} \ (*)$$

The asynchronous parallel execution rules are unchanged, other than having the condition $\neg(*)$.

In Figure 5.3, we present an example series of transitions derivable with this rule. Notice that there is no choice of which transition to derive — this particular path is derived completely deterministically from the given state. Following a τ-transition to remove the null process, we would now use the asynchronous transition rules to consider the interfering executions. Figure 5.4 shows ther reduced state space of the kitchen example. Notice that the it is much smaller than that derived by the asynchronous system, with only one branching point.

$$\begin{array}{rcl}
\wr r, s, t \int & \lhd & \left[\!\!\left[\begin{array}{l} \texttt{getB}(r,1) \\ \texttt{putR}(u,1) \\ \texttt{hold}(3) \end{array} \middle\| \begin{array}{l} \texttt{getR}(u,1) \\ \texttt{hold}(4) \\ \texttt{putR}(u,1) \end{array} \middle\| \begin{array}{l} \texttt{getR}(t,1) \\ \texttt{getR}(s,1) \\ \texttt{hold}(4) \\ \texttt{putR}(t,1) \\ \texttt{putR}(s,1) \end{array} \right]\!\!\right] \quad C'_P = \{1,3\} \\
\xrightarrow{\text{getB|getR}} \wr s \int & \lhd & \left[\!\!\left[\begin{array}{l} \\ \texttt{putR}(u,1) \\ \texttt{hold}(3) \end{array} \middle\| \begin{array}{l} \texttt{getR}(u,1) \\ \texttt{hold}(4) \\ \texttt{putR}(u,1) \end{array} \middle\| \begin{array}{l} \texttt{getR}(s,1) \\ \texttt{hold}(4) \\ \texttt{putR}(t,1) \\ \texttt{putR}(s,1) \end{array} \right]\!\!\right] \quad C'_P = \{3\} \\
\xrightarrow{\text{getR}} \wr \int & \lhd & \left[\!\!\left[\begin{array}{l} \\ \texttt{putR}(u,1) \\ \texttt{hold}(3) \end{array} \middle\| \begin{array}{l} \texttt{getR}(u,1) \\ \texttt{hold}(4) \\ \texttt{putR}(u,1) \end{array} \middle\| \begin{array}{l} \texttt{hold}(4) \\ \texttt{putR}(t,1) \\ \texttt{putR}(s,1) \end{array} \right]\!\!\right] \quad C'_P = \{3\} \\
\xrightarrow{\tau} \wr \int & \lhd & \left[\!\!\left[\begin{array}{l} \\ \texttt{putR}(u,1) \\ \texttt{hold}(3) \end{array} \middle\| \begin{array}{l} \texttt{getR}(u,1) \\ \texttt{hold}(4) \\ \texttt{putR}(u,1) \end{array} \middle\| \begin{array}{l} \\ \texttt{putR}(t,1) \\ \texttt{putR}(s,1) \end{array} \right]\!\!\right] \quad C'_P = \{3\} \\
\xrightarrow{\text{putR}} \wr t \int & \lhd & \left[\!\!\left[\begin{array}{l} \\ \texttt{putR}(u,1) \\ \texttt{hold}(3) \end{array} \middle\| \begin{array}{l} \texttt{getR}(u,1) \\ \texttt{hold}(4) \\ \texttt{putR}(u,1) \end{array} \middle\| \begin{array}{l} \\ \\ \texttt{putR}(s,1) \end{array} \right]\!\!\right] \quad C'_P = \{3\} \\
\xrightarrow{\text{putR}} \wr s, t \int & \lhd & \left[\!\!\left[\begin{array}{l} \\ \texttt{putR}(u,1) \\ \texttt{hold}(3) \end{array} \middle\| \begin{array}{l} \texttt{getR}(u,1) \\ \texttt{hold}(4) \\ \texttt{putR}(u,1) \end{array} \middle\| \varepsilon \right]\!\!\right] \quad I'_P = \{3\}
\end{array}$$

Figure 5.3: Path derived, deterministically, using synchronous parallel rule



Filled circles represent states in which $(*)$ holds. Double lines indicate a transition derived using the synchronous parallel rule.

