Process Abstraction in the Verification of Temporal Properties

Glenn R. Bruns

Doctor of Philosophy

University of Edinburgh

1997

Abstract

The automatic verification of temporal properties of systems usually suffers from two problems. First, the size of the system that can be verified is very limited. Secondly, the results reflect only the behaviour of a system having particular parameters and initial conditions. Both problems are addressed by *abstracting* the system model relative to the property of interest. This thesis investigates two abstraction methods for processes.

In the first method unary process operators serve as abstraction operations. We show that an abstract process satisfies a property expressed as a temporal logic formula just if the original process satisfies a transformed formula. We define various abstraction operators and illustrate their use in verification with examples. The method is also used to derive two well-known verification techniques.

In the second method an abstract process and the original process are related by a process preorder. The weakly simulates and ready simulates preorders are used. For both we provide logical characterisations, abstraction operations, and algebraic laws.

Our work differs from existing work on process abstraction in that we abstract process expressions directly and take account of the particular property to be verified. We show the practical value of our methods by using them to help verify properties of Dekker's mutual exclusion algorithm and Ben-Ari's concurrent garbage collection algorithm.

Acknowledgements

I owe much to Colin Stirling, my thesis advisor. He was generous with his time, and taught me much about temporal logic and concurrency theory.

I was fortunate to do my thesis work at LFCS, a place full of talented and genial people. Jane Hillston very kindly reviewed a draft of my thesis in detail. Javier Esparza, Julian Bradfield, and Alex Simpson discussed topics related to the thesis with me. George Cleland was a frequent source of support and advice. Paul Anderson, Morna Findlay, and others at LFCS provided a stable computing environment.

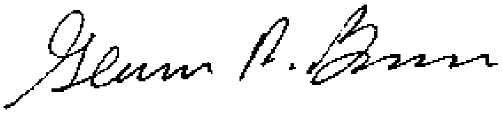
I am most grateful to those who have expressed confidence in my ability. John Heistand in particular was a great influence, as he sparked my interest in computing. His enthusiasm and humour is missed. Mark Ardis and Susan Gerhart enouraged me to pursue a PhD. For them I reserve a mixture of emotions, including gratitude.

Finally, I thank my friends at LFCS and elsewhere who helped me through the dips and bumps along the way. Renaud Marlet in particular has been a friend in need. My wife Jan and son Evan provided encouragement and moral support, but more importantly they helped me to remember the things that really matter. This thesis is dedicated to my parents, who have always been behind me.

Declaration

I declare that this thesis was composed by myself, and the work contained in it is my own unless otherwise noted.

Parts of this thesis have already been published elsewhere. An early version of Chapter 4 appeared as as [14]. An early version of part of Chapter 6 appeared in [38]. A model of Ben-Ari's algorithm similar to that of Section 8.5 appeared in [15].



Glenn R. Bruns

Table of Contents

Chapter 1

Introduction

A fundamental problem of computer science is the *verification problem*, which is to show that a program satisfies a behavioural property. Research on the problem has largely treated verification in a mathematical framework. Informal approaches suffer from ambiguity in the description of program properties and from errors in arguments about the satisfaction of properties. The problem has proved to be difficult. After thirty years of work on formal verification it is rarely used in the industrial development of systems, even those that put human life at risk.

Early research on program verification focussed on *transformational* systems, which accept some inputs and produce an output. Here verification involves showing that a program terminates and that on termination a desired relation holds between program inputs and output. For example, verifying a sort program involves showing that the output is really sorted. Much recent research concerns *reactive* systems, which interact continuously with their environment. Examples of reactive systems are control systems and communication protocols. Here verification involves showing that a program satisfies properties of its ongoing input/output behaviour. For example, in verifying the software in a flight control system we might want to show that if the system receives a landing-gear-down instruction then the proper signals will be sent to the landing gear. Concurrent programs can be transformational or reactive, but in their analysis one usually treats their components as reactive. These programs are particularly hard to verify because of the many possible interactions between the components.

To verify systems formally we need precise notation for describing systems and their properties. *Process algebra* is a suitable notation for describing reactive systems. Here a process is built up from *actions*, which model real-world events, and *operators*, which control the sequencing of actions. Especially important is the operator for concurrent composition, which allows the behaviour of a system to be described in terms of the behaviour of its components. *Temporal logic* is a suitable notation for describing the behavioural properties of reactive systems. In temporal logic one can express that a property always holds, that a property eventually holds, and that one property holds until another does. For example, the property that the landing-gear-down instruction is responded to properly can be expressed in temporal logic.

A connection between process algebra and temporal logic is that both can be defined in terms of the same underlying mathematical structures. The meaning of a process expression can be given as a *labelled transition system* (LTS), which is a directed graph with edges labelled by actions. The nodes of an LTS represent process states, which are processes themselves, and the edges represent transitions between states. A process is *finite-state* if its LTS has only finitely-many states. The meaning of a temporal logic formula can be given as the set of LTS's that satisfy the formula. A process then satisfies a temporal logic formula if its LTS is in the set denoted by the formula.

An advance of the past ten years has been the development of *automatic verification* systems, which compute whether a finite-state process satisfies a temporal logic formula. The classical scheme for automatic verification is simple (see Figure 1–1). It consists of a *model builder* that accepts a process and produces a data structure that represents an LTS, and a *model checker* [20], which accepts this LTS and a temporal logic formula, and produces *true* if the LTS satisfies the formula, and *false* otherwise.

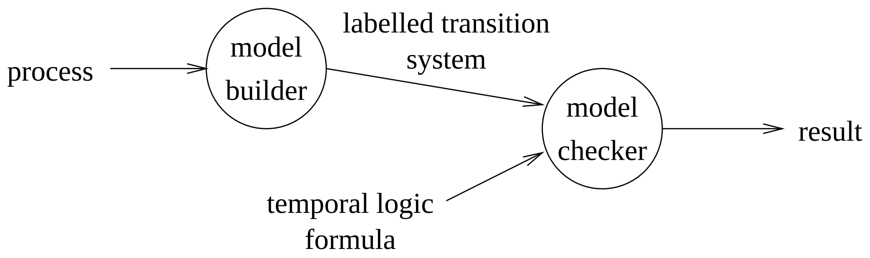


Figure 1–1: Classical Automatic Verification

These tools are now widely used for the automatic verification of communication protocols and other distributed systems. Two main questions guiding research on automatic verification are how larger systems can be verified and how the generality of verification results can be improved. The approach proposed in this thesis is to simplify processes relative to a particular temporal logic formula. We take processes to be terms of the process algebra CCS [72] and temporal logic formulas to be formulas of the modal mu-calculus [59].

The work in this thesis was largely motivated by practical problems in verifying properties of systems. Therefore we have tried to make the thesis accessible to those whose primary interests are also practical. We have especially tried to introduce the modal mu-calculus carefully, as there is little introductory material available about it. Some readers may find supplementary material on CCS [72] and the modal mu-calculus [88] useful.

1.1 Automatic Verification in System Design

To appreciate the value and limitations of automatic verification it is useful to understand how process algebra and temporal logic are used in the design of systems. Take for example the design of computer systems. These systems are often organised as a set of components connected via a bus or high-speed network. Each component is *synchronous*, with its sub-components running in lock-step according to a clock.

The components interact asynchronously through protocols, such as communication protocols and cache-consistency protocols. As CCS processes run asynchronously, CCS is not very useful in the design of the components, but is useful in the design of the protocols. Indeed, the CCS-based notation LOTOS [9] is an international standard for the description of communication protocols.

Consider the process of designing a protocol. One usually begins with an informal description of some of the properties required of the protocol. For example, a property required of a shared-memory communication protocol might be that no processor may read data that is currently being written. The goal of design is to describe the components of the protocol such that, when assembled, the protocol will satisfy these global properties. The behaviour of the components may be described using transition diagrams, basic process notations like CCS, or higher-level process notations like value-passing CCS and LOTOS. Ambiguities and omissions in the informal statement of the properties are often discovered as design proceeds. When the design process has ended, one should have a clear idea of the properties the design satisfies and an understanding of how the components of the protocol work together to satisfy the properties.

Automatic verification tools can be used in the analysis of the design once the design has been expressed in the required process notation and the behavioural properties expressed in the required temporal logic. The most useful tools do not just check whether a design satisfies its properties, but also give diagnostic information when a property is not satisfied.

1.2 Improving Automatic Verification

Despite its acceptance, automatic verification has some well-known problems that limit its application. The *size problem* is that the LTS for even simple processes can be huge. In particular, the state space of a process can grow exponentially with the size of its components. This is known as the *state explosion* problem. Thus a system with 10 components, each having only 10 states, can have up to 10^{10} states, far exceeding the abilities of classical automatic verification systems. A second problem is the time complexity of model checking. Depending on the temporal logic, the complexity can grow exponentially with the length of the formula. Finally, automatic verification suffers from the *generality problem*, which is that verifying a property of a particular finite-state process says nothing about closely-related processes. For example, verifying that a channel with the capacity to buffer two messages satisfies a property tells us nothing about a channel with capacity three.

One approach to the size problem is the development of more efficient model-checking algorithms. Many model-checking algorithms have been developed for the modal mu-calculus [36,24,30,67]. The algorithms differ by their space and time complexity, whether they are able to check formulas of the entire modal mu-calculus or just a fragment, and whether they use the entire LTS (global model checking) or just the part that is relevant to showing that a state satisfies a formula (local model checking).

Another approach to the size problem is to reduce the size of the LTS before model checking. A general technique for this is to find an equivalence on LTS's that preserves the temporal property of interest, and then to minimize the LTS with respect to this equivalence. There are many results that describe the properties preserved by bisimulation relations [62,83,52,6,88,19]. For example, in [88] it is shown that two bisimilar states satisfy the same set of modal mu-calculus formulas. Other results describe the properties preserved by equivalences induced by symmetries in processes and LTS's [37,21]. To minimise an LTS with respect to an equivalence, one partitions the states of the LTS by the equivalence [57], and then takes each equivalence class to be a state of the minimised LTS. A different technique for reducing the size of an LTS is to eliminate transitions of it by considering the independence of transitions in the concurrent components of the process [92,44].

The size of an LTS can also be reduced by using a more efficient representation. In classical automatic verification, the transition relation of an LTS is represented by enumerating the elements of the relation. This representation is wasteful if the transition relation exhibits much structure. An alternative is to represent the relation by its characteristic function, which can in turn be represented as a *Binary Decision Diagram* (BDD) [16], a normal form representation of boolean functions. This method is a heuristic one that can save much space compared to simpler representations but is not guaranteed to. Model-checking algorithms have been developed that work directly on BDD's [17]. An LTS can also be represented with the aid of *trace automata* [45]. A trace automaton is a kind of reduced LTS. Taken together with a relation that represents the independence between transitions of components in a process, a trace automaton represents the full LTS. However, for the purpose of finding whether a state is reachable it is enough to use the trace automaton alone.

It is wasteful to build the full LTS for a process if only part of the LTS is needed to determine if the process satisfies a particular property. Similarly, it is wasteful to build the LTS if it will be immediately minimised with respect to some equivalence. In *on-the-fly* methods, model checking or minimisation is done while the LTS is being built. Methods for on-the-fly model checking are discussed in [39,40,56]; a method for on-the-fly minimisation is discussed in [10]. Proof systems for local model checking [89,24,96], which explore only the states relevant to the property being checked, are naturally suited for on-the-fly model-checking algorithms.

Sometimes one can avoid building an LTS representing the joint behaviour of all components of a process. In *compositional* approaches [31,85,65,2] a property of a process is verified by verifying properties of its components. In [22] a component of a process can be simplified if the property to be checked does not directly concern it.

In *abstraction* methods the process to be verified is transformed relative to the property to be proved such that if the transformed process satisfies the temporal logic formula of interest, then so does the original process. For example, one can simplify a process using algebraic laws for a process equivalence. If a process is simplified using the algebraic laws of CCS, then the resulting abstract process will be bisimilar to the original one, and will satisfy many of the same properties.

Notable examples of abstraction methods are for value-passing processes and for processes with replicated components. For value-passing processes the goal is to show that if a process with a certain data domain satisfies a property, then processes with larger domains also satisfy it [97]. For processes with replicated copies of a component the goal is to show that if a process with a fixed number of components satisfies a property, then processes with more copies also satisfy it [84,98,60,13,61, 42]. Abstraction methods are important because they not only address the size problem of automatic verification, but also the generality problem. Furthermore, they can be used independently of techniques to improve model building and model checking.

1.3 Abstraction with Process Preorders

The abstraction problem addressed in this thesis is how to abstract a process relative to a temporal logic formula. In one technique explored here we take as abstract any process that lies in a certain relation to the process of interest. Let E and F stand for processes in an LTS. If $\sqsubseteq$ is a preorder on processes and $E \models \phi$ means that E satisfies temporal logic formula ϕ, then $\sqsubseteq$ *preserves* a temporal logic formula ϕ if, for all E and F

$$(F \models \phi \text{ and } E \sqsubseteq F) \Rightarrow E \models \phi.$$

Here E is the process of interest and F is an abstract version of E. If F satisfies ϕ, then so does E. However, it may be that F does not satisfy ϕ even though E does. Therefore we must choose our abstract process with care.

As an example we take a mutual exclusion system, which is used to ensure that at most one user has access to a shared resource at any time. Figure 1–2 shows a labelled transition system describing the behaviour of a simple two-user mutual exclusion system. Action r_i represents a request by the ith process for use of the shared resource. Action en_i represents the entrance by the ith process to its critical section, in which the shared resource is accessed. Action ex_i represents the exit by the ith process from its critical section. The state labelled s is the initial system state. The left-hand side of the transition system covers the case where process 1 makes a request before process 2; the right-hand side covers the opposite case.

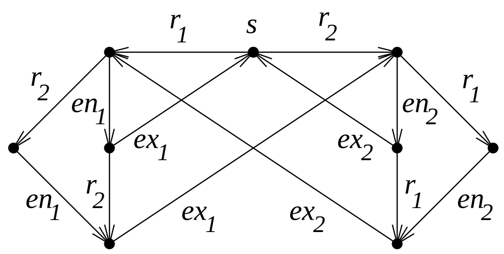
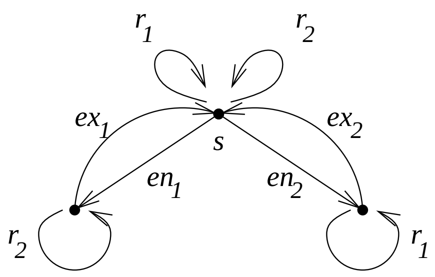


Figure 1–2: A Model of the Mutual Exclusion System

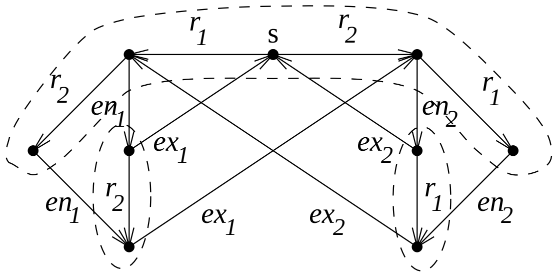
The most important property of the example is that of *mutual exclusion*: at most one process can be in its critical section at a time. In terms of the actions in the labelled transition system, this means that enter and exit actions must happen in alternation. Another important property is that requests must eventually be granted. If one process requests entry to its critical section then the other process cannot continually enter and exit its own critical section without the first process entering its critical section. These two properties can be expressed as temporal logic formulas. By examining Fig. 1–2 one can see that the system model satisfies both the properties.

To abstract the example we use the *simulation* preorder. Informally, a process F simulates a process E if every action of E can be matched by an action of F such that the process reached by F simulates the process reached by E. The property of mutual exclusion is preserved by this preorder. To prove mutual exclusion of the example we can therefore use the following abstract process (s is again the initial state).



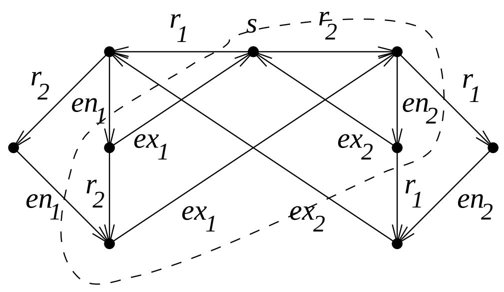
This LTS simulates the example, and satisfies the mutual exclusion property, so the example satisfies it too.

Finding an abstract LTS and checking that it simulates the concrete LTS can be difficult, especially for LTS's with hundreds or thousands of states. It is easier to build an abstract model by applying *abstraction operations* to the concrete model. One abstraction operation is the merging of states in an LTS. An abstract model obtained by merging states is guaranteed to simulate the concrete model. The abstract model just shown can be obtained from the example by merging states as shown here.



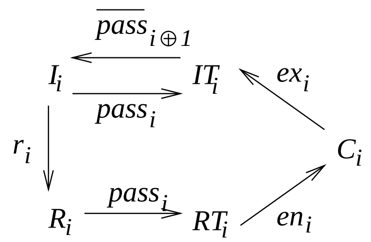
The states of each group are merged to form a single state of the abstract model. For each transition from a state in a group there is a transition from the new merged state. Similarly, for each transition to a state in a group there is a transition to the new merged state.

Next we consider the second property of the example, that requests are eventually granted. Consider the particular case in which process 1 makes a request. To prove this property, we can merge the states that process 1 can pass through between entering its critical section and making a subsequent request.



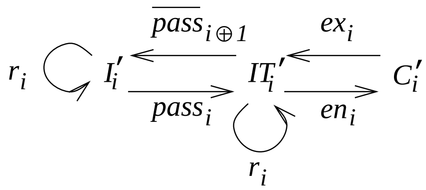
Merging the states gives the abstract model. Requests of process 1 are granted here, so they must be in the concrete model.

Abstracting from a concrete model representing the global behaviour of a system is unrealistic because designers rarely work with such models. For most systems they are too complicated to describe directly. Instead, models are built by combining sub-models representing the behaviour of system components. A mutual exclusion system for two users can be described using two components. The ith component has the following LTS, where $\oplus$ stands for addition modulo 2.



State I_i is so named because it represents process i in its *idle* state. It can request entry to its critical section (and so become process R_i), and can synchronise on action $\texttt{pass}_i$ to receive a "token" from the other process (and so become process IT_i). The other process states are IT (idle, with token), R (requested entry), RT (requested entry, with token), and C (in critical section). No $\texttt{r}_i$ transition is allowed from state IT_i to state RT_i because such a transition would let process i continue to enter its critical section even if process $i \oplus 1$ had requested entry to its own critical section.

The operation of merging states can be used to abstract components individually. Merging states of component i in which the token is not held and those in which the token is held yields this abstract component.



The simulation relation is preserved by composition of components, so the complete abstract model simulates the concrete model.

Sometimes components of a design are more conveniently described in a process notation. The mutual exclusion system component shown above can be written as the following CCS process.

$$\begin{aligned}
I_i &\stackrel{\text{def}}{=} \mathtt{r}_i\,.R_i + \mathtt{pass}_i\,.IT_i \\
IT_i &\stackrel{\text{def}}{=} \overline{\mathtt{pass}}_{i\oplus 1}\,.I_i \\
R_i &\stackrel{\text{def}}{=} \mathtt{pass}_i\,.RT_i \\
RT_i &\stackrel{\text{def}}{=} \mathtt{en}_i\,.C_i \\
C_i &\stackrel{\text{def}}{=} \mathtt{ex}_i\,.IT_i
\end{aligned}$$

CCS processes are built up from a small set of process operators. Process $a.E$ can perform action a and thereby become process E. Process $E+F$ can perform actions of either process E or F. Process definitions have the form $A \stackrel{\text{def}}{=} E$, where A is an atomic process *constant*.