Figure 5.4: State space for kitchen example derived by the synchronous transition system

As mentioned before, though, the reduced state space comes at a cost. The transition system is not complete with respect to the operational semantics of Demos programs because the operational semantics defines asynchronous action. In the sequel, $\models_a$ shall represent satisfaction in **PBI** with the normal, asynchronous semantics, and $\models_s$ shall represent satisfaction with the synchronous parallel rule. We see, then, that $\models_s$ is not complete with respect to $\models_a$.

Proposition 5.3.1 $\models_s$ is not complete with respect to $\models_a$. That is, there is a formula φ and σDemos state $\mathbf{R} \triangleleft [\![P]\!]$ with $\mathbf{R} \triangleleft [\![P]\!] \models_a \varphi$ and $\mathbf{R} \triangleleft [\![P]\!] \not\models_s \varphi$.

Proof Take $\mathbf{R} \triangleleft [\![P]\!]$ to be the first state in the sequence above. Then $\mathbf{R} \triangleleft [\![P]\!] \models_a \langle - \rangle_{\mathbf{any}} s * t * I_R$ but $\mathbf{R} \triangleleft [\![P]\!] \not\models_s \langle - \rangle_{\mathbf{any}} s * t * I_R$. □

Moreover, $\models_s$ is not even sound with respect to $\models_a$ (and, therefore, the operational semantics of Demos):

Proposition 5.3.2 $\models_s$ is not sound with respect to $\models_a$. That is, there is a formula φ and σDemos state $\mathbf{R} \triangleleft [\![P]\!]$ with $\mathbf{R} \triangleleft [\![P]\!] \models_s \varphi$ but $\mathbf{R} \triangleleft [\![P]\!] \not\models_a \varphi$.

Proof With $\mathbf{R} \triangleleft [\![P]\!]$ as before, $\mathbf{R} \triangleleft [\![P]\!] \not\models_a \neg(\langle - \rangle_{\mathbf{any}} s * t * I_R)$ but $\mathbf{R} \triangleleft [\![P]\!] \models_s \neg(\langle - \rangle_{\mathbf{any}} s * t * I_R)$. □

This holds even in a system without negation, *e.g.,* consider evaluating $\langle - \rangle(s * I_R)$ on the first state of the above sequence.

We postulate, though, that there is a syntactically defined class of formulæ, Φ_s, for which $\models_s$ is sound and complete. We anticipate that deadlock freedom, $\mathfrak{D}_f^D$, is in this class. Though the definition of this class is beyond us at the moment, we shall proceed to show that deadlock freedom is, indeed, sound and complete in the synchronous system.

5.4 Deadlock freedom equivalence in asynchronous and synchronous systems

Recall that the formula for deadlock freedom for Demos in our HML-type logic is:

$$\mathfrak{D}_f^D = \bigwedge_{i \in \mathbb{N} \cup \{\omega\}} [Act]_{\mathbf{any}}{}^i(\langle Act \rangle_{\mathbf{any}} \top \vee I_P).$$

This formula will be equivalent in both transition systems if

- any state reachable in the synchronous system that is deadlocked is also reachable in the asynchronous system, and
- any deadlocked state reachable in the asynchronous system is also reachable in the synchronous system.

The first condition relates to soundness of deadlock detection in the synchronous system (dually, completeness of deadlock freedom); the second relates to completeness of deadlock detection in the synchronous system (dually, soundness of deadlock freedom).

More formally, for arbitrary $\kappa, \lambda \in \mathbb{N}_\omega$ there exist κ', λ' such that:

$$\begin{array}{l} \text{If } \mathbf{R} \triangleleft [\![P]\!] \models_s \langle - \rangle^{\kappa}_{\mathbf{any}}(\neg(\langle - \rangle_{\mathbf{any}} \top \vee I_P)) \text{ then} \\ \mathbf{R} \triangleleft [\![P]\!] \models_a \langle - \rangle^{\kappa'}_{\mathbf{any}}(\neg(\langle - \rangle_{\mathbf{any}} \top \vee I_P)) \end{array} \tag{5.1}$$

$$\begin{array}{l} \text{If } \mathbf{R} \triangleleft [\![P]\!] \models_a \langle - \rangle^{\lambda}_{\mathbf{any}}(\neg(\langle - \rangle_{\mathbf{any}} \top \vee I_P)) \text{ then} \\ \mathbf{R} \triangleleft [\![P]\!] \models_s \langle - \rangle^{\lambda'}_{\mathbf{any}}(\neg(\langle - \rangle_{\mathbf{any}} \top \vee I_P)) \end{array} \tag{5.2}$$

5.4.1 Soundness of deadlock detection

In order to show the soundness of deadlock detection, we shall firstly show that any state reachable in one transition by the synchronous system is reachable by the asynchronous system.

Lemma 5.4.1 There exists a finite number of transitions, n, such that:

$$\text{If } \mathbf{R} \triangleleft [\![P]\!] \xrightarrow{\sigma}_s \mathbf{R}' \triangleleft [\![P']\!] \text{ then } \mathbf{R} \triangleleft [\![P]\!] \xrightarrow{\sigma}{}^{n}_{a} \mathbf{R}' \triangleleft [\![P']\!]$$

Proof *By induction on the structure of synchronous transition derivations.* The non-parallel cases are trivially true — in neither system do we use the parallel execution rules.

The interesting case is where we have a `try` command in parallel with some other process, as presented in §5.2.

The other cases for (SPAR) follow quite easily from the fact that we can 'decompose' the derivation using (SPAR) to form two asynchronous derivations. From the fact that the two processes do not interfere — otherwise, we would not be using (SPAR) — we can deduce that the resources necessary to take both actions will be available in the asynchronous case if they were in the synchronous case.

Returning to the `try` command case, assume that

$$\mathbf{R} \lhd [\![P]\!] \equiv \mathbf{S} \uplus \mathbf{S}' \uplus \mathbf{H} \lhd [\![\texttt{try}\ [e_1]\ \texttt{then}\ P_1\ \texttt{etry}\ [e_2]\ \texttt{then}\ P_2 \| Q]\!]\,.$$

If we are to derive a transition, we must have the sub-derivation $\mathbf{S} \uplus \mathbf{H} \lhd [\![\texttt{try}\ [e_1]\ \texttt{then}\ P_2\ \texttt{etry}\ [e_2]\ \texttt{then}\ P_2]\!] \xrightarrow{\text{try}} \mathbf{H} \lhd [\![P']\!]$, where $P' \equiv P_1$ or P_2. We know that neither e_1 nor e_2 require any resources referenced by process Q by $(*)$. Consequently, any transition from process Q will not use the resources required by e_1 or e_2, and so they are not contained in $\mathbf{S}'$. Therefore, any resources that could be used by the `try` command, thereby affecting the path chosen, will not be hidden by inclusion in $\mathbf{S}'$. Thus adding the resources in $\mathbf{S}'$ to the environment will not affect the transition, so we could equivalently derive:

$$\mathbf{S} \uplus \mathbf{S}' \uplus \mathbf{H} \lhd [\![\texttt{try} \ldots]\!] \xrightarrow{\text{try}} \mathbf{H} \uplus \mathbf{S}' \lhd [\![P']\!]$$

As this does not use a parallel execution rule — it uses (TRY), this holds for both systems. By induction, there exists an integer m such that

$$\mathbf{S}' \uplus \mathbf{H} \lhd [\![Q]\!] \xrightarrow{\sigma}{}^{m}_{a}\ \mathbf{H} \lhd [\![Q']\!]\,.$$

If, then, we apply the asynchronous parallel rule to first derive

$$\mathbf{S} \uplus \mathbf{S}' \uplus \mathbf{H} \lhd [\![\texttt{try} \ldots \| Q]\!] \xrightarrow{\text{try}} \mathbf{S}' \uplus \mathbf{H} \lhd [\![P' \| Q]\!]\,,$$

we can apply the same sequence of m transitions on Q to derive the required state. □

It follows easily by transfinite (ordinal) induction that any state reachable by a sequence of transitions in the synchronous system is reachable by a sequence of transitions in the asynchronous system.