To combine two such components we write

$$MES \stackrel{\text{def}}{=} (IT_1 \mid I_2)\backslash\{\mathtt{pass}\}.$$

Process $E \mid F$ is the concurrent composition of processes E and F. Composed processes can synchronise on complementary actions of the components, like $\mathtt{pass}_1$

and $\overline{\texttt{pass}}_1$. Process $E\backslash L$, where L is a set of actions, can perform any action a of E unless a or $\overline{a}$ is contained in L. The operation of merging states can be defined as a transformation on the set of process definitions. By applying a merging operation to the CCS process I_i above we obtain

$$\begin{aligned} I_i' &\stackrel{\text{def}}{=} \texttt{r}_i\,.I_i' + \texttt{pass}_i\,.IT_i' \\ IT_i' &\stackrel{\text{def}}{=} \overline{\texttt{pass}}_{i\oplus 1}\,.I_i' + \texttt{en}_i\,.C_i' \\ C_i' &\stackrel{\text{def}}{=} \texttt{ex}_i\,.IT_i' \end{aligned}$$

Since process composition preserves the simulation relation we know that

$$MES' \stackrel{\text{def}}{=} (IT_1' \mid I_2')\backslash\{pass\}$$

simulates MES.

Similar techniques for abstraction have been investigated. Lynch [69] abstracts processes written as I/O automata [68] using the simulation preorder, but does not provide abstraction operations. A simulation relation must be invented and checked. The logical effect of abstraction with simulation is not given; it is only stated that simulation is a stronger relation than trace inclusion. Clarke *et al* [19] abstract finite-state, procedural programs by mappings on program inputs and outputs. No abstraction operations on control structure are given. The technique is justified by showing that a kind of homomorphic mapping on transition systems preserves formulas of the temporal logic CTL* [35] with only the universal path quantifier. This mapping is stronger than the simulation relation, and admits only operations that immediately reduce the state space.

At the beginning of this section we defined what it means for a process preorder to preserve a temporal logic formula. Another view of abstraction comes from looking at the contrapositive of the condition given in that definition.

$$(E \not\models \phi \text{ and } E \sqsubseteq F) \Rightarrow F \not\models \phi$$

Let process E here be the abstract version of process F. Then the formula says that if the abstract process does not satisfy ϕ, then the concrete process does not satisfy it either. This abstraction idea can be used to show that a process has an error by showing that an abstract version has the error.

1.4 Abstraction with Process Operators

The other abstraction technique explored in this thesis is based on a different model-theoretic idea. Instead of using abstraction operators that preserve a property, we see how abstraction operators transform a property. Let $\mathcal{O}$ be a unary abstraction operator on processes. We write $\mathcal{O}(E)$ for the application of $\mathcal{O}$ to process E. Function $\mathcal{W}_{\mathcal{O}}$ is a *property transformer* for $\mathcal{O}$ if

$$\mathcal{O}(E) \models \phi \Leftrightarrow E \models \mathcal{W}_{\mathcal{O}}(\phi).$$

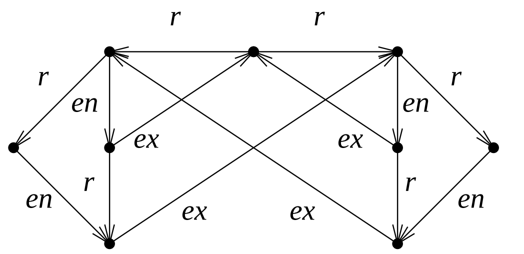
The transformed property $\mathcal{W}_{\mathcal{O}}(\phi)$ holds of the concrete process E if ϕ holds of the abstract process $\mathcal{O}(E)$.

The notation $\mathcal{W}$ was chosen to suggest a connection with Dijkstra's weakest precondition transformer [32]. That transformer gives the weakest predicate that must be satisfied if a program is to satisfy a given postcondition on termination. Let S_1 and S_2 be statements in a sequential programming language, and let ; be sequential composition. The weakest precondition for $S_1; S_2$ and postcondition ϕ is defined as

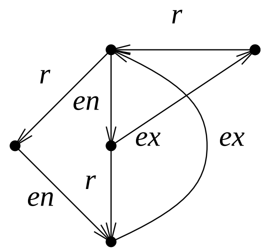
$$\mathrm{wp}(S_1; S_2, \phi) = \mathrm{wp}(S1, \mathrm{wp}(S2, \phi)).$$

More closely related to this thesis is the work of Larsen and Liu [65]. They define processes in terms of *contexts*, and show how a context transforms properties expressed in the modal mu-calculus.

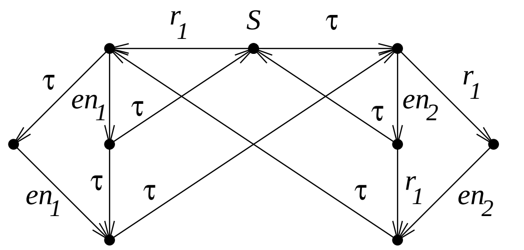
To abstract our example with our second abstraction technique, we take as an abstraction operation the function that removes indices from transition labels. The abstract mutual exclusion model is then as follows.



By transforming the property of mutual exclusion we find that this abstract LTS satisfies mutual exclusion just if the concrete LTS does. Abstraction has not made the model smaller, but has made it symmetrical. Folding the symmetrical model in half yields a model that satisfies the same properties.



Now suppose we want to prove that requests of process 1 are granted. Here the relevant actions are the request actions of process 1 and the enter actions of both processes. As an abstraction function we take the function that maps these events to themselves, and all other events to the action τ, which represents internal activity of a system.



The abstract model again has just as many states as the concrete model. However, further simplification is possible using special properties of τ actions.

As with the previous technique, we seek to apply this abstraction idea to components and to process expressions. CCS has a *relabelling* operator that maps actions to actions. The mutual exclusion process with indices removed from actions is written as:

$$MES[\mathtt{r}/\mathtt{r}_1, \mathtt{r}/\mathtt{r}_2, \mathtt{en}/\mathtt{en}_1, \mathtt{en}/\mathtt{en}_2, \mathtt{ex}/\mathtt{ex}_1, \mathtt{ex}/\mathtt{ex}_2]$$

The bracketed list of pairs is the relabelling operator that maps $\mathtt{r}_1$ and $\mathtt{r}_2$ to $\mathtt{r}$, and similarly for `en` and `ex` actions. Using algebraic laws for CCS the operator can be moved inside the definition of *MES*. For example, a law for relabelling says that $(a.E)[f]$ and $f(a).(E[f])$ are equivalent. The mapping of actions to τ actions can also be accomplished with process operators.

1.5 Synopsis

To summarise, this thesis examines two abstraction techniques that are intended to aid in the practical application of automatic verification methods to system design. The first is based on the use of property-preserving relations on processes; the second is based on treating abstraction operations as property transformers. Both methods can be applied to individual components of a design, and to process expressions. We demonstrate the usefulness of the techniques by applying them to realistic problems.

In Chapter 2 we describe CCS. First a general semantic theory of processes is described. The main elements of this theory are bisimulation equivalence and the simulation preorder. Then the particular theory of CCS is defined. We show that the strong and weak simulation preorders preserve all CCS operators.

In Chapter 3 we describe three modal logics for processes: Hennessy-Milner logic, dynamic logic, and the modal mu-calculus. A theme of the chapter is the relative ability of the logics to express properties of systems. Varieties of the modal mu-calculus are defined, and we show that the simulation preorders on processes preserve diamond-free formulas of the modal mu-calculus.

In Chapter 4 we describe abstraction with process operators. We define a class of static, unary process operators to be used as abstraction operators. For each such operator we define a transformer on modal mu-calculus formulas, and show that an abstract process satisfies a formula just if the concrete process satisfies the transformed formula. Three particular kinds of abstraction operators are defined: generalized relabelling operators, abstraction criteria, and priority operators. We illustrate their use in verification with examples.

In Chapter 5 we attempt to show the fundamental nature of this abstraction technique by deriving other verification techniques from it. The Interface Law of [22] allows components of a process to be simplified in verifying certain properties of it. We derive two versions of the Interface Law and illustrate their use on an n-user arbiter. A related verification idea is to hide actions of a process not mentioned in a formula. Using our abstraction technique we show that this idea is sound if the formula is a weak, diamond free modal mu-calculus property.

Results on data-independent processes [97] show how the data domain of a value-passing process can be abstracted to obtain a smaller domain. We define a CCS-based notion of data-independence and show how results on data-independent processes are easily derived from our abstraction technique.

In Chapter 6 we describe abstraction with the weak simulation preorder. We define algebraic laws for this preorder that can be used as abstraction operations. For example, one law allows abstraction by hiding the synchronisation actions in components of a process. Another allows process constants in a family of definitions to be merged. We show how the operations can be used to abstract a task scheduling process.

Chapter 7 is an extended example. We show how the techniques of chapters 4 and 6 can be used in verifying properties of Dekker's mutual exclusion algorithm. We first abstract the algorithm relative to the property that at most one process can be in its critical section at any time. Then we abstract the algorithm relative to the property that one process cannot continue to enter its critical section while the other process is waiting to enter its own critical section.

In Chapter 8 we describe abstraction with the anonymous ready simulates preorder. We first define this preorder and give it a logical characterization. The operation we use to abstract a process is to attach priorities to its actions. We show that a process ready simulates an abstract version of it if all instances of an action in the abstract process have the same priority. We also present algebraic laws for simplifying processes with priorities. The technique is used in verifying that a version of Ben-Ari's concurrent garbage collection algorithm is unsafe. It is also used to show that Dekker's algorithm fails to satisfy a liveness property.

In Chapter 9 we summarise the results of the thesis and discuss topics for further work.

Chapter 2

CCS Processes

In this chapter we describe the process algebra CCS and some of the main process equivalences and preorders that have been defined for it. We also describe a value-passing version of CCS, in which processes and actions can be parametrised with values.

2.1 Background

The study of concurrency was originally motivated by work in operating systems. Early operating systems ran user jobs in “batch” mode, one after the other. However, batch processing wastes computer resources. For example, the processor is mostly idle while data is input. To reduce waste, multiprogramming was introduced. Here, several user and system jobs are run “simultaneously” on a single computer.

Multiprogramming exposed programmers to new problems. For example, a problem that can arise in scheduling jobs is that a job waits indefinitely because jobs with higher priority are always present. A problem that can arise in sharing resources between jobs is that each of two jobs waits for the other to release a resource before proceeding.

It was also recognised that these features were not unique to multiprogramming operating systems, but also found in systems having several computers, and even in reactive systems, the purpose of which is ongoing interaction. A distinct field of concurrency emerged, in which "processes" with overlapping lifetimes are studied.

Robin Milner developed CCS to provide a basis for a mathematical study of concurrency. An analogy can be made to the lambda calculus, which has served well as a basis for a study of functions. The intention behind both the lambda calculus and CCS is to find a minimal set of fundamental ingredients and then to derive everything else. For example, it is possible to define the natural numbers as functions in the lambda calculus, and to define sequential composition as a special kind of concurrent composition in CCS.

Over the past fifteen years, a large body of theory has been built on CCS and similar process notations such as CSP [54] and ACP [7]. The basic notations have been extended to include time [73,80,3], probability [18,91], dynamic network topology [71], and value-passing [51,23]. Connections to the lambda calculus have been made [82], and yet more fundamental approaches are being developed. However, we will mostly use basic CCS. It is simple, it is concrete, it has a clear connection to concurrent programs, and it is widely known.

2.2 A Semantic Theory of Processes

The theory of CCS has two parts. One part concerns the semantic modelling of processes. The other concerns process notation and algebraic laws for manipulating process expressions. The first part of the theory is general and can be applied to other notations. Furthermore, it is easier to describe and understand if separated from the process notation. Therefore, we begin by looking at the semantic underpinnings of CCS, leaving the notational aspects for the next section.

The models for CCS processes are *labelled transitions systems* (LTS's). An LTS is a structure of the form

$$(S, A, \{\xrightarrow{a} \mid a \in A\})$$

where S is a set of states, A is a set of transition labels, and $\{\xrightarrow{a} \mid a \in A\}$ is a set of transition relations, with $\xrightarrow{a} \subseteq S \times S$ for each a in A. The states represent system states, the labels represent events, and the transitions represent the change of state associated with the occurrence of events. We refer to the states as *processes* and the labels as *actions*. We let $E, F, \ldots$ range over processes and let $a, b, \ldots$ range over actions.

We use the following abbreviations with transition relation $\xrightarrow{a}$:

$$\begin{aligned} E \xrightarrow{a} E' &\stackrel{\text{def}}{=} (E, E') \in \xrightarrow{a} \\ E \xrightarrow{a} &\stackrel{\text{def}}{=} \text{exists } E' \text{ such that } E \xrightarrow{a} E' \\ E \rightarrow &\stackrel{\text{def}}{=} \text{exists } a, E' \text{ such that } E \xrightarrow{a} E' \end{aligned}$$

We write $E \stackrel{a}{\not\rightarrow} E'$, $E \stackrel{a}{\not\rightarrow}$, and $E \not\rightarrow$ respectively for the negation of these notions. Figure 1–2 of the previous chapter shows an LTS representing the behaviour of a system for mutual exclusion.

Finite-state automata (FSA) are another common semantic model for processes, so it is worth mentioning some of the differences between FSA and LTS's. First, an FSA has finitely-many states, while an LTS may have infinitely-many states. Secondly, an FSA has a designated start state and some designated final states, while an LTS has neither. Finally, FSA are usually compared with respect to the languages they generate, while finer equivalences are commonly used for LTS's.

In considering a process E we are usually interested only in the part of the LTS containing processes reachable from E. We define a set $\mathcal{E}$ of processes to be *transition-closed* if $E \in \mathcal{E}$ and $E \xrightarrow{a} E'$ implies $E' \in \mathcal{E}$. The set of processes *reachable* from E, denoted $\mathrm{Rch}(E)$, is then the smallest transition-closed set of processes containing E. A process E is *finite-state* if the set $\mathrm{Rch}(E)$ is finite. A process E is *image-finite* if, for all a in A and for all F in $\mathrm{Rch}(E)$, the set of a-derivatives of F is finite.

A central topic of concurrency theory has been process equivalences based on observable behaviour. The conception of processes as states in an LTS suggests

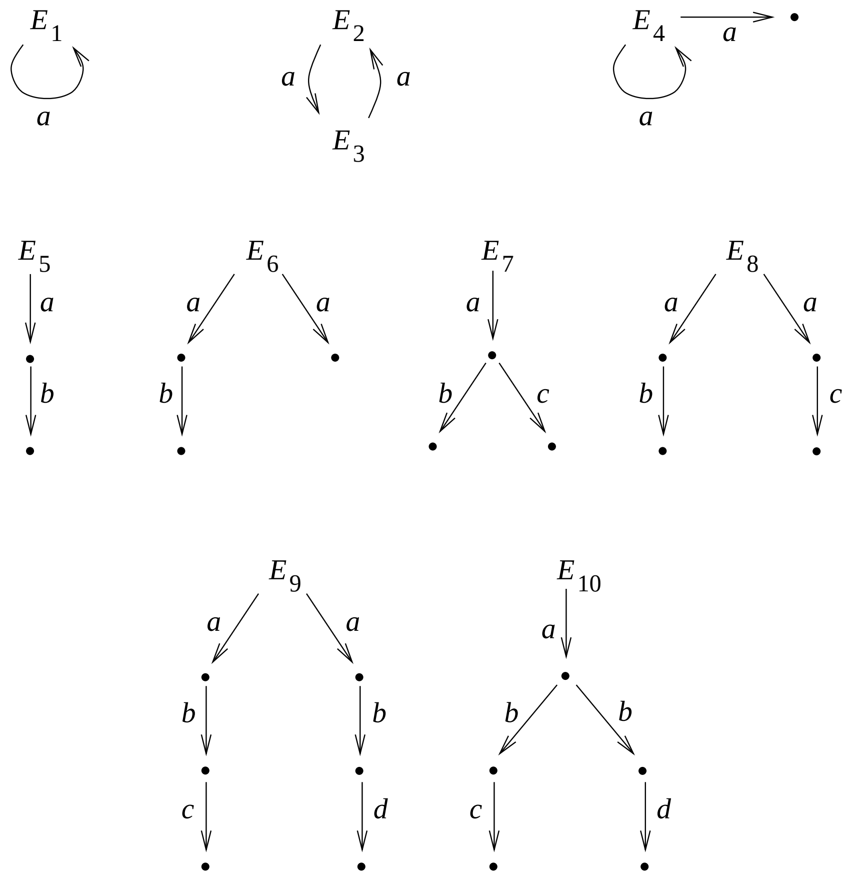


Figure 2–1: Examples of Some Simple Processes.

graph isomorphism as a process equivalence. This equivalence is too fine, however, as it can distinguish processes that differ only in their structure. For example, processes E_1 and E_2 of Figure 2–1 are not isomorphic but clearly have the same behaviour, and should be regarded as equivalent.

Alternatively, language equivalence can be used as a process equivalence, as in classical automata theory. We say that finite string $a_1 \cdots a_n$ of actions is a *trace* of E if there exist $E_1, \ldots, E_n$ such that $E \xrightarrow{a_1} E_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} E_n$. The empty string ε is also a trace of every E. Two processes are *trace equivalent* if they have the same traces. This equivalence is too coarse, as it does not take enough of the observable behaviour of processes into account. For example, it equates processes E_1 and E_4 of Figure 2–1, even though one process can deadlock and the other cannot.

2.2.1 Bisimulation Equivalence

A finer equivalence that does distinguish deadlock-free processes from others is obtained by requiring that equivalent processes be able to match each other's actions, and that the derived processes also be equivalent.

Definition 2.1 *Let $(S, A, \{\xrightarrow{a} \mid a \in A\})$ be an LTS. A binary relation $\mathcal{R}$ on S is a* bisimulation *if $(E, F) \in \mathcal{R}$ implies, for all a in A:*

1. *Whenever $E \xrightarrow{a} E'$, then $F \xrightarrow{a} F'$ for some F' such that $(E', F') \in \mathcal{R}$,*

2. *Whenever $F \xrightarrow{a} F'$, then $E \xrightarrow{a} E'$ for some E' such that $(E', F') \in \mathcal{R}$.*

Definition 2.2 *Processes E and F are* bisimulation equivalent*, written $E \simeq F$, if there exists a bisimulation $\mathcal{R}$ such that $(E, F) \in \mathcal{R}$.*

Thus, two processes can be proved bisimulation equivalent by exhibiting a bisimulation that contains them. For example, take processes E_1 and E_2 of Figure 2–1. Relation $\{(E_1, E_2), (E_1, E_3)\}$ is a bisimulation, so $E_1 \simeq E_2$.

It is easy to see that bisimulation equivalent processes have the same traces, so bisimulation equivalence is at least as strong as trace equivalence. Processes E_1 and E_4 of Figure 2–1, which are not bisimulation equivalent, show that bisimulation equivalence is strictly stronger than trace equivalence. However, deterministic processeses are bisimulation equivalent just if they are trace equivalent. The following definition of deterministic process and proof that determinism is preserved by derivation come from Milner in Section 11.1 of [72].

Definition 2.3 *A process E is* deterministic *if, for all processes F in $Rch(E)$, $F \xrightarrow{a} F'$ and $F \xrightarrow{a} F''$ implies $F' \simeq F''$.*

Proposition 2.1 *Determinism is preserved by derivation.*

Proposition 2.2 *Bisimulation equivalence coincides with trace equivalence for deterministic processes.*

Proof. We show that

$$\mathcal{R} = \{(E, F) \mid E, F \text{ are trace equivalent and deterministic}\}$$

is a bisimulation. Suppose $E \xrightarrow{a} E'$. Then $F \xrightarrow{a} F'$ for some F' since E and F are trace equivalent, and E' and F' are deterministic because determinism is preserved by derivation. We have only to show that E' and F' are trace equivalent. Suppose s is a trace of E'. Then as is a trace of E, and therefore a trace also of F, since E and F are trace equivalent. Then s is a trace of some process F'' such that $F \xrightarrow{a} F''$, and because F is deterministic, $F' \simeq F''$. Therefore F' and F'' also have the same traces, so s is a trace of F'. A symmetrical argument shows that every trace s of F' is also a trace of E'. Therefore E' and F' are trace equivalent.

By a symmetrical argument, every transition of F can be matched by a transition of E such that their derivatives are in $\mathcal{R}$, so $\mathcal{R}$ is a bisimulation. □

Checking for bisimulation equivalence is decidable if the processes are finite-state. The time complexity of the best known algorithm [74] for bisimulation checking is $O(\log(k) * l)$, where k is the number of reachable states in the two processes and l is the number of transitions. Thus the mechanical checking of bisimulation equivalence is practical for many examples.

2.2.2 Bisimulation Equivalence as a Greatest Fixed-Point

A different view on bisimulation equivalence can be gained by regarding it as the greatest fixed-point of a certain function on process relations. From this perspective the bisimulation proof technique is just the application of a simple fixed-point induction rule.

Definition 2.4 *Let $f : A \rightarrow A$ be a function. Then an element x of A is a* fixed-point *of f if $f(x) = x$. Suppose set A is equipped with an order relation. Then an element x of A is the* least fixed-point *of f (written μf) if it is the least element of the set of fixed-points of f, and the* greatest fixed-point *of f (written νf) if it is the greatest element of the set of fixed-points of f.*

The Knaster-Tarski fixed-point theorem [90] gives a set-theoretic characterization of the least and greatest fixed-points of functions on certain domains. We present here a version of the theorem for set-valued, monotonic functions.

Definition 2.5 *Let A be a set ordered by $\leq$ and let x, y be elements of A. Then function $f : A \rightarrow A$ is* monotonic *if $x \leq y \Rightarrow f(x) \leq f(y)$.*

Theorem 2.1 (Knaster-Tarski) *Let A be a set and let $f : 2^A \rightarrow 2^A$ be a monotonic function. Then f has a least fixed-point μf and a greatest fixed-point νf that are characterized as follows:*

$$\begin{aligned} \mu f &= \bigcap\{X \subseteq A \mid f(X) \subseteq X\} \\ \nu f &= \bigcup\{X \subseteq A \mid X \subseteq f(X)\} \end{aligned}$$

An induction rule for greatest fixed-points follows immediately from Knaster-Tarski.

Proposition 2.3 (Greatest fixed-point induction rule) *Let A be a set and let $f : 2^A \rightarrow 2^A$ be a monotonic function. Then for every subset X of A,*

$$X \subseteq f(X) \Rightarrow X \subseteq \nu f.$$

We now return to bisimulation equivalence as a fixed-point. Function $\mathcal{F}$ on process relations is defined as follows: a pair (E, F) of processes is in $\mathcal{F}(R)$ just if, for all a in A,

1. Whenever $E \xrightarrow{a} E'$, then $F \xrightarrow{a} F'$ for some F' such that $(E', F') \in R$,

2. Whenever $F \xrightarrow{a} F'$, then $E \xrightarrow{a} E'$ for some E' such that $(E', F') \in R$.

A relation R is therefore a bisimulation just if $R \subseteq \mathcal{F}(R)$. Furthermore, $\mathcal{F}$ is monotonic: $R \subseteq R'$ implies $\mathcal{F}(R) \subseteq \mathcal{F}(R')$.

By definition, processes E and F are bisimulation equivalent just if they belong to some bisimulation. In other words, bisimulation equivalence is the union of all bisimulations:

$$\simeq \; = \bigcup \{R \subseteq S \times S \mid R \subseteq \mathcal{F}(R)\}.$$

So by the Knaster-Tarski theorem bisimulation equivalence is just the greatest fixed-point of $\mathcal{F}$. Furthermore, the bisimulation proof technique, in which one exhibits a bisimulation to prove two processes equivalent, is a direct application of the greatest fixed-point induction rule.

An example of the usefulness of this view of bisimulation equivalence is seen in the proof of the following proposition.

Proposition 2.4 *$E \simeq F$ just if, for all a in A:*

1. *Whenever $E \xrightarrow{a} E'$, then $F \xrightarrow{a} F'$ for some F' such that $E' \simeq F'$,*

2. *Whenever $F \xrightarrow{a} F'$, then $E \xrightarrow{a} E'$ for some E' such that $E' \simeq F'$.*

Proof. Letting $\mathcal{F}$ be defined as above, the proposition says equivalently that $\simeq \; = \mathcal{F}(\simeq)$, which holds because $\simeq$ is a fixed-point of $\mathcal{F}$. $\square$

2.2.3 Bisimulation up to $\simeq$

Finding and checking a bisimulation relation becomes harder as the relation becomes larger. The bisimulation up to $\simeq$ technique [72] often reduces the size of relation needed to prove bisimulation equivalence. In a bisimulation up to $\simeq$, the derivatives

E' and F' need not be in the relation themselves, but must only be bisimulation equivalent to a pair of processes in the relation.

Definition 2.6 *Let* $(S, A, \{\xrightarrow{a} \mid a \in A\})$ *be an LTS. A binary relation* $\mathcal{R}$ *on* S *is a* bisimulation up to $\simeq$ *if* $(E, F) \in \mathcal{R}$ *implies, for all* a *in* A*:*

1. *Whenever* $E \xrightarrow{a} E'$*, then* $F \xrightarrow{a} F'$ *for some* F' *such that* $(E', F') \in \simeq \circ \mathcal{R} \circ \simeq$*,*

2. *Whenever* $F \xrightarrow{a} F'$*, then* $E \xrightarrow{a} E'$ *for some* E' *such that* $(E', F') \in \simeq \circ \mathcal{R} \circ \simeq$*.*

A bisimulation up to $\simeq$ is not a bisimulation itself, but as with a bisimulation every pair of processes belonging to it are bisimulation equivalent.

Proposition 2.5 *If* $\mathcal{R}$ *is a bisimulation up to* $\simeq$ *then* $\mathcal{R} \subseteq \simeq$*.*

Proof. Suppose $\mathcal{R}$ is a bisimulation up to $\simeq$. We first prove that $\simeq \circ \mathcal{R} \circ \simeq$ is a bisimulation. Suppose $(E, F) \in \simeq \circ \mathcal{R} \circ \simeq$, which we also write as $E \simeq \circ \mathcal{R} \circ \simeq F$. Then by def. of $\circ$ there exist E_1 and F_1 such that

$$E \simeq E_1 \,\mathcal{R}\, F_1 \simeq F.$$

Suppose $E \xrightarrow{a} E'$. Since $E \simeq E_1$ there exists by Prop. 2.4 an E_1' such that $E_1 \xrightarrow{a} E_1'$ and $E' \simeq E_1'$. Since $E_1 \,\mathcal{R}\, F_1$ and $\mathcal{R}$ is a bisimulation there exists an F_1' such that $F_1 \xrightarrow{a} F_1'$ and $E_1' \,\mathcal{R}\, F_1'$. Since $F_1 \simeq F$ there exists an F' such that $F \xrightarrow{a} F'$ and $F_1' \simeq F'$. In summary, there exists an F' such that $F \xrightarrow{a} F'$ and $E' \simeq \circ \mathcal{R} \circ \simeq F'$. By a symmetrical argument we can show that for every transition $F \xrightarrow{a} F'$ there exists an E' such that $E \xrightarrow{a} E'$ and $E' \simeq \circ \mathcal{R} \circ \simeq F'$. Therefore $\simeq \circ \mathcal{R} \circ \simeq$ is a bisimulation.

Let $\equiv$ stand for the identity relation on processes. The relations $\equiv$ and $\simeq \circ \mathcal{R} \circ \simeq$ are bisimulations, so by definition of $\simeq$ we have $\equiv \subseteq \simeq$ and $\simeq \circ \mathcal{R} \circ \simeq \subseteq \simeq$. Therefore

$$\mathcal{R} \subseteq \equiv \circ \mathcal{R} \circ \equiv \subseteq \simeq \circ \mathcal{R} \circ \simeq \subseteq \simeq .$$

and $\mathcal{R} \subseteq \simeq$ by transitivity of $\subseteq$. □

2.2.4 Simulation

Process equivalences can be used to verify that a system satisfies its specification. Here one describes the intended and the actual behaviour of a system as processes, and verifies that the two are equivalent. However, sometimes one does not want to specify exactly what a system should do, but only to put a bound on the behaviour to be allowed of the system. Process preorders are often suited to this style of specification.

A basic process preorder is the simulation preorder, which is closely related to bisimulation equivalence. If F simulates E then every action of E can be matched by F such that F's derivative simulates E's derivative.

Definition 2.7 *Let* $(S, A, \{\xrightarrow{a} \mid a \in A\})$ *be an LTS. A binary relation* $\mathcal{R}$ *on* S *is a* simulation *if* $(E, F) \in \mathcal{R}$ *implies that, for all* a *in* A*:*

Whenever $E \xrightarrow{a} E'$*, then* $F \xrightarrow{a} F'$ *for some* F' *such that* $(E', F') \in \mathcal{R}$*.*

Definition 2.8 *Process* F simulates E*, written* $E \leq F$*, if there exists a simulation* $\mathcal{R}$ *such that* $(E, F) \in \mathcal{R}$*.*

Simulation can be used to describe bisimulation elegantly: a bisimulation is just a symmetric simulation. However, if two processes simulate each other, it does not follow that they are bisimilar. For example, this is the case for processes E_5 and E_6 of Figure 2–1.

Just as bisimulation equivalence is finer than trace equivalence, the simulation preorder is finer than trace inclusion. In fact it is strictly finer, as shown by processes E_7 and E_8 of Figure 2–1. The traces of the first process are also traces of the second, but the second does not simulate the first.

The up to technique used for bisimulation equivalence can also be applied to the simulation preorder. It allows one to prove $E \leq F$ using a smaller relation than would otherwise be necessary.

Definition 2.9 *A binary relation $\mathcal{R}$ on processes is a* simulation up to $\leq$ *if $(E, F) \in \mathcal{R}$ implies, for all a in A:*

Whenever $E \xrightarrow{a} E'$, then $F \xrightarrow{a} F'$ for some F' such that $(E', F') \in \leq \circ \mathcal{R} \circ \leq$.

The proof of the following fact is very similar to the proof of Proposition 2.5.

Proposition 2.6 *If (E, F) belongs to a simulation up to $\leq$, then $E \leq F$.*

2.3 The Process Algebra CCS

While the semantic theory just outlined is useful on its own, an algebraic notation for processes brings additional benefits. First, it allows processes to be written concisely as strings. More importantly, it enables reasoning about processes to be performed by syntactic manipulation. For example, we will see that bisimulation equivalence can be axiomatised as a set of algebraic laws, so that the equivalence of two processes can be proved by applying the laws.

2.3.1 Syntax

CCS is based on a set of actions, each of which is either a *name*, a *co-name*, or the special action τ, which shall be discussed shortly. We let $a, b, \ldots$ range over names and let $\overline{a}, \overline{b}, \ldots$ range over co-names. Names and co-names are collectively called *labels*. We write the set of all labels as $\mathcal{L}$ and let $l, l', \ldots$ range over $\mathcal{L}$. Any label l can be complemented to obtain $\overline{l}$, with complementation satisfying $\overline{\overline{l}} = l$. We let $L, K, \ldots$ range over $\mathcal{L}$ and write $\overline{L}$ to stand for the set of complements of the labels in L. Finally, we write the set $\mathcal{L} \cup \{\tau\}$ of all CCS actions as Act and let $\alpha, \beta, \ldots$ range over Act.

Also basic to CCS are *relabelling functions*, which are functions from $\mathcal{L}$ to $\mathcal{L}$. Every relabelling function f preserves complementation, so $f(\bar{l}) = \overline{f(l)}$, and is extended to Act by $f(\tau) = \tau$. We write $l_1'/l_1, l_2'/l_2, \ldots, l_n'/l_n$ for the relabelling function that maps l_i to l_i', for $1 \le i \le n$, and is the identity elsewhere.

The abstract syntax of CCS processes is defined as follows:

$$E ::= \alpha.E \;\Big|\; \sum_{i \in I} E_i \;\Big|\; E \mid F \;\Big|\; E\backslash L \;\Big|\; E[f] \;\Big|\; A$$

where I is an indexing set, L ranges over sets of labels, f ranges over relabelling functions, and A ranges over process *constants*. We write the set of all processes as $\mathcal{P}$ and let $E, F, \ldots$ range over it. We write $E + F$ for summation over processes E and F, and write 0 for summation over the empty set. We write $E \equiv F$ to mean that processes E and F are syntactically identical.

A *family* of recursive process definitions is a set

$$\{A_i \stackrel{\text{def}}{=} E_i \mid i \in I\}$$

that is *closed*, with each E_i containing only constants A_j $(j \in I)$, and *determinate*, with each A_i defined only once. When we need to specify that process E should be interpreted relative to a family $\mathcal{F}$ of definitions, we write E of $\mathcal{F}$.

The following definitions are an example of a family:

$$\begin{aligned} A &\stackrel{\text{def}}{=} \texttt{coin}.(\overline{\texttt{a}}.A + \tau.\texttt{reject}.A) \\ B &\stackrel{\text{def}}{=} \texttt{a}.\texttt{coffee}.B \\ CM &\stackrel{\text{def}}{=} (A \mid B)\backslash\{a\} \end{aligned}$$

Constant CM describes the behaviour of a coffee machine, while constants A and B describe the two components of the machine. The behaviour of process CM is explained in the following section.

2.3.2 Semantics

The meaning of CCS processes is given as the LTS

$$(\mathcal{P}, Act, \{\xrightarrow{\alpha} \mid \alpha \in Act\})$$

where $\xrightarrow{\alpha} \subseteq \mathcal{P} \times \mathcal{P}$ for all α in Act. The semantics of CCS defines the transition relations of this LTS inductively by a set of rules, one or more for each CCS operator. Each rule has a conclusion and zero or more premises. The transitions of a process are just those that can be deduced by the rules. This style of semantics is called *structured operational semantics* [77].

The transition rules for CCS are as follows:

$$. \;:\quad \alpha.E \xrightarrow{\alpha} E$$

$$\textstyle\sum \;:\quad \frac{E_j \xrightarrow{\alpha} E'_j}{\sum_{i\in I} E_i \xrightarrow{\alpha} E'_j}\; j \in I$$

$$\mid \;:\quad \frac{E \xrightarrow{\alpha} E'}{E \mid F \xrightarrow{\alpha} E' \mid F} \qquad \frac{F \xrightarrow{\alpha} F'}{E \mid F \xrightarrow{\alpha} E \mid F'} \qquad \frac{E \xrightarrow{l} E' \quad F \xrightarrow{\overline{l}} F'}{E \mid F \xrightarrow{\tau} E' \mid F'}$$

$$\backslash \;:\quad \frac{E \xrightarrow{\alpha} E'}{E\backslash L \xrightarrow{\alpha} E'\backslash L}\; \alpha \notin L \cup \overline{L}$$

$$f \;:\quad \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$$

$$\mathrm{def} \;:\quad \frac{E \xrightarrow{\alpha} E'}{A \xrightarrow{\alpha} E'}\; A \stackrel{\mathrm{def}}{=} E$$

Operator '.' is the *prefix* operator. Process $\alpha.E$ can perform α and thereby become process E. Operator $\sum$ is the *summation* operator. Process $\sum_{i\in I} E_i$ can perform an action of any E_i and thereby become the derivative of E_i. Operator $\mid$ is the *composition* operator. Process $E \mid F$ is the concurrent composition of E and F. Process E and F can act independently or synchronise to produce a τ action. Operator $[f]$ is the *relabelling* operator. Process $E[f]$ is the process E with its actions relabelled by f. Relabelling is typically used to "glue" networks of processes together. Operator $\backslash L$ is the *restriction* operator. Process $E\backslash L$ is the process E with its actions restricted to actions not in L or $\overline{L}$. Restriction is typically used to force composed processes to synchronise.

The transition rules of CCS show the special character of the τ action. While actions generally represent system events, the τ action represents a system event that is invisible to an observer of the system. This idea explains why synchronisation of two components in the third rule for $|$ produces a τ action — an observer cannot detect when two components of a system synchronise. It also explains why synchronisation is impossible on τ actions.

We now return to the coffee machine example of the previous section. Component A of CM models the "coin handling sub-system" of the coffee-machine. It accepts a coin and either rejects it or signals its acceptance. The LTS for process A is as shown in Figure 2–2. Component B of CM models the "coffee delivery sub-system" of the coffee-machine. It simply waits for a signal and then delivers the coffee.

The LTS for process CM is shown in Figure 2–3. A slight simplification has been made in the diagram. According to the semantics of processes, the transition labelled `reject` from process $(\texttt{reject}.A \mid B)\backslash\{\texttt{a}\}$ leads to process $(A \mid B)\backslash\{\texttt{a}\}$. However, Figure 2–3 shows the transition leading to process CM. Since CM can perform exactly the same transitions as $(A \mid B)\backslash\{\texttt{a}\}$ the two processes are identified in the diagram for simplicity. This convention is also followed in later transition diagrams.

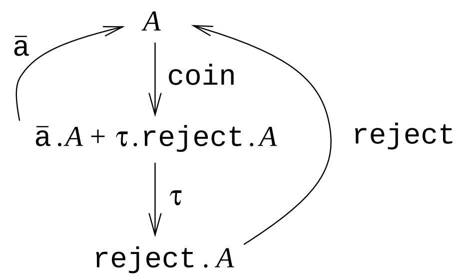


Figure 2–2: Transition Diagram for a Coffee Machine Component

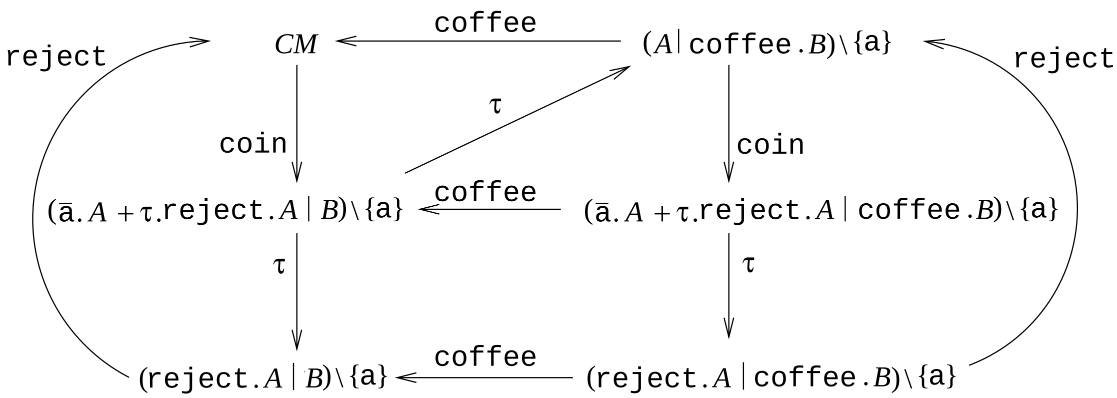


Figure 2–3: Transition Diagram for the Coffee Machine

The operators of CCS can be classified as either *static* or *dynamic*. A static operator is retained across transitions. By examining the transition rules it is easy to see that the composition, restriction, and relabelling operators are static. The other operators, choice and prefix, are dynamic.

The *syntactic sort* of a process E, written $\mathcal{L}(E)$, is the set of all labels that the process might potentially perform. It is defined to be the least set satisfying the following rules:

$$\begin{array}{rcl}
\mathcal{L}(l.E) & = & \{l\} \cup \mathcal{L}(E) \\
\mathcal{L}(\tau.E) & = & \mathcal{L}(E) \\
\mathcal{L}(\sum_{i\in I} E_i) & = & \bigcup_{i\in I} \mathcal{L}(E_i) \\
\mathcal{L}(E_1 \mid E_2) & = & \mathcal{L}(E_1) \cup \mathcal{L}(E_2) \\
\mathcal{L}(E\backslash L) & = & \mathcal{L}(E) - (L \cup \overline{L}) \\
\mathcal{L}(E[f]) & = & \{f(l) \mid l \in \mathcal{L}(E)\} \\
\mathcal{L}(A) & \supseteq & \mathcal{L}(E) \qquad (A \stackrel{\text{def}}{=} E)
\end{array}$$

The clause for $\tau.E$ shows that τ actions are never found in the sort of a process. The clause for A shows that the sort of a constant must be large enough to contain the sort of the body in its definition. The sort $\mathcal{L}(A)$ of process $A \stackrel{\text{def}}{=} a.A$ is therefore $\{a\}$.