Lemma 5.4.2 For any $\kappa \in \mathbb{N}_\omega$ there is a $\kappa' \in \mathbb{N}_\omega$ such that:

$$\text{If } \mathbf{R} \lhd [\![P]\!] \xrightarrow{\sigma}{}^{\kappa}_{s}\ \mathbf{R}' \lhd [\![P']\!] \text{ then } \mathbf{R} \lhd [\![P]\!] \xrightarrow{\sigma}{}^{\kappa'}_{a}\ \mathbf{R}' \lhd [\![P']\!]$$

□

We now prove that if deadlock is detected according to the synchronous transition system then the state really can deadlock.

Theorem 5.4.3 (Soundness of deadlock detection (5.1))

$$\mathbf{R} \lhd [\![P]\!] \models_s \langle - \rangle^{\kappa}_{\mathbf{any}} (\neg(\langle - \rangle_{\mathbf{any}} \top \vee I_P)) \implies \mathbf{R} \lhd [\![P]\!] \models_a \langle - \rangle^{\kappa}_{\mathbf{any}} (\neg(\langle - \rangle_{\mathbf{any}} \top \vee I_P))$$

Proof If we dissect the formula $\langle - \rangle^{\kappa}_{\mathbf{any}} \neg(\langle - \rangle_{\mathbf{any}} \top \vee I_P)$, it expresses that $\neg(\langle - \rangle_{\mathbf{any}} \top \vee I_P)$ holds in some state reachable from $\mathbf{R} \lhd [\![P]\!]$. Suppose, then, that the synchronous transition system leads to a state $\mathbf{R}' \lhd [\![P']\!] \models_s \neg(\langle - \rangle_{\mathbf{any}} \top \vee I_P)$. By Lemma 5.4.2, we know that we can reach this state in the asynchronous system. We must, then, show that $\mathbf{R}' \lhd [\![P']\!] \models_a \neg(\langle - \rangle_{\mathbf{any}} \top \vee I_P)$.

Firstly, notice that $\neg(\langle - \rangle_{\mathbf{any}} \top \vee I_P)$ is logically equivalent to $\neg\langle - \rangle_{\mathbf{any}} \top \wedge \neg I_P$. Consequently, from the semantics of $\wedge$, we have $\mathbf{R}' \lhd [\![P']\!] \models_s \neg I_P$. As this is not

a modal formula, the differing transition system will not affect its satisfaction, so $\mathbf{R}' \lhd [\![P']\!] \models_a \neg I_P$. All that remains to be proven, then, is that $\mathbf{R}' \lhd [\![P']\!] \models_s \neg\langle - \rangle_{\mathbf{any}} \top$ implies $\mathbf{R}' \lhd [\![P']\!] \models_a \neg\langle - \rangle_{\mathbf{any}} \top$. Assuming $\mathbf{R}' \lhd [\![P']\!] \models_s \neg\langle - \rangle_{\mathbf{any}} \top$, there are two cases to consider: $(*)$ holds and $(*)$ does not hold.

If $(*)$ does not hold, we are unable to derive a transition using the asynchronous parallel execution rules, and the synchronous parallel rule may not be used. Immediately, therefore, we are unable to derive any transition in the purely asynchronous system, so $\mathbf{R}' \lhd [\![P']\!] \models_a \neg\langle - \rangle_{\mathbf{any}} \top$.