The sort $\mathcal{L}(CM)$ of the coffee machine is $\{\texttt{coin}, \texttt{coffee}, \texttt{reject}\}$. Figure 2–4 is a *flow diagram* of the coffee machine, which reflects its static structure. To construct a flow diagram for a process, one starts by drawing a box for each component of the process. Labels in the sort of a component are marked on the edge of its box. Marks of complementary labels are connected by lines. Finally, if a label is restricted, it and its complement are removed from the diagram.

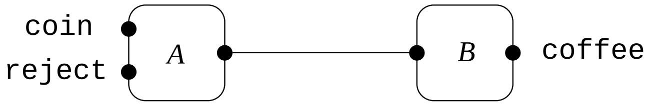


Figure 2–4: Flow Diagram of the Coffee Machine

2.3.3 Strong Equivalence

We have just seen that a CCS process is understood semantically as a state in an LTS. Now we will apply the equivalences and preorders of Section 2.2 to the particular case of CCS. The strong equivalence of CCS is just bisimulation equivalence on the states of the CCS LTS.

Definition 2.10 *Two CCS processes are* strongly equivalent*, written $E \sim F$, if they are bisimulation equivalent relative to the LTS of CCS.*

A process equivalence is a *congruence* if it is preserved by all CCS operators and recursive definition. Strong equivalence is a congruence, so for example $E \sim F$ implies $E \mid G \sim F \mid G$ for all processes E, F, and G.

Theorem 2.2 *Strong equivalence is a congruence.*

Proof. See propositions 4.10 and 4.12 of [72].

The notion of process congruence can be characterized using process contexts. A *context* is a process with a hole, where the hole is written $[\,]$. For example, $a.0 + [\,]$ is a context. If C is a context, we write $C[E]$ for process E put in context C. A process equivalence $\cong$ is then a congruence just if $E \cong F$ implies $C[E] \cong C[F]$ for all contexts C. Thus congruent processes can be used interchangeably, allowing algebraic manipulation of processes. For example, since $E \mid 0$ and E are strongly equivalent, so are $(E \mid 0 \mid G)\backslash L$ and $(E \mid G)\backslash L$. The following are some algebraic laws for strong equivalence.

Proposition 2.7 *For all processes E, F, and G,*

$$
\begin{array}{rcll}
E + F & \sim & F + E & \\
E + (F + G) & \sim & (E + F) + G & \\
E + E & \sim & E & \\
E + 0 & \sim & E & \\
E \mid F & \sim & F \mid E & \\
E \mid (F \mid G) & \sim & (E \mid F) \mid G & \\
E \mid 0 & \sim & E & \\
E & \sim & A & \quad \textit{if } A \stackrel{\text{def}}{=} E
\end{array}
$$

Proof. See Propositions 4.7, 4.8, and 4.14 of [72]. □

Note that both $+$ and $\mid$ are commutative and associative. Consequently we need not worry about parentheses or order in writing the sum or composition of processes.

Special algebraic laws are needed to deal with processes containing constants. For example, consider processes $A \stackrel{\text{def}}{=} a.A$ and $B \stackrel{\text{def}}{=} a.a.B$. Showing these processes to be strongly equivalent can be done by showing that they are both solutions to the equation $X \sim a.a.X$. To support this kind of reasoning one must show that such equations have unique solutions up to strong equivalence.

Here we allow variables like X as processes, and write $E\{F/X\}$ for the substitution of all occurrences of X in E by F. The equation $X \sim E$, in which E contains at most variable X, is guaranteed to have a solution if all occurrences of X in E occur within the scope of an action prefix.

Definition 2.11 *Variable X is* weakly guarded *in E if each occurrence of X is within some subexpression $\alpha.F$ of E.*

For example, X is not weakly guarded on the right-hand side of equation $X \sim X$, for which every process is a solution. Milner shows in [72] that systems of weakly guarded equations have unique solutions up to strong equivalence. We present here a form of the result for a single equation.

Proposition 2.8 *Let E contain at most variable X, and let X be weakly guarded in E. Then for closed processes F and G,*

$$\textit{if } F \sim E\{F/X\} \textit{ and } G \sim E\{G/X\} \textit{ then } F \sim G.$$

Proof. See Proposition 4.14 of [72].

To prove A and B above strongly equivalent we show that $A \sim a.A \sim a.a.A$ and $B \sim a.a.B$. The equation $X \sim a.a.X$ is weakly guarded, and A and B are solutions of it, so $A \sim B$.

2.3.4 Observation Equivalence and Congruence

In Section 2.2 we argued that trace equivalence is too weak to use as a general process equivalence because it does not always distinguish a process that can deadlock from one that cannot. A more comprehensive argument against trace equivalence is based on the notion of observation used in CCS. Here, observation is understood in an active sense as interaction with a process, not in a passive sense as the simple watching of a process. Thus, processes E_7 and E_8 of Figure 2–1 are regarded as having different behaviour because after interacting with E_7 on action a, an observer is able to interact on both b and c. With E_8, after interaction on a it is only possible to interact on either b or c.

Strong equivalence treats τ like any other action, and therefore makes distinctions between processes that an observer could not. For example, an observer cannot distinguish $a.\tau.0$ and $a.0$. However, an observer can distinguish $a.0 + b.0$ and $a.0 + \tau.b.0$. An observer can always interact with the first process on actions a and b (until one is chosen), while the second process can invisibly evolve to $b.0$, in which interaction on a is impossible.

An observation equivalence can be derived from bisimulation equivalence by defining a transition relation that represents observable activity. We first define a transition representing state change that can occur invisibly.

$$E \stackrel{\varepsilon}{\Rightarrow} E' \stackrel{\text{def}}{=} E(\stackrel{\tau}{\rightarrow})^* E'$$

We next define a transition representing state change that can occur as the observation of the label l.

$$E \stackrel{l}{\Rightarrow} E' \stackrel{\text{def}}{=} E \stackrel{\varepsilon}{\Rightarrow} \circ \stackrel{l}{\rightarrow} \circ \stackrel{\varepsilon}{\Rightarrow} E'$$

The *weak CCS LTS* is

$$(\mathcal{P}, Act, \{\stackrel{\alpha}{\Rightarrow} \mid \alpha \in \mathcal{L} \cup \{\varepsilon\}\})$$

Notice that the $\stackrel{\alpha}{\Rightarrow}$ relations are defined over labels and the symbol ε. However, we sometimes write $E \stackrel{\tau}{\Rightarrow} E'$ to mean that E can reach E' via one or more $\stackrel{\tau}{\rightarrow}$ transitions. We refer to the CCS transition relation $\rightarrow$ as the *strong* transition relation and the transition relation $\Rightarrow$ as the *weak* transition relation.

Definition 2.12 *CCS processes E and F are* observation equivalent*, written $E \approx F$, if they are bisimulation equivalent relative to the weak CCS LTS.*

Amongst the equivalences designed to detect only observable differences between processes (see [94] for a classification of many of them), observation equivalence is one of the finest. Unfortunately, observation equivalence is not a congruence, although it is preserved by all process operators except choice. For example,

$$b.0 \approx \tau.b.0$$

but

$$a.0 + b.0 \not\approx a.0 + \tau.b.0.$$

This example illustrates the "pre-emptive power of τ". Process $a.0 + \tau.b.0$ can commit to synchronisation on action b by performing τ, but process $a.0 + b.0$ can never make such a commitment. This distinction is similar to that made in CSP [54] between internal choice (where the process chooses) and external choice (where the environment chooses).

Since observation equivalence is not always preserved by choice, it is not convenient for algebraic reasoning about processes. However, by strengthening it slightly we get a congruence.

Definition 2.13 *Processes E and F are* observation congruent, *written $E = F$, if for all α in Act,*

1. *Whenever $E \xrightarrow{\alpha} E'$, then $F \overset{\alpha}{\Rightarrow} F'$ for some F' such that $E' \approx F'$,*

2. *Whenever $F \xrightarrow{\alpha} F'$, then $E \overset{\alpha}{\Rightarrow} E'$ for some E' such that $E' \approx F'$.*

Theorem 2.3 *Observation congruence is a congruence.*

Proof. See propositions 7.7 and 7.8 of [72].

Observation congruence and observation equivalence differ only in their treatment of the initial transitions of processes. Indeed, only initial τ transitions are significant. Using this fact we can often prove $E = F$ given $E \approx F$.

Definition 2.14 *A process E is* stable *if it has no τ-derivatives.*

Proposition 2.9 *If $E \approx F$, and both are stable, then $E = F$.*

Proof. See [72].

Observation congruence is weaker than strong equivalence, so all the algebraic laws for $\sim$ are also laws for $=$. The following are laws for $=$ but not $\sim$.

Proposition 2.10 *For all processes E,*

$$\begin{aligned} \alpha.\tau.E &= \alpha.E \\ E + \tau.E &= \tau.E \end{aligned}$$

Proof. See Prop. 3.2 of [72].

The first law shows that if a τ cannot pre-empt a choice then it can be removed without affecting the observable behaviour of a process.

To summarise, the main CCS equivalences are strong equivalence ($\sim$), observation equivalence ($\approx$), and observation congruence ($=$). The following inclusions hold between them:

$$\sim\ \subset\ =\ \subset\ \approx$$

These examples show the inclusions are strict:

$$a.\tau.0 \not\sim a.0 \quad \text{but} \quad a.\tau.0 = a.0$$

$$\tau.a.0 \neq a.0 \quad \text{but} \quad \tau.a.0 \approx a.0$$

Equivalences $\sim$ and $=$ are congruences, while $\approx$ is not.

2.3.5 Simulation

We say that one CCS process simulates another if they are related by the simulation preorder generated by the strong CCS LTS.

Definition 2.15 *CCS process F* simulates *process E, written $E < F$, if F simulates E relative to the strong CCS LTS.*

A *precongruence* is a preorder preserved by all CCS operators and recursive definition. The simulation preorder on CCS processes is a precongruence.

Proposition 2.11 *The simulates relation is preserved by all CCS operators.*

Proof. We assume $E < F$ and prove the following cases.

- $\alpha.E < \alpha.F$. Suppose $\alpha.E \xrightarrow{\alpha} E$. We have $\alpha.F \xrightarrow{\alpha} F$ directly, and $E < F$ by assumption.

- $E + \sum_{i \in I} G_i < F + \sum_{i \in I} G_i$. Suppose $E + \sum_{i \in I} G_i \xrightarrow{\alpha} E'$. The transition can be inferred in two ways. First, suppose $E \xrightarrow{\alpha} E'$. Then by assumption $F \xrightarrow{\alpha} F'$ and $E' < F'$. But then $F + \sum_{i \in I} G_i \xrightarrow{\alpha} F'$. Secondly, suppose $G_j \xrightarrow{\alpha} E'$ for some $j \in I$. Then we know $F + \sum_{i \in I} G_i \xrightarrow{\alpha} E'$, and $E' < E'$ as $<$ is reflexive.

- $E \mid G < F \mid G$. We prove that the relation

  $$\mathcal{R} \stackrel{\text{def}}{=} \{(E \mid G, F \mid G) \bigm| E < F\}$$

  is a simulation. Suppose $E \mid G \xrightarrow{\alpha} E' \mid G'$. There are three cases to consider. Suppose $G \xrightarrow{\alpha} G'$ and $E \equiv E'$. Then $F \mid G \xrightarrow{\alpha} F \mid G'$, and $(E \mid G', F \mid G') \in \mathcal{R}$ by definition of $\mathcal{R}$.

  For the second case suppose that $E \xrightarrow{\alpha} E'$ and $G \equiv G'$. Then by assumption $F \xrightarrow{\alpha} F'$ and $E' < F'$, and by definition of $\mid$ we have $F \mid G \xrightarrow{\alpha} F' \mid G$. Furthermore, by assumption and by definition of $\mathcal{R}$ we have $(E' \mid G, F' \mid G) \in \mathcal{R}$.

  Finally, suppose $E \xrightarrow{l} E'$ and $G \xrightarrow{\bar{l}} G'$. Then by assumption $F \xrightarrow{l} F'$ and $E' < F'$, so $F \mid G \xrightarrow{\tau} F' \mid G'$ and $(E' \mid G', F' \mid G') \in \mathcal{R}$.

- $E \backslash L < F \backslash L$. We prove that

  $$\mathcal{R} \stackrel{\text{def}}{=} \{(E \backslash L, F \backslash L) \mid E < F \text{ and } L \subseteq \mathcal{L}\}$$

  is a simulation. Suppose $E \backslash L \xrightarrow{\alpha} E' \backslash L$. Then $E \xrightarrow{\alpha} E'$ and $\alpha \notin L \cup \overline{L}$. By assumption $F \xrightarrow{\alpha} F'$ and $E' < F'$, so $F \backslash L \xrightarrow{\alpha} F' \backslash L$, and $(E' \backslash L, F' \backslash L) \in \mathcal{R}$.

- $E[f] < F[f]$. We prove that

  $$\mathcal{R} \stackrel{\text{def}}{=} \{(E[f], F[f]) \mid E < F \text{ and } f \text{ a relabelling function}\}$$

  is a simulation. Suppose $E[f] \stackrel{\alpha}{\rightarrow} E'[f]$. Then $E \stackrel{\beta}{\rightarrow} E'$ and $f(\beta) = \alpha$. By assumption $F \stackrel{\beta}{\rightarrow} F'$ and $E' < F'$. Then by definition of $[f]$ it follows that $F[f] \stackrel{f(\beta)}{\rightarrow} F'[f]$ or equivalently $F[f] \stackrel{\alpha}{\rightarrow} F'[f]$, and $(E'[f], F'[f]) \in \mathcal{R}$. □

To show that recursive definition preserves simulation we allow variables $X, Y, \ldots$ as processes and define the simulates relation between processes with variables. Suppose processes E and F each contain variable X at most, and let a *closed* process be one without variables. Then $E < F$ if $E\{P/X\} < F\{P/X\}$ for all closed processes P. We say that $<$ preserves recursive definition if $E < F$ implies $A < B$, where A is defined as $E\{A/X\}$ and B is defined as $F\{B/X\}$.

For the case of multiple recursive definitions we consider processes with multiple variables. Suppose E, F are processes containing distinct variables $X_1, \ldots, X_n$, and $G_1, \ldots, G_n$ are closed processes. We write $E\{G_1/X_1, \ldots, G_n/X_n\}$ for E with all X_i simultaneously replaced by G_i. Then $E < F$ if $E\{G_1/X_1, \ldots, G_n/X_n\} < F\{G_1/X_1, \ldots, G_n/X_n\}$ for all closed processes $G_1, \ldots, G_n$.

Proposition 2.12 *Let $E_1, \ldots, E_n$ and $F_1, \ldots, F_n$ be processes containing at most free variables $X_1, \ldots, X_n$. If we define*

$$\begin{aligned} A_i &\stackrel{\text{def}}{=} E_i\{A_1/X_1, \ldots, A_n/X_n\} \\ B_i &\stackrel{\text{def}}{=} F_i\{B_1/X_1, \ldots, B_n/X_n\} \end{aligned}$$

then $E_i < F_i$ $(1 \leq i \leq n)$ implies $A_i < B_i$ $(1 \leq i \leq n)$.

Proof. We show only the case for $n = 1$. Thus we have processes E and F, and define A and B in terms of them.

Suppose $E < F$. We prove that $A < B$ by showing that the relation

$$\mathcal{R} \stackrel{\text{def}}{=} \{(G\{A/X\}, G\{B/X\}) \mid G \text{ contains at most } X \text{ free}\}$$

is a simulation up to $<$. From this we know $A < B$ because (A, B) is an element of $\mathcal{R}$ when G is taken to be just X.

To show that $\mathcal{R}$ is a simulation up to $<$, we prove that, for all α in *Act*,

> Whenever $G\{A/X\} \xrightarrow{\alpha} E'$, then $G\{B/X\} \xrightarrow{\alpha} F'' > F'$ for some F', F'' such that $(E', F') \in \mathcal{R}$.

The proof is by induction on the depth of the inference of transition $G\{A/X\} \xrightarrow{\alpha} E'$. We do case analysis on the structure of G.

Case $G \equiv X$**.** Suppose $X\{A/X\} \equiv A \xrightarrow{\alpha} E'$. Then $E\{A/X\} \xrightarrow{\alpha} E'$ by a shorter inference, and by induction $E\{B/X\} \xrightarrow{\alpha} F'' > F'$ and $(E', F') \in \mathcal{R}$. By the assumption $E < F$ we get $F\{B/X\} \xrightarrow{\alpha} F'''$ and $F'' < F'''$. Then $B \xrightarrow{\alpha} F'''$ by definition of B and $G\{B/X\} \xrightarrow{\alpha} F'''$ since $G \equiv X$. Furthermore $F' < F'''$ by transitivity of $<$.

Case $G \equiv \alpha.G'$**.** Suppose $\alpha.G'\{A/X\} \xrightarrow{\alpha} G'\{A/X\}$. Then $\alpha.G'\{B/X\} \xrightarrow{\alpha} G'\{B/X\}$ and $(G'\{A/X\}, G'\{B/X\}) \in \mathcal{R}$ by the definition of $\mathcal{R}$.

Case $G \equiv \sum_{i\in I} G_i$**.** Suppose $(\sum_{i\in I} G_i)\{A/X\} \equiv \sum_{i\in I}(G_i\{A/X\}) \xrightarrow{\alpha} E'$. The transition can be inferred from a transition $G_j\{A/X\} \xrightarrow{\alpha} E'$ for some $j \in I$. Then by induction $G_j\{B/X\} \xrightarrow{\alpha} F'' > F'$ and $(E', F') \in \mathcal{R}$. We can then infer the transition $\sum_{i\in I}(G_i\{B/X\}) \xrightarrow{\alpha} F''$, so $(\sum_{i\in I} G_i)\{B/X\} \xrightarrow{\alpha} F'' > F'$.

Case $G \equiv G_1 \mid G_2$**.** Suppose $(G_1 \mid G_2)\{A/X\} \xrightarrow{\alpha} E_1' \mid E_2'$. The transition can be inferred in one of three ways. If it is inferred from transition $G_1\{A/X\} \xrightarrow{\alpha} E_1'$ of G_1 then by induction $G_1\{B/X\} \xrightarrow{\alpha} F_1'' > F'$ and $(E_1', F_1') \in \mathcal{R}$. The pair

(E_1', F_1') can be written $(H\{A/X\}, H\{B/X\})$, where H contains at most X free. Now

$$\begin{array}{rlcl} & (G_1 \mid G_2)\{B/X\} & & \\ \equiv & (G_1\{B/X\} \mid G_2\{B/X\}) & \xrightarrow{\alpha} & F_1'' \mid E_2' \\ & & > & F_1' \mid E_2' \\ & & \equiv & H\{B/X\} \mid G_2\{B/X\} \\ & & \equiv & (H \mid G_2)\{B/X\}. \end{array}$$

Furthermore $E_1' \mid E_2' \equiv H\{A/X\} \mid G_2\{A/X\} \equiv (H \mid G_2)\{A/X\}$, and so $((H \mid G_2)\{A/X\}, (H \mid G_2)\{B/X\}) \in \mathcal{R}$. The second case, in which G's transition is inferred from a transition of G_2, is symmetrical.

In the third case G's transition is inferred by transitions $G_1\{A/X\} \xrightarrow{l} E_1'$ and $G_2\{A/X\} \xrightarrow{\overline{l}} E_2'$. Then by induction we know $G_1\{B/X\} \xrightarrow{l} F_1'' > F_1'$, $G_2\{B/X\} \xrightarrow{\overline{l}} F_2'' > F_2'$, and (E_1', F_1') and (E_2', F_2') are in $\mathcal{R}$. These pairs can be written $(H_1\{A/X\}, H_1\{B/X\})$ and $(H_2\{A/X\}, H_2\{B/X\})$, respectively, where H_1, H_2 contain at most X free. Now

$$\begin{array}{rlcl} & (G_1 \mid G_2)\{B/X\} & & \\ \equiv & (G_1\{B/X\} \mid G_2\{B/X\}) & \xrightarrow{\tau} & (F_1'' \mid F_2'') \\ & & > & (F_1' \mid F_2') \\ & & \equiv & (H_1\{B/X\} \mid H_2\{B/X\}) \\ & & \equiv & (H_1 \mid H_2)\{B/X\} \end{array}$$

Furthermore, $(H \mid G_2\{A/X\}, H \mid G_2\{B/X\}) \in \mathcal{R}$.

Case $G \equiv G'\backslash L$**.** Suppose $G'\backslash L\{A/X\} \xrightarrow{\alpha} E'\backslash L$. Then $G'\{A/X\} \xrightarrow{\alpha} E'$ and $\alpha \notin L \cup \overline{L}$. By induction $G'\{B/X\} \xrightarrow{\alpha} F'' > F'$ and $(E', F') \in \mathcal{R}$. We can write (E', F') as $(H\{A/X\}, H\{B/X\})$, where H contains at most X free. Now

$$G'\{B/X\}\backslash L \equiv G'\backslash L\{B/X\} \xrightarrow{\alpha} F''\backslash L > F'\backslash L$$

and $(E'\backslash L, F'\backslash L) \equiv (H\{A/X\}\backslash L, H\{B/X\}\backslash L) \equiv (H\backslash L\{A/X\}, H\backslash L\{B/X\})$. Since $H\backslash L$ contains at most X free, we know $(E'\backslash L, F'\backslash L) \in \mathcal{R}$.

Case $G \equiv G'[f]$. This case is nearly identical to the previous one.

Case $G \equiv A'$. Here A' is a constant defined by $A' \stackrel{\text{def}}{=} E'$, with X not appearing in E'. Since A' is a constant, $A'\{A/X\}$ and $A'\{B/X\}$ are identical so both have E' as an α-derivative. Since $E' \equiv E'\{A/X\} \equiv E'\{B/X\}$, the pair $(E'\{A/X\}, E'\{B/X\})$ is in $\mathcal{R}$. □

Theorem 2.4 *The simulates relation is a precongruence.*

Proof. Immediate from Propositions 2.11 and 2.12. □

2.3.6 Weak Simulation

We say that one CCS process weakly simulates another if they are related by the simulation preorder generated by the weak CCS transition system.

Definition 2.16 *CCS process F* weakly simulates *E, written $E \ll F$, if F simulates E relative to the weak CCS LTS.*

The weakly simulates relation is also a precongruence, which is somewhat surprising because observation equivalence is not a congruence. The latter fails to hold because of the pre-emptive power of τ actions: $\tau.a.0$ and $a.0$ are observation equivalent, but $\tau.a.0 + b.0$ and $a.0 + b.0$ are not. However, $\tau.a.0 + b.0$ is weakly simulated by $a.0 + b.0$, and conversely.

In proving that $\ll$ is a precongruence the following characterization of it is useful. Let $\widehat{\alpha}$ be ε if α is τ and α otherwise.

Proposition 2.13 *A binary relation $\mathcal{R}$ on CCS processes is a* weak simulation *just if $(E, F) \in \mathcal{R}$ implies that, for all α in Act:*

Whenever $E \xrightarrow{\alpha} E'$, then $F \stackrel{\widehat{\alpha}}{\Rightarrow} F'$ for some F' such that $(E', F') \in \mathcal{R}$.

Proposition 2.14 *$E \ll F$ just if there exists a weak simulation relation $\mathcal{R}$ such that $(E,F) \in \mathcal{R}$.*

Proposition 2.15 *The weakly simulates relation is a precongruence.*

Proof. The proof is very similar to the proof that $<$ is a precongruence. We first show that all CCS operators preserve $\ll$. The differences between the proof for $<$ and $\ll$ are:

- Case $|$, third sub-case: From $F \overset{l}{\Rightarrow} F'$ and $G \overset{\bar{l}}{\rightarrow} G'$ we can infer $F \mid G \overset{\varepsilon}{\Rightarrow} F' \mid G'$.
- Case $\backslash L$: From $F \overset{\widehat{\alpha}}{\Rightarrow} F'$ and $\alpha \notin L \cup \overline{L}$ we can infer $F \backslash L \overset{\widehat{\alpha}}{\Rightarrow} F' \backslash L$
- Case $[f]$: From $F \overset{\widehat{\alpha}}{\Rightarrow} F'$ we can infer $F[f] \overset{\widehat{f(\alpha)}}{\Rightarrow} F'[f]$.

We secondly show that recursive definition preserves $\ll$. Here a weak simulation up to $\ll$ is used. The relation is defined analogously to simulation up to $<$. The differences in the second part of the proof occur in the same cases and in the same way as in the first part. □

2.3.7 Discussion

We have seen that bisimulation equivalence is strong enough to detect deadlock, has an associated proof technique that is useful for hand proofs, has an efficient checking algorithm, and allows for the algebraic manipulation of CCS processes. Furthermore, bisimulation equivalence is a natural equivalence on processes in the sense that it can be characterized in several quite different ways: logically [52], game-theoretically [88], and as a testing equivalence [1]. For example, in [52] it is shown that two processes are bisimulation equivalent just if they satisfy the same formulas of a modal logic.

However, bisimulation equivalence has been criticised for making distinctions that are not really observable. Bloom, Istrail, and Meyer [8] assume that an observer can see at most the *completed* traces of a process. A trace $\alpha_1 \cdots \alpha_n$ of process E is completed if there exist $E_1, \ldots, E_n$ such that $E \xrightarrow{\alpha_1} E_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} E_n$ and $E_n \sim 0$. They show that some processes are not strongly equivalent but have the same completed traces in every CCS context. Processes E_9 and E_{10} of Figure 2–1 are given in [8] as an example.

One might suspect that E_9 and E_{10} could be distinguished by their completed traces if CCS were extended with new operators. Bloom *et al* argue in [8] that this is not the case. They define a rule format for defining the structured operational semantics of process operators, called the *GSOS* format, and argue that this format is general enough to allow every reasonable process operator to be defined. The operators of CCS, CSP, and ACP can all be defined in the GSOS format. They show that E_9 and E_{10} have the same completed traces in every context that can be built from operators defined in the GSOS format. If two processes have the same completed traces in every GSOS context we say they are *completed trace congruent*. This congruence has been characterized logically [8] and through a bisimulation-like relation [63].

In [47] Groote and Vaandrager define an alternative rule format called the *tyft/tyxt* format. Tyft/tyxt rules satisfying a condition required for modularity (see Section 7 of [47]) are called *pure*. In pure tyft/tyxt format one can define some operators not allowed in GSOS, but bisimulation is still strictly stronger than completed trace congruence. In [49] Groote defines the *ntyft/ntyxt* format, which is an extension of the tyft/tyxt format that allows rules with negative premises. The pure ntyft/ntyxt format subsumes the GSOS format and pure tyft/tyxt format. Groote shows in [49] that bisimulation and completed trace congruence do coincide for the pure ntyft/ntyxt format.

2.4 Value-passing

Value-passing CCS is an extension to CCS that allows actions and agents to be parametrised by values. It also provides conditional expressions. This extended notation is useful when modelling systems in which components communicate by message passing.

Here we follow Milner [72] and give the semantics of value-passing processes by translation to basic processes. Value-passing processes can also be defined directly through an operational semantics (e.g., [23]). The advantage of a translational definition is that it shows how values can be understood as processes. Furthermore, it allows results about basic processes to be applied directly to value-passing ones. The drawback of the translational approach is that it obscures one's intuition about the behaviour of value-passing processes.

Our definition takes as given a domain *Val* of values, ranged over by $v, v_1, \ldots, v_n$, and a language *Exp* of value expressions, ranged over by $e, e_1, e_2, \ldots$. We assume that *Val* includes the values 'true' and 'false'. We also assume an evaluation function $[\![\;]\!]$, which maps each variable-free expression e in *Exp* to a value $[\![e]\!]$ in *Val*.

2.4.1 Syntax

The abstract syntax of value-passing processes is defined as follows:

$$\begin{aligned} E \quad ::= \quad & \tau.E \;\Big|\; a(x).E \;\Big|\; \overline{a}(e).E \;\Big|\; \sum_{i \in I} E_i \;\Big|\; E \mid F \;\Big|\; E\backslash L \;\Big|\; E[f] \;\Big| \\ & A(e_1, \ldots, e_n) \;\Big|\; \text{if } e \text{ then } E \text{ else } F \end{aligned}$$

Recall that by our convention a ranges over names, L ranges over label sets, f ranges over relabelling functions, and A ranges over process constants. As with basic processes, we write 0 as an abbreviation for summation over the empty set. We also write 'if e then E' as an abbreviation for 'if e then E else 0'.

A value-passing process definition has the form

$$A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} E$$

Families of value-passing definitions are formed just as for basic processes.

An occurrence of a variable x in a value-passing process E is *bound* if there is a subexpression of E of form $a(x).F$; otherwise it is *free*. A value-passing process is *closed* if there are no free occurrences of any variable in it.

2.4.2 Semantics

We write $E\{v/x\}$ for the result of substituting (the syntactic form of) value v for each free occurrence of x in process E. Similarly $E\{v_1/x_1, \ldots, v_n/x_n\}$ stands for the simultaneous substitution of each v_i for every free occurrence of x_i in E. We use parametrised constructions for labels and process constants. Given a label l and a value v, $l(v)$ is a new label. Similarly, given a process constant A and an n-tuple $(v_1, \ldots, v_n)$ of values, $A(v_1, \ldots, v_n)$ is a new process constant. Thus we take $a(1)$ as a legal CCS label. In his translation, Milner translates an a action parametrised with value 1 to the basic action a_1.

A *data domain* D is a mapping from $\mathcal{L}$ to subsets of *Val* such that $D(l) = D(\bar{l})$. Parameters of label l can range over values in the set $D(l)$. For A a constant with n parameters, we write $D(A)$ for the set of n-tuples that A's parameters can range over. We write (E, D) for the translation of a closed process E relative to data domain D. We define (E, D) as follows.

E	(E, D)
$\tau.F$	$\tau.(F, D)$
$a(x).F$	$\sum_{v \in D(a)} a(v).(F\{v/x\}, D)$
$\overline{a}(e).F$	$\overline{a}(\llbracket e \rrbracket).(F, D)$
$\sum_{i \in I} F_i$	$\sum_{i \in I}(F_i, D)$
$F_1 \mid F_2$	$(F_1, D) \mid (F_2, D)$
$F \backslash L$	$(F, D)\backslash\{l(v) \mid v \in D(l) \text{ and } l \in L\}$
$F[f]$	$(F, D)[\hat{f}]$, where $\hat{f}(l(v)) \stackrel{\text{def}}{=} (f(l))(v)$
$A(e_1, \ldots, e_n)$	$A(v_1, \ldots, v_n)$, where $\llbracket e_i \rrbracket = v_i \ (1 \leq i \leq n)$
if b then F_1 else F_2	$\begin{cases} (F_1, D) & \text{if } \llbracket b \rrbracket = \text{true} \\ (F_2, D) & \text{otherwise} \end{cases}$

A definition of form $A(x_1, \ldots, x_n) \stackrel{\text{def}}{=} E$ is translated to the set

$$\{A(v_1, \ldots, v_n) \stackrel{\text{def}}{=} (E\{v_1/x_1, \ldots, v_n/x_n\}, D) \mid (v_1, \ldots, v_n) \in D(A)\}$$

of basic CCS definitions.

Chapter 3

Modal Logics for Processes

In this chapter we describe three modal logics for processes: Hennessy-Milner logic, propositional dynamic logic, and the modal mu-calculus. We focus on the model theory of the logics, especially on results showing how the logics characterize process relations. We also examine the relative expressiveness of the logics.

3.1 Background

Logic has long been used as a tool for verifying the correctness of programs. For sequential programs, one sense of correctness is “if the program input values satisfy the initial conditions, then upon termination the input and output values will be in their intended relationship”. This idea can be formalised in *Hoare logic* [53] as the triple

$$\phi\,\{p\}\,\psi$$

where ϕ is a logical formula that expresses the initial condition, p is the program, and ψ is a logical formula that expresses the intended relationship between input and output values.

For reactive programs, correctness instead concerns the ongoing interaction between a program and its environment. We may ask if a particular program event will ever occur, if it can occur before another event, or if it will always continue to occur. Such properties can be formalised in *modal logics*, which contain the modal operator $[a]$. A program satisfies formula $[a]\phi$ if logical formula ϕ holds after program event a occurs.

Various modal logics are obtained by combining this operator with other logical operators. Hennessy-Milner logic [52] is obtained by adding it to propositional logic. Dynamic logic [79,41,50] is obtained by adding it to propositional or first-order logic, and by letting the modal operator take a program as an argument. The modal mu-calculus [59,86] is obtained by adding it to propositional logic and adding a fixed-point operator.

Modal logic can be applied to sequential programs as well as reactive ones. For example, the Hoare triple above can be written as the dynamic logic formula

$$\phi \Rightarrow [p]\psi.$$

Here the program event of interest is the execution of the entire program p. In our use of modal logic with processes the events of interest are the occurrences of actions.

3.2 Hennessy-Milner Logic

Hennessy-Milner logic (HML) is a propositional modal logic. It has propositional operators for negation and conjunction, plus modal operator $[a]$ (pronounced "box a"), where a is an action. Informally, a process satisfies formula $[a]\phi$ if ϕ must hold after an occurrence of a.

HML formulas have the following abstract syntax[1]:

$$\phi ::= \mathtt{ff} \;\Big|\; \neg\phi \;\Big|\; \phi_1 \wedge \phi_2 \;\Big|\; [a]\phi$$

[1]The presentation of HML in [52] contains a diamond operator instead of a box operator, indexed instead of binary conjunction, and no formula `ff`.

where a ranges over a set A of actions. The HML obtained from action set A is said to have *sort* A.

HML formulas with sort A are interpreted on an LTS

$$\mathcal{T} = (S, A, \{\xrightarrow{a} \mid a \in A\})$$

with action set A. Recall that here S is a set of states, A is a set of transition labels, and each $\xrightarrow{a}$ is a transition relation over S. The semantic function $\|\phi\|^{\mathcal{T}}$ is understood as the set of processes satisfied by formula ϕ.

$$\begin{aligned}
\| \mathtt{ff} \|^{\mathcal{T}} &\stackrel{\text{def}}{=} \emptyset \\
\|\neg\phi\|^{\mathcal{T}} &\stackrel{\text{def}}{=} S - \|\phi\|^{\mathcal{T}} \\
\|\phi_1 \wedge \phi_2\|^{\mathcal{T}} &\stackrel{\text{def}}{=} \|\phi_1\|^{\mathcal{T}} \cap \|\phi_2\|^{\mathcal{T}} \\
\|[a]\phi\|^{\mathcal{T}} &\stackrel{\text{def}}{=} \|[a]\|^{\mathcal{T}}(\|\phi\|^{\mathcal{T}})
\end{aligned}$$

where

$$\|[a]\|^{\mathcal{T}}(\mathcal{E}) \stackrel{\text{def}}{=} \{E \in S \mid \forall E' \in S . E \xrightarrow{a} E' \Rightarrow E' \in \mathcal{E}\}.$$

A process E of an LTS $\mathcal{T}$ *satisfies* formula ϕ, written $E \models^{\mathcal{T}} \phi$, if $E \in \|\phi\|^{\mathcal{T}}$. We write simply $E \models \phi$ when $\mathcal{T}$ is understood.

The first three clauses of $\|\phi\|^{\mathcal{T}}$ are familiar from propositional logic: no process satisfies $\mathtt{ff}$, a process satisfies $\neg\phi$ if it does not satisfy ϕ, and a process satisfies $\phi_1 \wedge \phi_2$ if it satisfies both ϕ_1 and ϕ_2. The fourth clause states that a process satisfies $[a]\phi$ if all its a-derivatives satisfy ϕ.

Additional operators can be defined as derived operators. Formula $\mathtt{tt}$ is defined as $\neg\,\mathtt{ff}$; it is satisfied by all processes. Operators $\vee$, $\rightarrow$, and $\leftrightarrow$ are defined by:

$$\begin{aligned}
\phi_1 \vee \phi_2 &\stackrel{\text{def}}{=} \neg(\neg\phi_1 \wedge \neg\phi_2) \\
\phi_1 \rightarrow \phi_2 &\stackrel{\text{def}}{=} \neg\phi_1 \vee \phi_2 \\
\phi_1 \leftrightarrow \phi_2 &\stackrel{\text{def}}{=} \phi_1 \rightarrow \phi_2 \wedge \phi_2 \rightarrow \phi_1
\end{aligned}$$

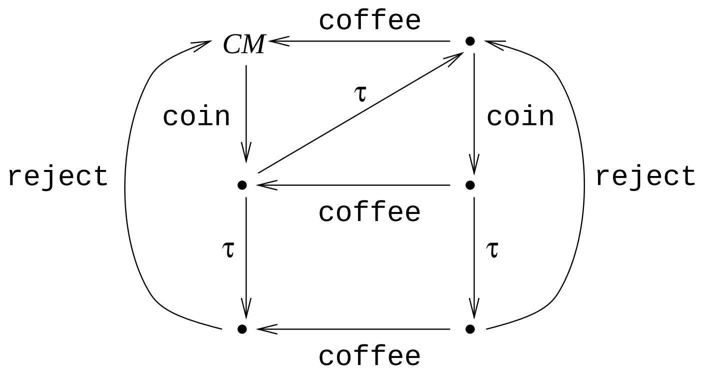


Figure 3–1: Transition Diagram of the Coffee Machine

Modal operator $\langle a \rangle$ (pronounced "diamond a") is defined as the dual of $[a]$:

$$\langle a \rangle \phi \stackrel{\text{def}}{=} \neg [a] \neg \phi$$

A process satisfies $\langle a \rangle \phi$ if some of its a-derivatives satisfies ϕ. Thus $[a]$ and $\langle a \rangle$ effect universal and existential quantification over transitions.

We now present some examples based on the coffee machine of the previous chapter. The LTS of the machine is reproduced in Figure 3–1. The structure of states is omitted for simplicity.

Formula $[a]\,\texttt{ff}$ expresses that no a action can occur. Therefore

$$CM \models [\texttt{coffee}]\,\texttt{ff}\,.$$

Formula $[a]\langle b \rangle\,\texttt{tt}$ expresses that, after every a action, a b action is possible. Therefore

$$CM \models [\texttt{coin}]\langle \tau \rangle\,\texttt{tt}\,.$$

Formula $\langle\texttt{coin}\rangle\langle \tau \rangle\langle\texttt{coin}\rangle\,\texttt{tt}$ expresses that two coins may be accepted before any coffee is delivered. Therefore

$$CM \models \langle\texttt{coin}\rangle\langle \tau \rangle\langle\texttt{coin}\rangle\,\texttt{tt}$$

A key result about HML relates its expressive power to the discriminating ability of bisimulation equivalence. Hennessy and Milner [52] showed that two image-finite processes are equivalent just if they satisfy the same set of HML formulas.

Theorem 3.1 *Let $\mathcal{T} = (S, A, \{\xrightarrow{a} \mid a \in A\})$, and let E and F be image-finite processes in S. Then $E \simeq F$ just if, for all HML formulas ϕ,*

$$E \models^{\mathcal{T}} \phi \Leftrightarrow F \models^{\mathcal{T}} \phi.$$

Proof. See [52].

A useful consequence of this result is that $E \models^{\mathcal{T}} \phi$ can be proved by showing $F \models^{\mathcal{T}} \phi$ for any F bisimilar to E. For example, F might be obtained by simplifying E in a way that preserves bisimilarity. On the other hand, if two processes are not bisimilar, then we know that an HML formula distinguishes them. Image-finiteness is needed only to show that processes satisfying the same HML formulas are bisimilar, not the converse.

Thus HML is expressive enough to characterize every (bisimulation) equivalence class of processes. Yet it is not expressive enough in the sense that no HML formula is satisfied by just the deadlock-free processes. Informally this is because HML formulas can only examine a process to a fixed transition depth n, and for any such n there is a deadlock-free process E and a non-deadlock-free process F that differ only at a depth greater than n. In the next chapter the notion of expressiveness shall be formalised.

3.3 Propositional Dynamic Logic

While HML concerns the occurrence of atomic actions, dynamic logic concerns the execution of structured programs. Thus the modal operator of dynamic logic takes a program, not an action, as its argument. For example, the dynamic logic formula $[s_1; s_2]\phi$ states that ϕ must hold after execution of the program consisting of the sequential composition of statements s_1 and s_2.