If $(*)$ holds, we know that there is at least one process with a non-interfering command at its head. Furthermore, we know that if this command is a `get`X (where X could be `R`, `B` or `S`) command, the resource is available. There is no reason, therefore, why we cannot make a transition from this state for every process indexed by $C'_P \cap A^{\mathbf{R}}_P$. We may deduce that this case does not arise: $\mathbf{R}' \lhd [\![P']\!] \models_a \neg\langle - \rangle_{\mathbf{any}} \top$ is never true if $(*)$ holds, so we may vacuously conclude $\mathbf{R}' \lhd [\![P']\!] \models_s \neg\langle - \rangle_{\mathbf{any}} \top$. □

5.4.2 Completeness of deadlock detection

In order to show that the synchronous system is complete with respect to the asynchronous system — that is, if we detect deadlock in the asynchronous system, we will in the synchronous system — we shall first show that if it is not possible to make a transition in the asynchronous system from an arbitrary state, it is not possible to make a transition in the synchronous system.

Lemma 5.4.4 For all states $\mathbf{R} \lhd [\![P]\!]$,

$$\mathbf{R} \lhd [\![P]\!] \models_a \neg\langle - \rangle_{\mathbf{any}} \top \implies \mathbf{R} \lhd [\![P]\!] \models_s \neg\langle - \rangle_{\mathbf{any}} \top$$

Proof *By induction on* $\mathbf{R} \lhd [\![P]\!]$
Assume that $\mathbf{R} \lhd [\![P]\!] \models_a \neg\langle - \rangle_{\mathbf{any}} \top$. As usual, we shall consider only the case where P is the parallel composition of processes and $(*)$ holds. Otherwise, we would use exactly the same transition system and the statement would be easily provable. Then there is no derivation concluding, for any state $\mathbf{R}' \lhd [\![P']\!]$, $\mathbf{R} \lhd [\![P]\!] \xrightarrow{\sigma}_a \mathbf{R}' \lhd [\![P']\!]$. Consequently, as P is the parallel composition of processes, there is no transition derivable for any of these. In the synchronous system, $(*)$ holds, so we would require at least one derivation of a transition for a sub-process to derive a transition for P. By induction, we can make no transition for a sub-process in the synchronous system, so we can derive no transition for P in the synchronous system. [1] □

To prove the completeness of deadlock detection is quite complicated. We shall, therefore, present only an outline of the proof. Moreover, we shall consider only acquisition through `getR`; the cases for `getS` and `getB` are similar, as is acquisition through `try`.

[1] An alternative proof would note that this case is vacuously true: if $(*)$ holds, we must be able to make a transition, so we must be able to make a transition in the asynchronous system. We have given the full proof to illustrate how we could prove the property with a more relaxed condition $(*)$.

Theorem 5.4.5 (Completeness of deadlock detection (5.2)) If there is a path to a deadlocked state in the asynchrounous system, there is a path to a state in the synchronous system. That is, for arbitrary $\mathbf{R} \triangleleft [\![P]\!]$,

$$\mathbf{R} \triangleleft [\![P]\!] \models_a \langle - \rangle^{\kappa}_{\mathbf{any}} \neg (\langle - \rangle_{\mathbf{any}} \top \vee I_P) \implies \mathbf{R} \triangleleft [\![P]\!] \models_s \langle - \rangle^{\kappa}_{\mathbf{any}} \neg (\langle - \rangle_{\mathbf{any}} \top \vee I_P)$$

Proof Outline We start by assuming an arbitrary state $\mathbf{R}' \triangleleft [\![P']\!]$ such that $\mathbf{R}' \triangleleft [\![P']\!] \models_a \neg(\langle - \rangle_{\mathbf{any}} \top \vee I_P)$. This is equivalent to $\mathbf{R}' \triangleleft [\![P']\!] \models_a \neg\langle - \rangle_{\mathbf{any}} \top \wedge \neg I_P)$. By the semantics of $\wedge$, $\mathbf{R}' \triangleleft [\![P']\!] \models_a \neg\langle - \rangle_{\mathbf{any}} \top$ and $\mathbf{R}' \triangleleft [\![P']\!] \models_a \neg I_P$. By Lemma 5.4.4, we see that $\mathbf{R}' \triangleleft [\![P']\!] \models_s \neg\langle - \rangle_{\mathbf{any}} \top$. Whether I_P holds in a state is independent of the transition system, so $\mathbf{R}' \triangleleft [\![P']\!] \models_s I_P$.