Historically, dynamic logic was invented as a modal logic for sequential programs [79]. Formula $[p]\phi$ is interpreted as "after program p terminates, formula ϕ holds".

Proving the correctness of p under pre-condition ψ involves showing the *validity* of the dynamic logic formula $\psi \Rightarrow [p]\phi$. In contrast, we has just seen that verification with HML involves showing the *satisfaction* of a formula by a process. However, it is possible to use the program constructs of dynamic logic simply as a way of adding new modal operators to HML.

There are many dynamic logics, each with its own particular set of logical and program operators. Here we will consider the propositional dynamic logic (PDL) of [41]. PDL formulas have the following abstract syntax[2]:

$$\phi ::= \mathtt{ff} \;\Big|\; \neg\phi \;\Big|\; \phi_1 \wedge \phi_2 \;\Big|\; [p]\phi$$

where p ranges over programs. The abstract syntax of programs is:

$$p ::= a \;\Big|\; p^* \;\Big|\; \phi? \;\Big|\; p_1 ; p_2 \;\Big|\; p_1 \cup p_2$$

where a ranges over a set A of atomic programs and ϕ ranges over formulas. We write Pr for the set of all programs. An important insight of [41] is that the structure of while programs can be captured as regular expressions.

The semantics of PDL formulas and programs is defined by mutual induction. Let $\mathcal{T} = (S, A, \{\xrightarrow{a} \mid a \in A\})$ be an LTS. Here S is understood as a set of program states and $\xrightarrow{a}$ as the state change produced by atomic program a. We define $||\phi||^{\mathcal{T}}$, the set of states satisfying formula ϕ, and $\xrightarrow{p}_D$, the transition relation for program p, as follows.

$$\begin{aligned}
||\,\mathtt{ff}\,||^{\mathcal{T}} &\stackrel{\text{def}}{=} \emptyset \\
||\neg\phi||^{\mathcal{T}} &\stackrel{\text{def}}{=} S - ||\phi||^{\mathcal{T}} \\
||\phi_1 \wedge \phi_2||^{\mathcal{T}} &\stackrel{\text{def}}{=} ||\phi_1||^{\mathcal{T}} \cap ||\phi_2||^{\mathcal{T}} \\
||[p]\phi||^{\mathcal{T}} &\stackrel{\text{def}}{=} ||[p]||^{\mathcal{T}}(||\phi||^{\mathcal{T}})
\end{aligned}$$

[2]We have only $\mathtt{ff}$ as an atomic formula; in [41] general atomic formulas are included. We also omit the blocked program construct of [41].

$$\begin{aligned}
\xrightarrow{a}_D &\stackrel{\text{def}}{=} \xrightarrow{a} \\
\xrightarrow{p^*}_D &\stackrel{\text{def}}{=} (\xrightarrow{p}_D)^* \\
\xrightarrow{\phi?}_D &\stackrel{\text{def}}{=} \{(E,E) \mid E \in \|\phi\|^{\mathcal{T}}\} \\
\xrightarrow{p_1;p_2}_D &\stackrel{\text{def}}{=} \xrightarrow{p_1}_D \circ \xrightarrow{p_2}_D \\
\xrightarrow{p_1 \cup p_2}_D &\stackrel{\text{def}}{=} \xrightarrow{p_1}_D \cup \xrightarrow{p_2}_D
\end{aligned}$$

where

$$\|[p]\|^{\mathcal{T}}(\mathcal{E}) \stackrel{\text{def}}{=} \{E \in S \mid \forall E' \in S. E \xrightarrow{p}_D E' \Rightarrow E' \in \mathcal{E}\}.$$

A process E of $\mathcal{T}$ *satisfies* PDL formula ϕ, written $E \models^{\mathcal{T}} \phi$, if $E \in \|\phi\|^{\mathcal{T}}$. We write simply $E \models \phi$ when $\mathcal{T}$ is understood.

Informally, p^* means p executed zero or more times, $\phi?$ means proceed if ϕ is true, $p_1; p_2$ means do p_1 then p_2, and $p_1 \cup p_2$ means do p_1 or p_2. Some useful derived program expressions are $(\phi?; p)^*; \neg\phi?$, which expresses *while* ϕ *do* p, and $(\phi?; p) \cup (\neg\phi?; q)$, which expresses *if* ϕ *then* p *else* q. Derived formula $\mathtt{tt}$ and operators $\vee$, $\rightarrow$, $\leftrightarrow$, and $\langle p \rangle$ are defined as for HML.

We now express some properties of the coffee machine example as PDL formulas. A coin does not necessarily lead to coffee, as shown by

$$CM \models \langle \mathtt{coin}; \tau \rangle [\mathtt{coffee}]\, \mathtt{ff}\,.$$

In fact, after accepting a coin the machine may not deliver coffee after any amount of internal activity.

$$CM \models \langle \mathtt{coin}; \tau \rangle [\tau^*; \mathtt{coffee}]\, \mathtt{ff}\,.$$

A coffee will never be delivered before a coin is accepted:

$$CM \models [(\mathtt{reject} \cup \tau)^*][\mathtt{coffee}]\, \mathtt{ff}\,.$$

The machine is ready to accept a coin after any number of coffee deliveries:

$$CM \models [(\mathtt{coin}; \tau; \mathtt{coffee})^*]\langle \mathtt{coin} \rangle\, \mathtt{tt}\,.$$

PDL and HML can be compared in terms of their relative expressiveness. Suppose ML_1 and ML_2 are two logics interpreted over the same set of models. A formula ϕ_1 of ML_1 *implies* a formula ϕ_2 of ML_2, written $\phi_1 \Rightarrow \phi_2$, if for every model $\mathcal{M}$ and every process E, E satisfies ϕ_2 with respect to $\mathcal{M}$ whenever E satisfies ϕ_1 with respect to $\mathcal{M}$. Formulas ϕ_1 and ϕ_2 are *equivalent*, written $\phi_1 \Leftrightarrow \phi_2$, if each implies the other. ML_1 is *as expressive as* ML_2 if, for every formula ϕ_2 of ML_2 there exists an equivalent formula ϕ_1 of ML_1. ML_1 is *more expressive* than ML_2 if ML_1 is as expressive as ML_2 but not vice versa.

To show that a logic ML_1 is more expressive than a logic ML_2, one typically shows that ML_1 is as expressive as ML_2, but that there exists a formula of ML_1 that is not equivalent to any formula of ML_2. In the case of PDL and HML, the PDL formula $[a^*]\langle a\rangle\,\mathtt{tt}$ is not equivalent to any HML formula.

Proposition 3.1 *PDL is more expressive than HML.*

Proof. Every HML formula is clearly equivalent to the syntactically identical PDL formula. We now show that no HML formula is equivalent to the PDL formula $[a^*]\langle a\rangle\,\mathtt{tt}$. Let the depth $D(\phi)$ of an HML formula ϕ be defined inductively by $D(\mathtt{ff}) = 0$; $D(\neg\phi) = D(\phi)$; $D(\phi_1 \wedge \phi_2) = \max(\phi_1, \phi_2)$; and $D([a]\phi) = D(\phi) + 1$. Also let the k-depth process E^k of a process E and a natural number k be defined by the semantic rule

$$\frac{E \xrightarrow{\alpha} F}{E^k \xrightarrow{\alpha} F^{k-1}}\; k > 0$$

No path from E^k is longer than k. It is easy to show by induction on the structure of ϕ that $E \models \phi \Leftrightarrow E^k \models \phi$ for every HML formula ϕ and every $k \geq D(\phi)$.

Suppose there is an HML formula ϕ equivalent to PDL formula $[a^*]\langle a\rangle\,\mathtt{tt}$. Since $P \stackrel{\mathrm{def}}{=} a.P$ satisfies $[a^*]\langle a\rangle\,\mathtt{tt}$, it also satisfies ϕ, and therefore $P^{D(\phi)}$ satisfies ϕ. But $P^{D(\phi)}$ does not satisfy $[a^*]\langle a\rangle\,\mathtt{tt}$, so $[a^*]\langle a\rangle\,\mathtt{tt}$ and ϕ are not equivalent, contrary to assumption. $\square$

3.4 The Modal Mu-Calculus

An alternative way to increase the expressive power of HML is to add a *fixed-point* operator, allowing properties to be defined using recursion. For example, deadlock-freedom can be expressed informally using recursion as: "some transition is possible, and no matter which transition is taken, deadlock-freedom holds". For a process with only action a in its sort, this property can be written as $\nu X.\langle a\rangle\,\mathtt{tt} \wedge [a]X$. The operator ν is the new fixed-point operator; the symbol X is a propositional variable. Adding fixed-points to HML results in a logic that has few operators yet is very expressive.

The *propositional μ-calculus* of Kozen [59] contains propositional variables and constants, the propositional connectives, modal operator $\langle a\rangle$, and the least fixed-point operator μ. Kozen's main motivation was to show that this logic is more expressive than PDL. We call the closely-related logic used here the *modal mu-calculus*, which we write as μL. The important difference between μL and Kozen's logic is that the modal operator of μL takes a set of actions, not a single action. This extension was introduced in [86].

Formulas of μL have the following abstract syntax:

$$\phi ::= X \;\Big|\; \neg\phi \;\Big|\; \phi_1 \wedge \phi_2 \;\Big|\; [L]\phi \;\Big|\; \nu X.\phi$$

where X ranges over variables and L ranges over subsets of a set A of actions. The operator ν is a binder; free occurrences of X in ϕ are bound in $\nu X.\phi$. Formulas of μL are required to satisfy the syntactic condition that, in a formula $\nu X.\phi$, variable X may occur freely in ϕ only within an even number of negations.

Formulas of μL are interpreted over *modal models* of the form $\mathcal{M} = (\mathcal{T}, \mathcal{V})$ consisting of an LTS $\mathcal{T} = (S, A, \{\xrightarrow{a} \mid a \in A\})$ and a *valuation* $\mathcal{V}$, which maps propositional variables to sets $\mathcal{E} \subseteq S$ of processes. We write $()$ for the valuation that maps all variables to $\emptyset$ and $\mathcal{V}[X := \mathcal{E}]$ for the valuation that is like $\mathcal{V}$ except that variable X is mapped to the set $\mathcal{E}$. We define $\|\phi\|^{\mathcal{T}}_{\mathcal{V}}$, the set of processes satisfied by ϕ under modal model $\mathcal{M}$, as follows (recall from Def. 2.4 that νf is the greatest fixed-point of function f).

$$
\begin{array}{rcl}
\|X\|^{\mathcal{T}}_{\mathcal{V}} & \stackrel{\text{def}}{=} & \mathcal{V}(X) \\
\|\neg\phi\|^{\mathcal{T}}_{\mathcal{V}} & \stackrel{\text{def}}{=} & S - \|\phi\|^{\mathcal{T}}_{\mathcal{V}} \\
\|\phi_1 \wedge \phi_2\|^{\mathcal{T}}_{\mathcal{V}} & \stackrel{\text{def}}{=} & \|\phi_1\|^{\mathcal{T}}_{\mathcal{V}} \cap \|\phi_2\|^{\mathcal{T}}_{\mathcal{V}} \\
\|[L]\phi\|^{\mathcal{T}}_{\mathcal{V}} & \stackrel{\text{def}}{=} & \|[L]\|^{\mathcal{T}}(\|\phi\|^{\mathcal{T}}_{\mathcal{V}}) \\
\|\nu X.\phi\|^{\mathcal{T}}_{\mathcal{V}} & \stackrel{\text{def}}{=} & \nu f
\end{array}
$$

where

$$
\begin{array}{rcl}
\|[L]\|^{\mathcal{T}}(\mathcal{E}) & \stackrel{\text{def}}{=} & \{E \in S \mid \forall E' \in S, a \in L. E \xrightarrow{a} E' \Rightarrow E' \in \mathcal{E}\} \\
f(\mathcal{E}) & \stackrel{\text{def}}{=} & \|\phi\|^{\mathcal{T}}_{\mathcal{V}[X:=\mathcal{E}]}
\end{array}
$$

A μL formula is *closed* if all its variables are bound. Properties of processes are expressed using only closed formulas. A closed μL formula ϕ *satisfies* E, written $E \models^{\mathcal{T}} \phi$, if $E \in \|\phi\|^{\mathcal{T}}_{()}$. We often write simply $E \models \phi$ when $\mathcal{T}$ is understood.

The syntactic condition on the occurrence of negation symbols within fixed-point formulas ensures the monotonicity of $\|\phi\|^{\mathcal{T}}_{\mathcal{V}}$ with respect to valuations. Suppose that $\mathcal{V}_1$ and $\mathcal{V}_2$ are valuations ranging over set S, and that the condition $\mathcal{V}_1(X) \subseteq \mathcal{V}_2(X)$ holds for all variables X. Then, if ϕ is a μL formula with all free variables occurring within an even number of negations,

$$\|\phi\|^{\mathcal{T}}_{\mathcal{V}_1} \subseteq \|\phi\|^{\mathcal{T}}_{\mathcal{V}_2}.$$

Thus the function f defined above is set-valued and monotonic, so by the Knaster-Tarski theorem (Theorem 2.1) we know that the greatest fixed-point νf of f exists and is equal to

$$\bigcup\{\mathcal{E} \subseteq S \mid \mathcal{E} \subseteq f(\mathcal{E})\}.$$

Derived operators $\vee$, $\rightarrow$, and $\leftrightarrow$ and modal operator $\langle a\rangle$ are defined as in HML. Let $\phi[\psi/X]$ be the formula obtained by the syntactic substitution of ψ for all free occurrences of variable X in ϕ. The least fixed-point formula $\mu X.\phi$ is defined as the dual of $\nu X.\phi$:

$$\mu X.\phi \stackrel{\text{def}}{=} \neg\nu X.\neg\phi[\neg X/X].$$

Derived formulas `tt` and `ff` are defined as $\nu X.X$ and $\mu X.X$, respectively. The following abbreviations are used with $[L]$:

$$\begin{aligned}
[-L]\phi &\stackrel{\text{def}}{=} [A-L]\phi \\
[-]\phi &\stackrel{\text{def}}{=} [A]\phi \\
[a_1,\ldots,a_n]\phi &\stackrel{\text{def}}{=} [\{a_1,\ldots,a_n\}]\phi \\
[-a_1,\ldots,a_n]\phi &\stackrel{\text{def}}{=} [-\{a_1,\ldots,a_n\}]\phi
\end{aligned}$$

The use of sets of actions in modal operators and these abbreviations make it easy to express common properties of processes in a general way. For example, the property of deadlock-freedom can be expressed as formula $\nu X.\langle -\rangle\,\mathtt{tt} \wedge [-]X$.

3.4.1 Reading Fixed-point Formulas

The fixed-point operator makes μL very expressive, but does not have a direct temporal reading like the "always", "eventually", and "until" operators found in some temporal logics. One way to understand fixed-point formulas is through *unfolding*. By definition, $f(\mathcal{E}) = \mathcal{E}$ for any fixed point $\mathcal{E}$ of f. Syntactically this means that $\nu X.\phi$ is equivalent to $\phi[\nu X.\phi/X]$, which is its *unfolding*. For example, if we let DF stand for the μL formula $\nu X.\langle -\rangle\,\mathtt{tt} \wedge [-]X$ that expresses deadlock-freedom, then we have

$$\begin{aligned}
DF &\Leftrightarrow \langle -\rangle\,\mathtt{tt} \wedge [-]DF \\
&\Leftrightarrow \langle -\rangle\,\mathtt{tt} \wedge [-](\langle -\rangle\,\mathtt{tt} \wedge [-]DF) \\
&\Leftrightarrow \ldots
\end{aligned}$$

The same idea works for least fixed-point formulas as long as the formula is unfolded only finitely-many times. For example, formula $\mu X.[-a]X$, which expresses that no infinite sequence of non-a actions is possible, is equivalent to $[-a]\mu X.[-a]X$.

A second way to understand fixed-point formulas is via *approximants*. If $\mathcal{T}$ is an LTS with finitely-many states, then

$$E \models^{\mathcal{T}} \nu X.\phi \text{ iff } E \models^{\mathcal{T}} \bigwedge_{i \in I\!N} X^i$$

where X^i, the ith approximant of $\nu X.\phi$, is defined inductively by

$$\begin{aligned} X^0 &\stackrel{\text{def}}{=} \mathtt{tt} \\ X^{i+1} &\stackrel{\text{def}}{=} \phi[X^i/X] \end{aligned}$$

For example, DF is equivalent to

$$\begin{array}{ll} \mathtt{tt} & \wedge \\ (\langle - \rangle\, \mathtt{tt} \wedge [-]\, \mathtt{tt}) & \wedge \\ (\langle - \rangle\, \mathtt{tt} \wedge [-](\langle - \rangle\, \mathtt{tt} \wedge [-]\, \mathtt{tt})) & \wedge \quad \ldots . \end{array}$$

For the least fixed-point case the definition of X^i changes only for $X^0 \stackrel{\text{def}}{=} \mathtt{ff}$, and then $E \models^{\mathcal{T}} \mu X.\phi$ iff $E \models^{\mathcal{T}} \bigvee_{i \in I\!N} X^i$. For processes with infinite state spaces, approximants must be considered over all ordinals [87].

The Knaster-Tarski theorem (Theorem 2.1) gives a third way of understanding fixed-point formulas. Letting $f(\mathcal{E})$ be $\|\phi\|^{\mathcal{T}}_{\mathcal{V}[X:=\mathcal{E}]}$ we have, by Knaster-Tarski, that

$$\|\nu X.\phi\|^{\mathcal{T}}_{\mathcal{V}} = \nu f = \bigcup \{\mathcal{E} \subseteq S \mid \mathcal{E} \subseteq f(\mathcal{E})\}$$

Therefore we know, by the greatest fixed-point induction rule, that a process satisfies $\nu X.\phi$ if it belongs to a set $\mathcal{E} \subseteq S$ such that $\mathcal{E} \subseteq f(\mathcal{E})$. In the case of deadlock-freedom, $f(\mathcal{E})$ contains those processes that can perform an action and that reach a process in $\mathcal{E}$ by every transition. Therefore $\mathcal{E} \subseteq f(\mathcal{E})$ holds if all processes of $\mathcal{E}$ can perform an action and reach a process in $\mathcal{E}$ by every transition. So the processes in $\mathcal{E}$ are clearly deadlock-free.

Skill in reading μL formulas can also be gained by studying formulas commonly used to express properties of processes, such as:

$$\begin{aligned}
\mathit{always}(\phi) &\stackrel{\text{def}}{=} \nu X.\phi \wedge [-]X \\
\mathit{eventually}(\phi) &\stackrel{\text{def}}{=} \mu X.\phi \vee ([-]X \wedge \langle - \rangle\, \mathtt{tt}) \\
\mathit{until}(\phi_1, \phi_2) &\stackrel{\text{def}}{=} \nu X.\neg\phi_2 \rightarrow (\phi_1 \wedge [-]X)
\end{aligned}$$

Many properties of the coffee machine can be expressed in μL. The coffee machine is deadlock-free:

$$\mathit{CM} \models \nu X.\langle - \rangle\, \mathtt{tt} \wedge [-]X$$

Using the definition of *always* just given, deadlock freedom can equivalently be expressed as $\mathit{always}(\langle - \rangle\, \mathtt{tt})$.

The coffee machine may happen to deliver a coffee for every coin that is accepted:

$$\mathit{CM} \models \nu X.\langle \mathtt{coin} \rangle \langle \tau \rangle \langle \mathtt{coffee} \rangle X$$

Approximants can help in understanding the meaning of this formula. Since process CM has finitely-many states, it satisfies the formula just if it satisfies $X^0 \wedge X^1 \wedge X^2 \wedge \cdots$. Approximant X^0 is $\mathtt{tt}$, approximant X^1 is $\langle \mathtt{coin} \rangle \langle \tau \rangle \langle \mathtt{coffee} \rangle\, \mathtt{tt}$, and approximant X^2 is $\langle \mathtt{coin} \rangle \langle \tau \rangle \langle \mathtt{coffee} \rangle\, \langle \mathtt{coin} \rangle \langle \tau \rangle \langle \mathtt{coffee} \rangle\, \mathtt{tt}$. The ith approximant expresses that the coffee machine must be able to perform the sequence $(\mathtt{coin}\, \tau\, \mathtt{coffee})$ of actions i times.

The coffee machine may also reject coins forever:

$$\mathit{CM} \models \nu X.\langle \mathtt{coin} \rangle \langle \tau \rangle \langle \mathtt{reject} \rangle X$$

and it will not deliver coffee without first taking a coin:

$$\mathit{CM} \models \nu X.[\mathtt{coffee}]\, \mathtt{ff} \wedge [-\, \mathtt{coin}]X.$$

Using approximants we see that CM satisfies this formula just if it satisfies $X^0 \wedge X^1 \wedge X^2 \wedge \cdots$, where X^0 is $\mathtt{tt}$, X^1 is $[\mathtt{coffee}]\, \mathtt{ff} \wedge [-\, \mathtt{coin}]\, \mathtt{tt}$, or equivalently

$[\texttt{coffee}]\,\texttt{ff}$, and X^2 is $[\texttt{coffee}]\,\texttt{ff} \ \wedge\ [-\,\texttt{coin}][\texttt{coffee}]\,\texttt{ff}$. The ith approximant expresses that the coffee machine cannot deliver coffee after less than i non-$\texttt{coin}$ actions occur.

The property just presented only says something about what happens before the first cup of coffee is delivered. A fact about the machine's ongoing behaviour is that if coffee is continually delivered then coins are continually accepted:

$$CM \models \nu X.\mu Y.\nu Z.[\texttt{coin}]X \ \wedge\ [\texttt{coffee}]Y \ \wedge\ [-\,\texttt{coin},\texttt{coffee}]Z.$$

This formula contains alternating fixed-points and is more difficult to read than the previous examples. However, approximants are still useful in understanding it. Process CM satisfies the formula just if it satisfies $X^0 \ \wedge\ X^1 \ \wedge\ X^2 \ \wedge\ \cdots$, where X^0 is $\texttt{tt}$ and X^1 is $\mu Y.\nu Z.[\texttt{coin}]\,\texttt{tt} \wedge [\texttt{coffee}]Y \ \wedge\ [-\,\texttt{coin},\texttt{coffee}]Z$, which is equivalently $\mu Y.\nu Z.[\texttt{coffee}]Y \ \wedge\ [-\,\texttt{coin}]Z$. This least-fixed point formula can in turn be written $Y^0 \ \vee\ Y^1 \ \vee\ Y^2 \ \vee\ \cdots$, where Y^0 is $\texttt{ff}$, Y^1 is $\nu Z.[\texttt{coffee}]\,\texttt{ff} \wedge [-\,\texttt{coin}]Z$, and Y^2 is $\nu Z.[\texttt{coffee}]Y^1 \ \wedge\ [-\,\texttt{coin}]Z$. For $i > 0$, approximant Y^i expresses that at most $i-1$ coffees can be delivered before a coin is accepted. The disjunction of the Y^i's, or equivalently X^1, therefore expresses that at most finitely many coffees can be delivered before a coin is accepted. Later X^i's strengthen the property by further requiring that, after any number of coins has been accepted, coffee cannot continue to be delivered without coins being accepted.

3.4.2 Expressiveness

The modal mu-calculus is more expressive than PDL. A translation function from PDL formulas to equivalent μL formulas shows that μL is as expressive as PDL. Such a function was given by Pratt in [78] for a logic similar to μL, and by Bradfield in [12] for μL. By the translation in [12], the PDL formula $[(a;b)^*]\langle c\rangle\,\texttt{tt}$ translates to the μL formula $\nu X.\langle c\rangle\,\texttt{tt} \wedge [a][b]X$. Kozen used the property that all a paths have finite length, expressible in μL as formula $\mu X.[a]X$, in showing that μL is more expressive than PDL.

Proposition 3.2 *μL is more expressive than PDL.*

Proof. See [59].

Like HML, μL also characterizes strong equivalence.

Theorem 3.2 (μL Characterisation Theorem) *Let $\mathcal{T} = (S, A, \{\xrightarrow{a} \mid a \in A\})$, and let E and F be image-finite processes in A. Then $E \simeq F$ just if, for all μL formula ϕ,*

$$E \models^{\mathcal{T}} \phi \Leftrightarrow F \models^{\mathcal{T}} \phi.$$

Proof. See [88].

3.4.3 The Positive Mu-Calculus

A variant of μL is its positive form, which lacks the negation operator but contains duals of all the other operators. The formulas of this logic, which we call μL^{+}, have the following abstract syntax:

$$\phi ::= X \;\Big|\; \phi_1 \wedge \phi_2 \;\Big|\; \phi_1 \vee \phi_2 \;\Big|\; [L]\phi \;\Big|\; \langle L\rangle\phi \;\Big|\; \nu X.\phi \;\Big|\; \mu X.\phi$$

where, as before, X ranges over propositional variables and L ranges over sets of actions. Derived formulas $\texttt{tt}$ and $\texttt{ff}$ are defined as $\nu X.X$ and $\mu X.X$ respectively.

All closed μL formulas satisfying the syntactic requirement on fixed-point formulas can be written as equivalent μL^{+} formulas by moving negation symbols inwards according to the definitions of the dual operators. For example, μL formula $\neg\nu X.[a]\,\texttt{ff} \wedge [-]X$ can be converted in this way to μL^{+} formula $\mu X.\langle a\rangle\,\texttt{tt} \vee \langle -\rangle X$.

Although μL^{+} has more operators than μL, it has the advantage that the interpretation of every formula is monotonic with respect to valuations even without syntactic restrictions. Thus, monotonicity in μL^{+} is preserved by taking subformulas, making μL^{+} convenient in proofs that use induction on the structure of formulas.

The generated submodel property of modal logic (see for example [46] or [93]) states that the satisfaction of a formula by a process E depends only on the processes reachable from E. We now show this property of μL^+.

Definition 3.1 (Submodel) *Let $(\mathcal{T}, \mathcal{V})$ be a modal model, with $\mathcal{T} = (S, A, \{\xrightarrow{a} \mid a \in A\})$, and let $\mathcal{E}$ be a subset of S. Then*

$$\begin{aligned} \mathcal{T} \cap \mathcal{E} &\stackrel{\text{def}}{=} (\mathcal{E}, A, \{\xrightarrow{a} \cap (\mathcal{E} \times \mathcal{E}) \mid a \in A\}) \\ (\mathcal{V} \cap \mathcal{E})(X) &\stackrel{\text{def}}{=} \mathcal{V}(X) \cap \mathcal{E} \end{aligned}$$

Lemma 3.1 (μL^+ Submodel Lemma) *Let $(\mathcal{T}, \mathcal{V})$ be a modal model, with $\mathcal{T} = (S, A, \{\xrightarrow{a} \mid a \in A\})$, let $\mathcal{E}$ be a transition-closed subset of S, and let ϕ be a μL^+ formula. Then*

$$\|\phi\|^{\mathcal{T}}_{\mathcal{V}} \cap \mathcal{E} = \|\phi\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}}$$

Proof. First we show $\|\phi\|^{\mathcal{T}}_{\mathcal{V}} \cap \mathcal{E} \supseteq \|\phi\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}}$. We have $\|\phi\|^{\mathcal{T}}_{\mathcal{V}} \cap \mathcal{E} \supseteq \|\phi\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}} \cap \mathcal{E} = \|\phi\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}}$ by the monotonicity of μL^+.

Now we show $\|\phi\|^{\mathcal{T}}_{\mathcal{V}} \cap \mathcal{E} \subseteq \|\phi\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}}$ by induction on the structure of ϕ.

- Case $\phi \equiv X$. $\|X\|^{\mathcal{T}}_{\mathcal{V}} \cap \mathcal{E} = \mathcal{V}(X) \cap \mathcal{E} = (\mathcal{V} \cap \mathcal{E})(X) = \|X\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}}$.

- Case $\phi \equiv \phi_1 \wedge \phi_2$. We have $\|\phi_1 \wedge \phi_2\|^{\mathcal{T}}_{\mathcal{V}} \cap \mathcal{E} = \|\phi_1\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}} \cap \|\phi_2\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}}$ by the definition of $\wedge$ and the induction hypothesis. The latter set equals $\|\phi_1 \wedge \phi_2\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}}$ by the definition of $\wedge$.

- Case $\phi \equiv \phi_1 \wedge \phi_2$. Similar reasoning to the previous case.

- Case $\phi \equiv [L]\psi$. Suppose $E \in \|[L]\psi\|^{\mathcal{T}}_{\mathcal{V}} \cap \mathcal{E}$ and $E \xrightarrow{a} E'$ and $a \in L$. Then $E' \in \mathcal{E}$ since $\mathcal{E}$ is transition-closed, and $E' \in \|\psi\|^{\mathcal{T}}_{\mathcal{V}}$, so $E' \in \|\psi\|^{\mathcal{T}}_{\mathcal{V}} \cap \mathcal{E}$. By induction $E' \in \|\psi\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}}$ so $E \in \|[L]\psi\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}}$.

- Case $\phi \equiv \langle L\rangle\psi$. Similar reasoning to the previous case.

- Case $\phi \equiv \nu X.\psi$. Let $\mathcal{F}$ be $\|\nu X.\psi\|^{\mathcal{T}}_{\mathcal{V}}$. Then

$$\begin{aligned}\mathcal{F}\cap\mathcal{E} &= \|\psi\|^{\mathcal{T}}_{\mathcal{V}[X:=\mathcal{F}]}\cap\mathcal{E} && \text{(since } \mathcal{F} \text{ is a fixed-point)}\\ &= \|\psi\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}[X:=\mathcal{F}\cap\mathcal{E}]} && \text{(by the induction hypothesis).}\end{aligned}$$

Therefore $\mathcal{F}\cap\mathcal{E}$ is a fixed-point of $f(\mathcal{F}') \stackrel{\text{def}}{=} \|\psi\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}[X:=\mathcal{F}']}$, so

$$\|\nu X.\psi\|^{\mathcal{T}}_{\mathcal{V}}\cap\mathcal{E} = \mathcal{F}\cap\mathcal{E} \subseteq \|\nu X.\psi\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}}.$$

- Case $\phi \equiv \mu X.\psi$. Let $\mathcal{F}$ be $\|\mu X.\psi\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}}$. Then

$$\begin{aligned}\mathcal{F} &\supseteq \|\psi\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}[X:=\mathcal{F}]} && \text{(since } \mathcal{F} \text{ is a fixed-point)}\\ &= \|\psi\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}[X:=\mathcal{F}]\cap\mathcal{E}} && \text{(since } \mathcal{F}\subseteq\mathcal{E})\\ &= \|\psi\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}[X:=(S-\mathcal{E})\cup\mathcal{F}]\cap\mathcal{E}} && \text{(since } ((S-\mathcal{E})\cup\mathcal{F})\cap\mathcal{E} = \mathcal{F}\cap\mathcal{E})\\ &= \|\psi\|^{\mathcal{T}}_{\mathcal{V}[X:=(S-\mathcal{E})\cup\mathcal{F}]}\cap\mathcal{E} && \text{(by the induction hypothesis).}\end{aligned}$$

By simple set theory we get the inclusion $S-\mathcal{E} \supseteq \|\psi\|^{\mathcal{T}}_{\mathcal{V}[X:=(S-\mathcal{E})\cup\mathcal{F}]}\cap(S-\mathcal{E})$. Taking the union of the sides of the two inclusions gives

$$(S-\mathcal{E})\cup\mathcal{F} \supseteq \|\psi\|^{\mathcal{T}}_{\mathcal{V}[X:=(S-\mathcal{E})\cup\mathcal{F}]}$$

and therefore $\|\mu X.\psi\|^{\mathcal{T}}_{\mathcal{V}} \subseteq (S-\mathcal{E})\cup\mathcal{F}$ by Knaster-Tarski. Taking the intersection of each side of this inclusion with $\mathcal{E}$ we get

$$\|\mu X.\psi\|^{\mathcal{T}}_{\mathcal{V}}\cap\mathcal{E} \subseteq ((S-\mathcal{E})\cup\mathcal{F})\cap\mathcal{E} = \mathcal{F} = \|\mu X.\psi\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}}.$$

□

The μL^+ submodel lemma also holds for μL because every formula of μL is equivalent to a formula of μL^+.

Lemma 3.2 (**μL Submodel Lemma**) *Let $(\mathcal{T}, \mathcal{V})$ be a modal model, with $\mathcal{T} = (S, A, \{\xrightarrow{a} \mid a \in A\})$, let $\mathcal{E}$ be a transition-closed subset of S, and let ϕ be a μL formula. Then*

$$\|\phi\|^{\mathcal{T}}_{\mathcal{V}} \cap \mathcal{E} = \|\phi\|^{\mathcal{T}\cap\mathcal{E}}_{\mathcal{V}\cap\mathcal{E}}$$

The submodel property follows easily from this lemma, as the reachable states of a process are by definition a transition-closed set.

Theorem 3.3 (**μL Submodel Property**) *Let ϕ be a closed μL formula and let $\mathcal{T} = (S, A, \{\xrightarrow{a} \mid a \in A\})$ be an LTS with E in S. Then*

$$E \models^{\mathcal{T}} \phi \Leftrightarrow E \models^{\mathcal{T}\cap\mathrm{Rch}(E)} \phi.$$

3.4.4 The Diamond-Free Mu-Calculus

If we take all μL^{+} formulas not containing any $\langle L \rangle \phi$ subformulas we obtain the logic $\mu L\Box$. In this logic one can express that an action can never occur:

$$\nu X.[a]\,\mathtt{ff} \wedge [-]X,$$

and that an action must eventually occur on all infinite computations:

$$\mu X.[-a]X.$$

Formulas of $\mu L\Box$ are preserved by the simulation preorder. Our proof of this fact is based on Stirling's proof in [88] that strongly-equivalent processes satisfy the same μL formulas. The key idea of the proof is that of a *down-set* of processes, which is a set that is closed under the simulates relation. A valuation is *down-closed* if every set in its range is a down-set. Our main lemma states that interpreting a formula of $\mu L\Box$ under a down-closed valuation results in a down-set. A virtue of the proof technique is that is works directly on the structure of $\mu L\Box$ formulas; it does not appeal to a logic with infinitary conjunction and disjunction (as in [6]).

Definition 3.2 *Let $(S, A, \{\xrightarrow{a} \mid a \in A\})$ be an LTS. A subset $\mathcal{E}$ of S is a* down-set *if, for all E and F in S,*

$$(E \in \mathcal{E} \textit{ and } F \leq E) \Rightarrow F \in \mathcal{E}$$

We first show that down-sets are preserved by union, intersection, and the function defined by the $[L]$ operator.

Lemma 3.3 *Let $\mathcal{T} = (S, A, \{\xrightarrow{a} \mid a \in A\})$ be an LTS and let subsets $\mathcal{E}, \mathcal{F}$ of S be down-sets. Then*

1. *$\mathcal{E} \cup \mathcal{F}$ is a down-set*

2. *$\mathcal{E} \cap \mathcal{F}$ is a down-set*

3. *$\|[L]\|^{\mathcal{T}} \mathcal{E}$ is a down-set*

Proof.

1. Assume $E \in \mathcal{E} \cup \mathcal{F}$ and $F \leq E$. Then E must be in $\mathcal{E}$ or $\mathcal{F}$. If $E \in \mathcal{E}$ then $F \in \mathcal{E}$, else if $E \in \mathcal{F}$ then $F \in \mathcal{F}$, so $F \in \mathcal{E} \cup \mathcal{F}$.

2. Similar to the previous case.

3. Assume $E \in \|[L]\|^{\mathcal{T}} \mathcal{E}$ and $F \leq E$. Suppose that there exists an a in L and an F' such that $F \xrightarrow{a} F'$. Then since $F \leq E$, there exists an E' such that $E \xrightarrow{a} E'$ and $F' \leq E'$. By the first assumption we know that $E' \in \mathcal{E}$, and since $F' \leq E'$ and $\mathcal{E}$ is a down-set we know that $F' \in \mathcal{E}$. Therefore $F \in \|[L]\|^{\mathcal{T}} \mathcal{E}$. $\square$

We now define, for any set $\mathcal{E}$ of processes, the sets $\mathcal{E}^{sub}$ and $\mathcal{E}^{sup}$. The next few lemmas establish that $\mathcal{E}^{sub}$ is the largest down-set contained in $\mathcal{E}$, and that $\mathcal{E}^{sup}$ is the smallest down-set containing $\mathcal{E}$.

Definition 3.3 *Let* $(S, A, \{\xrightarrow{a} \mid a \in A\})$ *be an LTS with* $\mathcal{E}$ *a subset of* S. *Then*

$$\begin{aligned}
\mathcal{E}^{sub} &\stackrel{\text{def}}{=} \{E \in \mathcal{E} \mid \forall F \in S.F \leq E \text{ implies } F \in \mathcal{E}\} \\
\mathcal{E}^{sup} &\stackrel{\text{def}}{=} \{F \in S \mid \exists E \in \mathcal{E}.F \leq E\}
\end{aligned}$$

Lemma 3.4 *Let* $(S, A, \{\xrightarrow{a} \mid a \in A\})$ *be an LTS with* $\mathcal{E}, \mathcal{F}$ *subsets of* S. *Then*

1. $\mathcal{E}^{sub} \subseteq \mathcal{E}$
2. $\mathcal{E}^{sub}$ *is a down-set*
3. *if* $\mathcal{E} \subset \mathcal{F}$ *then* $\mathcal{E}^{sub} \subseteq \mathcal{F}^{sub}$

Proof.

1. trivial
2. Assume E is in $\mathcal{E}^{sub}$ and that $F \leq E$. By the definition of $\mathcal{E}^{sub}$, F is in $\mathcal{E}$. To show that F is in $\mathcal{E}^{sub}$, by the definition of $\mathcal{E}^{sub}$ we must show that every F', such that $F' \in S$ and $F' \leq F$, is in $\mathcal{E}$. So consider such an F'. By the transitivity of $\leq$, $F' \leq E$. Then since E is in $\mathcal{E}^{sub}$, F' is in $\mathcal{E}$. Therefore F is in $\mathcal{E}^{sub}$.
3. Assume E is in $\mathcal{E}^{sub}$. Then every F in S such that $F \leq E$ is in $\mathcal{E}$, and hence also in $\mathcal{E} \cup \mathcal{F}$. So E is in $\mathcal{F}^{sub}$. □

Lemma 3.5 *Let* $(S, A, \{\xrightarrow{a} \mid a \in A\})$ *be an LTS with* $\mathcal{E}, \mathcal{F}$ *subsets of* S. *Then*

1. $\mathcal{E} \subseteq \mathcal{E}^{sup}$
2. $\mathcal{E}^{sup}$ *is a down-set*
3. *if* $\mathcal{E} \subset \mathcal{F}$ *then* $\mathcal{E}^{sup} \subseteq \mathcal{F}^{sup}$

Proof.

1. trivial

2. Assume F is in $\mathcal{E}^{sup}$ and that $F' \leq F$. By the definition of $\mathcal{E}^{sup}$ there exists an E in $\mathcal{E}$ such that $F \leq E$. By transitivity, $F' \leq E$, so F' is in $\mathcal{E}^{sup}$ by definition.

3. We first show that $(\mathcal{E} \cup \mathcal{F})^{sup} = \mathcal{E}^{sup} \cup \mathcal{F}^{sup}$:

$$
\begin{aligned}
(\mathcal{E} \cup \mathcal{F})^{sup} &= \{F \in S \mid \exists E \in (\mathcal{E} \cup \mathcal{F}).F \leq E\} \\
&= \{F \in S \mid \exists E \in \mathcal{E}.F \leq E \vee \exists E \in \mathcal{F}.F \leq E\} \\
&= \{F \in S \mid \exists E \in \mathcal{E}.F \leq E\} \cup \{F \in S \mid \exists E \in \mathcal{F}.F \leq E\} \\
&= \mathcal{E}^{sup} \cup \mathcal{F}^{sup}
\end{aligned}
$$

So $\mathcal{E}^{sup} \subseteq \mathcal{E}^{sup} \cup \mathcal{F}^{sup} \subseteq (\mathcal{E} \cup \mathcal{F})^{sup}$. □

Lemma 3.6 *Let $(S, A, \{\xrightarrow{a} \mid a \in A\})$ be an LTS with $\mathcal{E}$ a subset of S. Then*

1. *$\mathcal{E}^{sub}$ is the largest down-set contained in $\mathcal{E}$*

2. *$\mathcal{E}^{sup}$ is the smallest down-set containing $\mathcal{E}$*

Proof.