Now, notice that the only rule we may fail to derive a transition by is (GETR); it is the only rule for which the side-condition[2] does not form a tautology with the other rules. As the state is deadlocked, the command at the head of every process, therefore, must be $\texttt{getR}(r_i, 1)$. Moreover, as the side condition is not met (because we are not able to make a transition from this state), we do not have any resource r_i that is requested.

We may only use the rule (SPAR) to execute certain commmands. It may not be used to pass any $\texttt{getR}(r_i, 1)$ command where either r_i is not available or r_i is referenced by some other process. Beyond here, the asynchronous system will be used. Notice that $(*)$ necessarily does not hold where we fail to derive a transition in the asynchronous system. We see, then, that

$$\begin{aligned} & \mathbf{R} \triangleleft [\![P]\!] \xrightarrow{\sigma^k}_a \mathbf{R}' \triangleleft [\![\texttt{getR}(r_1, 1) \| \ldots \| \texttt{getR}(r_n, 1)]\!] \\ \text{implies} \quad & \mathbf{R} \triangleleft [\![P]\!] \xrightarrow{\sigma^\ell}_s \mathbf{R}' \triangleleft [\![\texttt{getR}(r_1, 1) \| \ldots \| \texttt{getR}(r_n, 1)]\!] \end{aligned}$$

As $\neg(\langle - \rangle_{\mathbf{any}} \top \vee I_P)$ is equivalent in the two systems, then,

$$\mathbf{R} \triangleleft [\![P]\!] \models_a \langle - \rangle^{\kappa}_{\mathbf{any}} \neg (\langle - \rangle_{\mathbf{any}} \top \vee I_P).$$

□

It is interesting that in the above proofs we have not used the fact that we do not use the (SPAR) rule to pass by $\texttt{putR}(r, 1)$ commands where r is referenced by any other process. We could, therefore, remove this restriction and preserve the correctness of deadlock detection. Such a relaxation, though, would decrease the size of the class Φ_s of formulæ equivalent in the synchronous and asynchronous systems. The synchronous system would, for example, be incomplete in detecting whether any sub-process could ever have to wait on a resource.

It is also interesting to consider the behaviour of the synchronous parallel rule on the following program:

$$\begin{array}{l} \texttt{getR}(r, 1) \\ \texttt{getR}(s, 1) \end{array} \Bigg\| \begin{array}{l} \texttt{getR}(s, 1) \\ \texttt{getR}(r, 1) \end{array} \Bigg\| \begin{array}{l} \texttt{while}(\top)\{ \\ \quad \texttt{putB}(t, 1) \\ \quad \texttt{getB}(t, 1) \\ \} \end{array}$$

The synchronous execution rule, assuming a resource state $\wr r, s, t \int$, will continually 'execute' the third process. Though this is fine with respect to detecting

[2]The side-condition is implicit in the structure of the initial resource component

overall system deadlock, we will not be able to detect that the first two processes may become stuck: they will never pass their first commands, so they may both always make a transition.

Chapter 6

Conclusions

We have presented some of a theoretical framework capable of conducting analyses of discrete-event processes that interact through shared, discrete resources. This has involved taking the Logic of Bunched Implications and extending its semantics to be able to consider both processes and resources. In order to describe such processes, we have taken a simulation language, Demos, and defined its non-temporal action. Demos is an ideal language for this task: as well as being designed to model concurrently executing processes interacting through shared resources, its operational semantics are defined. This has allowed us to show that the transition system used by the logic that defines the action of processes is correct with respect to the non-temporal action of processes in Demos. We defined a kernel of Demos states capable of representing processes non-temporally.

6.1 Further work

A number of aspects remain to be considered.

Shared resource [1] Demos programs do not allow us to represent shared resource: we cannot represent, for example, that processes P_1 and P_2 have access to a resource whilst P_3 does not.[2] Our process logic, similarly, is not able to make such a distinction, and it is not clear to see how we may extend it to do so without significantly modifying the logic.

Fairness and justice We have allowed processes to execute in an arbitrary manner (*e.g.,* in §5.4.2 we allowed the third process to be executed all the time, ignoring the (blocked) first two processes). Such biased execution may be considered to model improperly the actual characterisics of a system. Following from the discussion of fairness in [GPSS80], Manna and Pnueli in [MP83] suggest two characteristics of proper executions: *fairness* and *justice*. We adapt their definition to our notion of concurrently executing processes.

[1] As suggested by D. J. Pym

[2] This could represent the resource having been taken by P_1 or P_2 which are "close to" each other, or the resource being generally unavailable to P_3.

- An execution sequence is said to be *fair* if it terminates finitely, it is infinite and all finite processes terminate or otherwise cannot continue to execute, or it is infinite and all non-halting processes are executed infinitely often.
- An execution sequence is said to be *just* if whenever a process is always able to take a transition, that process will act infinitely many times.

[CES86] describes how fairness may be integrated into a model-checker.

Adopting such a fairness condition in the example of §5.4.2 above would prevent the third process executing through every transition, and we could deduce that the system can deadlock. It appears particularly important to require such a notion of fairness when we consider a parallel execution rule that may favour particular processes, such as the one we defined.

Non-syntactic resource ownership Our approach would be more general had we not relied upon the fact that at a given state it is possible to deduce what resources are held by a process from the syntax of the process code. For example, if the following were allowed in Demos: `while [putB(r,1)]` (or even `try[putB(r,1)]`), we would not be able to deduce from the syntax the resources held by the process. It would, therefore, be necessary to keep a record of the resources held by each sub-process in the transition system. Just as processes explicitly hold other processes that are synchronised on them, we would have

$$\mathbf{R} \triangleleft \left[\!\!\left[\frac{P}{\wr r, s, t \wr} \right]\!\!\right]$$

representing a process P holding resources r, s and t. We chose not to do this in order to reduce the burden of notation.

Variables Most programmers will recognise the usefulness of variables, and they are, indeed, allowed in Demos 2000.[3] It would be a relatively simple matter to extend the transition system to take this into account. We would introduce a *variable state* component to take this into account. Transitions, then, would be of the form $\mathbf{R} \triangleleft [\![P]\!]\, s \xrightarrow{\sigma} \mathbf{R}' \triangleleft [\![P']\!]\, s'$. Variable state would be simply a map from variable labels to variable values, $s : Var \rightharpoonup Float$. We chose not to introduce this as our account is theoretical, and whilst it is not difficult to introduce variables into an implementation, proofs can become less succinct with their inclusion.

The semantics of **PBI** presented in this document have been defined with proof-search in mind. If we were to write a model-checker, though, it is anticipated that it will not be able to check all formulæ; for example, to check the formula

$$\mathbf{R} \triangleleft [\![P]\!] \models I_R \mathrel{-\!\!*} \varphi$$

could involve us considering adding all syntactically correct processes being placed in composition with P. Clearly, as the set of all processes is infinite,

[3] That is, numeric variables *e.g.*, $x := 4; y := 5$, rather than resource-variables.

such a simple implementation would lead to a non-complete model-checker. It would be acceptable, however, to provide a model checker complete for only a fragment of the logic. It may also be necessary to consider a logic other than the general Hennessy-Milner-style logic used here, similar to CTL or modal-μ.

Finally, we have not given a deduction system for our logic. Such a system should be proven correct (sound and complete) with respect to the semantics defined here.

To conclude, we have seen how we can extend the Logic of Bunched Implications to consider processes in a simple manner to achieve an expressive and clear way of formally reasoning about processes that interact on shared resources.

Bibliography

[AB96] Andrea Asperti and Nadia Busi. Mobile petri nets. Technical Report UBLCS-96-10, University of Bologna, Department of Computer Science, May 1996.

[BS01] Julian C. Bradfield and Colin Stirling. Modal logics and mu-calculi: an introduction. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 239–330. Elsevier, http://www.dcs.ed.ac.uk/∼jcb/, 2001.

[BTa] Graham Birtwistle and Chris Tofts. Getting demos models right. Part I: Practice.

[BTb] Graham Birtwistle and Chris Tofts. Getting demos models right Part II: . . . and theory.

[BT93] Graham Birtwistle and Chris Tofts. Operational semantics for process-based simulation languages. Part 1:πdemos. *Transactions of the Society for Computer Simulation*, 10(4):299 – 334, 1993.

[BT01] Graham Birtwistle and Chris Tofts. *DEMOS 2000 — A Semantically Justified Simulation Language*, August 2001. `http://www.demos2k.org`.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[FOL] The Free On-line Dictionary of Computing. `http://foldoc.doc.ic.ac.uk`.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[GPSS80] Dov M. Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173. ACM Press, 1980.

[HM80] Matthew C. B. Hennessy and Robin Milner. On observing nondeterminism and concurrency. In J. W. de Bakker and Jan van Leeuwen,

editors, *Automata, Languages and Programming, 7th Colloquium*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309, Noordweijkerhout, The Netherland, 14–18 July 1980. Springer-Verlag, New York, Berlin, Heidelberg.

[HM85] Matthew C. B. Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM (JACM)*, 32(1):137–161, 1985.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985. Series editor: C. A. R. Hoare.

[MBC+95] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Series in Parallel Computing. Wiley, 1995.

[Mil89] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall, 1989. Series editor: C. A. R. Hoare.

[MP83] Zohar Manna and Amir Pnueli. How to cook a temporal proof system for your pet language. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 141–154. ACM Press, 1983.

[MS] Faron Moller and Perdita Stevens. Edinburgh Concurrency Workbench user manual (version 7.1). Available from http://www.dcs.ed.ac.uk/home/cwb/.

[Mur89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[OED73] OED. *The Shorter Oxford English Dictionary*. Oxford University Press, third edition, 1973.

[OP99] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–243, June 1999.

[Pąc89] Pawel Pączkowski. Proving total correctness of concurrent programs without using auxilliary variables. Technical Report ECS-LFCS-89-100, University of Edinburgh, Laboratory for Foundations of Computer Science, November 1989.

[Per58] Alan J. Perlis. Announcement. *Communications of the ACM*, 1(1):1, 1958. Letter from Saul Gorn.

[Pit02] Andrew M. Pitts. Operational semantics and program equivalence. In G. Barthe, P. Dybjer, and J. Saraiva, editors, *Applied Semantics, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science, Tutorial*, pages 378–412. Springer-Verlag, 2002. International Summer School, APPSEM 2000, Caminha, Portugal, September 9–15, 2000.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, Denmark, Computer Science Department, 1981. `http://www.dcs.ed.ac.uk/home/gdp/`.

[Plo03] Gordon D. Plotkin. The origins of structural operational semantics. *Jornal of Functional and Logic Programming*, 2003.

[POY02] David J. Pym, Peter W. O'Hearn, and Hongseok Yang. Possible worlds and resources: The semantics of **BI**. *Journal of Theoretical Computer Science (to appear)*, 2002.

[Pym02] David J. Pym. *The Semantics and Proof Theory of the Logic of Buched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, Dordrecht, July 2002.

[TB94] Chris Tofts and Graham Birtwistle. Denotational semantics for process-based simulation languages. Part I: πdemos. Technical Report School of Computer Studies, University of Manchester, 1994.

[TB95] Chris Tofts and Graham Birtwistle. Denotational semantics for process-based simulation languages. Part I: μdemos. Technical Report School of Computer Studies, University of Manchester, 1995.

[TB01] Chris Tofts and Graham Birtwistle. Operational semantics of DEMOS 2000. Technical Report HPL-2001-263, Hewlett-Packard Laboratories, Graham Birtwistle, School of Computer Studies, University of Leeds, 2001.

[Tof01] Chris Tofts. Compiling DEMOS 2000 into Petri nets. Technical Report HPL-2001-274, Hewlett-Packard Laboratories, 2001.