1. Assume another down-set $\mathcal{F}$ exists that is contained in $\mathcal{E}$ and that there exists a process E belonging to $\mathcal{F}$ but not $\mathcal{E}^{sub}$. By definition of down-set, we know that $E \in \mathcal{E}$ and that every F in S such that $F \leq E$ is in $\mathcal{F}$. But then F is also in $\mathcal{E}$, so E satisfies the condition for membership in $\mathcal{E}^{sub}$. Therefore $E \in \mathcal{E}^{sub}$, contrary to assumption.

2. Assume another down-set $\mathcal{F}$ exists containing $\mathcal{E}$ and that there exists a process F belonging to $\mathcal{E}^{sup}$ but not $\mathcal{F}$. By the definition of $\mathcal{E}^{sup}$, there is an element E of $\mathcal{E}$ such that $F \leq E$. Since $\mathcal{E}^{sup}$ is a superset of $\mathcal{E}$, E also belongs to $\mathcal{E}^{sup}$.

Therefore E is in $\mathcal{E}^{sup}$ and $F \leq E$, but F is not in $\mathcal{F}$, so $\mathcal{F}$ is not a down-set, contrary to assumption. $\square$

Since $\mathcal{E}^{sub}$ is the largest subset of $\mathcal{E}$ that is also a down-set, $(\mathcal{E}^{sub})^{sub}$ is just $\mathcal{E}^{sub}$, and similarly, $(\mathcal{E}^{sup})^{sup}$ is just $\mathcal{E}^{sup}$. These properties are used in the proof of the following lemma, which states that if a formula of $\mu L\Box$ is interpreted on a down-closed valuation, the result is a down-set. First we extend some of our definitions from sets to valuations.

Definition 3.4 *A valuation $\mathcal{V}$ is* down-closed *if every set in its image is a down-set.*

Definition 3.5

$$\begin{aligned} \mathcal{V}^{sub}(X) &\stackrel{\text{def}}{=} \mathcal{V}(X)^{sub} \\ \mathcal{V}^{sup}(X) &\stackrel{\text{def}}{=} \mathcal{V}(X)^{sup} \end{aligned}$$

Lemma 3.7 *Let $(\mathcal{T}, \mathcal{V})$ be a modal model, with $\mathcal{T} = (S, A, \{\stackrel{a}{\rightarrow} \mid a \in A\})$ and $\mathcal{V}$ down-closed. Let ϕ be a formula of $\mu L\Box$. Then $\|\phi\|^{\mathcal{T}}_{\mathcal{V}}$ is a down-set.*

Proof. By simultaneous induction on:

1. if $\mathcal{V}$ is down-closed then $\|\phi\|^{\mathcal{T}}_{\mathcal{V}}$ is a down-set
2. if $\|\phi\|^{\mathcal{T}}_{\mathcal{V}} \subseteq \mathcal{E}$, then $\|\phi\|^{\mathcal{T}}_{\mathcal{V}^{sub}} \subseteq \mathcal{E}^{sub}$
3. if $\mathcal{E} \subseteq \|\phi\|^{\mathcal{T}}_{\mathcal{V}}$, then $\mathcal{E}^{sup} \subseteq \|\phi\|^{\mathcal{T}}_{\mathcal{V}^{sup}}$

- Case $\phi \equiv X$. Trivial for 1, and holds for 2 and 3 by Lemmas 3.4.3 and 3.5.3.
- Case $\phi \equiv \phi_1 \wedge \phi_2$. This holds for 1 by the induction hypothesis and Lemma 3.3.1. For 2, assume that

  $$\|\phi\|^{\mathcal{T}}_{\mathcal{V}} \subseteq \mathcal{E}.$$

By the monotonicity of $\mu L\Box$ formulas, $||\phi||^{\mathcal{T}}_{\mathcal{V}^{sub}} \subseteq ||\phi||^{\mathcal{T}}_{\mathcal{V}}$, which by the transitivity of $\subseteq$ gives

$$||\phi||^{\mathcal{T}}_{\mathcal{V}^{sub}} \subseteq \mathcal{E}.$$

By Lemma 3.4.3 we know that $(||\phi||^{\mathcal{T}}_{\mathcal{V}^{sub}})^{sub} \subseteq \mathcal{E}^{sub}$. Now $\mathcal{V}^{sub}$ is down-closed, so $||\phi||^{\mathcal{T}}_{\mathcal{V}^{sub}}$ is a down-set, and thus by Lemma 3.6 $(||\phi||^{\mathcal{T}}_{\mathcal{V}^{sub}})^{sub} = ||\phi||^{\mathcal{T}}_{\mathcal{V}^{sub}}$, giving

$$||\phi||^{\mathcal{T}}_{\mathcal{V}^{sub}} \subseteq \mathcal{E}^{sub}.$$

The proof of 3 is similar.

- Case $\phi \equiv \phi_1 \vee \phi_2$. The proof is similar to the previous case.

- Case $\phi \equiv [L]\psi$. It holds for 1 by the induction hypothesis, so that $||\psi||^{\mathcal{T}}_{\mathcal{V}}$ is a down-set, and by Lemma 3.3.2. For 2 and 3 the same argument is used as for the case $\phi \equiv \phi_1 \wedge \phi_2$.

- Case $\phi \equiv \nu X.\psi$. For 1, assume that $\mathcal{V}$ is down-closed, and let $\mathcal{E}$ be $||\nu X.\psi||^{\mathcal{T}}_{\mathcal{V}}$. Since $\mathcal{E}$ is a fixed point we know

  $$\mathcal{E} \subseteq ||\psi||^{\mathcal{T}}_{\mathcal{V}[X:=\mathcal{E}]}.$$

  By induction on 3,

  $$\mathcal{E}^{sup} \subseteq ||\psi||^{\mathcal{T}}_{(\mathcal{V}[X:=\mathcal{E}])^{sup}}.$$

  Since $\mathcal{V}$ is assumed to be down-closed, $(\mathcal{V}[X := \mathcal{E}])^{sup}$ is just $\mathcal{V}[X := \mathcal{E}^{sup}]$, giving

  $$\mathcal{E}^{sup} \subseteq ||\psi||^{\mathcal{T}}_{\mathcal{V}[X:=\mathcal{E}^{sup}]}.$$

  Therefore $\mathcal{E}^{sup}$ also satisfies the condition required of greatest fixed-points, and since $\mathcal{E} \subseteq \mathcal{E}^{sup}$, it holds that $\mathcal{E} = \mathcal{E}^{sup}$. Therefore $\mathcal{E}$ is a down-set, and so is $||\nu X.\psi||_{\mathcal{V}}$.

  For 2 and 3, the same argument is used as for the case $\phi \equiv \phi_1 \wedge \phi_2$.

- Case $\phi \equiv \mu X.\psi$. The proof is similar to the previous case, except that we use *sub* sets instead of *sup* sets. $\square$

Theorem 3.4 *Let $\mathcal{T} = (S, A, \{\xrightarrow{a} \mid a \in A\})$ be an LTS, let E, F be processes in S, and let ϕ be a formula of $\mu L\square$. Then*

$$(F \models^{\mathcal{T}} \phi \text{ and } E \leq F) \Rightarrow E \models^{\mathcal{T}} \phi.$$

Proof. The empty valuation () is down-closed, so $\|\phi\|^{\mathcal{T}}_{()}$ is a down-set by Lemma 3.7. Therefore, if F is in $\|\phi\|^{\mathcal{T}}_{()}$ and $E \leq F$, then E is in $\|\phi\|^{\mathcal{T}}_{()}$. $\square$

This preservation result can be used to prove that $\mu L\square$ is strictly less expressive than μL^+.

Proposition 3.3 *μL^+ is more expressive than $\mu L\square$.*

Proof. We show that the property of deadlock-freedom, expressible in μL^+ as

$$\nu X.\langle - \rangle \, \mathtt{tt} \wedge [-]X$$

is not expressible as a formula ϕ of $\mu L\square$. Suppose it were. We know $0 \leq A \stackrel{\text{def}}{=} a.A$ and that A is deadlock-free. But then by Theorem 3.4 process 0 must be deadlock-free, which it clearly is not. $\square$

Using the idea in this proof, one can also see that the property that an action must eventually occur on all computations, written as μL^+ formula $\mu X.[-a]X \wedge \langle - \rangle \, \mathtt{tt}$, cannot be expressed in $\mu L\square$, since $0 \leq a.0$.

Not only are $\mu L\square$ formulas preserved by the simulation preorder, they characterize it: an image-finite process F satisfies at least as many $\mu L\square$ formulas as another image-finite process E just if $E \leq F$.

Theorem 3.5 *Let $\mathcal{T} = (S, A, \{\xrightarrow{a} \mid a \in A\})$ be an LTS and let E, F be image-finite processes in S. Then*

$$(\forall \phi \in \mu L\square . F \models^{\mathcal{T}} \phi \Rightarrow E \models^{\mathcal{T}} \phi) \Rightarrow (E \leq F).$$

Proof. Using the idea in the proof of Theorem 5.3.2 in [87], we show that the relation

$$\mathcal{R} \stackrel{\text{def}}{=} \{(E, F) \mid \forall \phi \in \mu L\Box . F \models^{\mathcal{T}} \phi \Rightarrow E \models^{\mathcal{T}} \phi\}$$

is a simulation. The proof is by contradiction. Suppose $\mathcal{R}$ is not a simulation. Then there exists a pair $(E, F) \in \mathcal{R}$ such that either

1. $E \xrightarrow{a} E'$ and $F \not\xrightarrow{a}$, or
2. $E \xrightarrow{a} E'$ and $F \xrightarrow{a} F' \Rightarrow (E', F') \notin \mathcal{R}$, for all F'

If 1), then $F \models^{\mathcal{T}} [a]\,\mathtt{ff}$ but $E \not\models^{\mathcal{T}} [a]\,\mathtt{ff}$, contrary to the definition of $\mathcal{R}$.

If 2), then let $\{F_1, \ldots, F_n\}$ be the set $\{F' \mid F \xrightarrow{a} F'\}$ of F-derivatives. We know that $(E', F_i) \notin \mathcal{R}$ for all $1 \leq i \leq n$, so by definition of $\mathcal{R}$ there exists for each F_i a formula ϕ_i such that $F_i \models^{\mathcal{T}} \phi_i$ and $E' \not\models^{\mathcal{T}} \phi_i$. Let ϕ be $\phi_1 \wedge \ldots \wedge \phi_n$. Then $F \models^{\mathcal{T}} [a]\phi$ but $E \not\models^{\mathcal{T}} [a]\phi$, contrary to the definition of $\mathcal{R}$. $\Box$

Notice that in the proof we only need $\mu L\Box$ formulas built from operators $\mathtt{ff}$, $\wedge$, and $[L]$.

3.4.5 The Weak Modal Mu-Calculus

The logics we have seen so far in this chapter attach no particular significance to the τ action, and hence have no special connection to CCS. We can extend μL with a new modal operator $[\![L]\!]$, which examines weak transitions of CCS processes. Let $L \subseteq \mathcal{L} \cup \{\varepsilon\}$ be a set of actions. Then the new semantic clause for $\|[\![L]\!]\phi\|^{\mathcal{T}}_{\mathcal{V}}$, where $\mathcal{T}$ is here the LTS of CCS, is as follows:

$$\begin{aligned} \|[\![L]\!]\phi\|^{\mathcal{T}} &\stackrel{\text{def}}{=} \|[\![L]\!]\|^{\mathcal{T}}(\|\phi\|^{\mathcal{T}}_{\mathcal{V}}) \\ \|[\![L]\!]\|^{\mathcal{T}}(\mathcal{E}) &\stackrel{\text{def}}{=} \{E \in \mathcal{P} \mid \forall E' \in \mathcal{P}, l \in L . E \stackrel{l}{\Rightarrow} E' \Rightarrow E' \in \mathcal{E}\}. \end{aligned}$$

Note that we write $[\![-L]\!]\phi$ as shorthand for $[\![(\mathcal{L} \cup \{\varepsilon\}) - L]\!]\phi$. Each modal subformula $[\![L]\!]\phi$ can be written as a formula of μL itself.

Proposition 3.4 *Let $L \subseteq \mathcal{L} \cup \{\varepsilon\}$ be a set of actions. Then*

$$\llbracket\varepsilon\rrbracket\phi \Leftrightarrow \nu X.\phi \wedge [\tau]X$$

$$\llbracket L\rrbracket\phi \Leftrightarrow \begin{cases} \llbracket\varepsilon\rrbracket[L]\llbracket\varepsilon\rrbracket\phi & \text{if } \varepsilon \notin L \\ \llbracket\varepsilon\rrbracket[L-\varepsilon]\llbracket\varepsilon\rrbracket\phi \wedge \llbracket\varepsilon\rrbracket\phi & \text{otherwise} \end{cases}$$

Thus adding this new operator adds no new expressive power to μL. If we take the μL formulas containing only modal subformulas of form $\llbracket L\rrbracket\phi$ we get a logic in which only weak transitions can be observed. The abstract syntax of formulas of the *weak modal mu-calculus*, written $\mu\mathbb{L}$, is as follows:

$$\phi ::= X \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \llbracket L\rrbracket\phi \mid \nu X.\phi$$

where X ranges over variables and L ranges over subsets of $\mathcal{L} \cup \{\varepsilon\}$. The interpretation $||\phi||^{\mathcal{T}}_{\mathcal{V}}$ of a $\mu\mathbb{L}$ formula ϕ is defined as for μL extended by the clause for $||\llbracket L\rrbracket\phi||^{\mathcal{T}}_{\mathcal{V}}$ above.

Not surprisingly, $\mu\mathbb{L}$ characterizes observation equivalence.

Theorem 3.6 ($\mu\mathbb{L}$ Characterisation Theorem) *Let E and F be CCS processes that are image-finite with respect to the weak CCS transition relation. Then $E \approx F$ just if, for all $\mu\mathbb{L}$ formulas ϕ,*

$$E \models \phi \Leftrightarrow F \models \phi.$$

Proof. See [88].

The operator $\langle\langle\ \rangle\rangle$ is the dual of operator $\llbracket\ \rrbracket$. We write $\mu\mathbb{L}^+$ for the weak form of μL^+ and $\mu\mathbb{L}\Box$ for the weak form of $\mu L\Box$.

Chapter 4

Abstraction with Process Operators

In this chapter we examine how processes can be abstracted with process operators. Here a unary process operator is applied to a process to obtain an abstract version of the process. We show how to transform a property satisfied by the abstract process into a property satisfied by the concrete process. Then to show that a process satisfies a property, one shows that an abstract version of process satisfies another property that, when transformed, implies the property of interest.

The property transformer we define takes a process operator and a μL formula and yields a formula of μL extended with a new modal operator. However, for many operators the transformed formula is equivalent to a μL formula. We show this for three process operators: action mapping, trace mapping, and a priority operator. We illustrate the use of these operators in verification with examples.

4.1 Static Unary Process Operators

We use both CCS operators and new process operators for abstraction. Therefore we study abstraction in a general setting. We take as processes the elements of an arbitrary process language PL. We assume of PL that its semantics is given as an LTS $\mathcal{T}$ with expressions of PL as states and actions in a set A as labels:

$$\mathcal{T} = (PL, A, \{\xrightarrow{a} \mid a \in A\})$$

We take as an abstraction operator any unary, static process operator $\mathcal{O}$ that can be applied to processes in PL. By *unary* we mean that $\mathcal{O}$ takes a single process. We write $\mathcal{O}(E)$ for the application of an operator $\mathcal{O}$ to a process E of PL. By *static* we mean that if E is a process in PL and $\mathcal{O}(E) \xrightarrow{a} E'$, then $E' \equiv \mathcal{O}(E'')$ for some process E'' in PL. In other words, $\mathcal{O}$ is retained across transitions. Examples of static, unary operators are the restriction and relabelling operators of CCS.

We also assume of $\mathcal{O}$ that its semantics is given as LTS $\mathcal{O}(\mathcal{T})$ with actions in a set B as labels:

$$\mathcal{O}(\mathcal{T}) = (\{\mathcal{O}(E) \mid E \in PL\}, B, \{\xrightarrow{a}_{\mathcal{O}} \mid a \in B\})$$

We often use $\xrightarrow{a}$ to mean the union of $\xrightarrow{a}$ from $\mathcal{T}$ and $\xrightarrow{a}_{\mathcal{O}}$ of $\mathcal{O}(\mathcal{T})$.

Note that we do not require $\mathcal{O}$ to be an operator of PL, or that $\mathcal{O}$ be able to be applied to $\mathcal{O}(E)$, for E in PL. We provide this generality because we sometimes abstract CCS processes with operators that produce actions not in *Act*. However, our results apply to the case in which $\mathcal{O}$ is an operator of PL.

We extend the application of unary process operators to sets $\mathcal{E}$ of processes by $\mathcal{O}(\mathcal{E}) \stackrel{\text{def}}{=} \{\mathcal{O}(E) \mid E \in \mathcal{E}\}$. We similarly extend the application to valuations. If $\mathcal{V}$ is a valuation from variables to processes in PL, then $(\mathcal{O}(\mathcal{V}))(X) \stackrel{\text{def}}{=} \mathcal{O}(\mathcal{V}(X))$. The following properties hold of $\mathcal{O}(\mathcal{E})$ by definition.

Proposition 4.1 *Let $\mathcal{E}$ and $\mathcal{F}$ be sets of processes. Then*

1. $\mathcal{O}(\mathcal{E} \cap \mathcal{F}) = \mathcal{O}(\mathcal{E}) \cap \mathcal{O}(\mathcal{F})$

2. $\mathcal{O}(\mathcal{E} - \mathcal{F}) = \mathcal{O}(\mathcal{E}) - \mathcal{O}(\mathcal{F})$

3. $\mathcal{E} \subseteq \mathcal{F} \Leftrightarrow \mathcal{O}(\mathcal{E}) \subseteq \mathcal{O}(\mathcal{F})$

4.2 The Property Transformer $\mathcal{W}$

The property transformer $\mathcal{W}$ establishes a correspondence between abstract and concrete processes. Namely, an abstract process $\mathcal{O}(E)$ satisfies a property ϕ just if concrete process E satisfies $\mathcal{W}_{\mathcal{O}}(\phi)$. However, while ϕ is a μL formula, $\mathcal{W}_{\mathcal{O}}(\phi)$ is a formula of the logic obtained from μL by replacing the operator $[L]$ with $[L]_{\mathcal{O}}$, where L ranges over subsets of B. Informally, $[L]_{\mathcal{O}}\phi$ is satisfied by a process E if ϕ is satisfied by every E' such that $\mathcal{O}(E')$ is an L-derivitive of the abstract process $\mathcal{O}(E)$. The formal definition of $[L]_{\mathcal{O}}\phi$ is given by the following extension to the semantics of μL.

Definition 4.1 *Let L be a subset of B and let $\mathcal{E}$ be a subset of PL. Then*

$$\begin{aligned} \|[L]_{\mathcal{O}}\phi\|_{\mathcal{V}}^{\mathcal{T}} &\stackrel{\text{def}}{=} \|[L]_{\mathcal{O}}\|^{\mathcal{T}}(\|\phi\|_{\mathcal{V}}^{\mathcal{T}}) \\ \|[L]_{\mathcal{O}}\|^{\mathcal{T}}(\mathcal{E}) &\stackrel{\text{def}}{=} \{E \in PL \mid \forall E' \in PL, a \in L.\mathcal{O}(E) \xrightarrow{a} \mathcal{O}(E') \Rightarrow E' \in \mathcal{E}\} \end{aligned}$$

Property transformer $\mathcal{W}_{\mathcal{O}}$ replaces each modal operator $[L]$, where L is a subset of B, with $[L]_{\mathcal{O}}$.

Definition 4.2 *Let $\mathcal{O}$ be a static unary operator and ϕ be a μL formula with modal subformulas of form $[L]\psi$, where L is a subset of B. Then $\mathcal{W}_{\mathcal{O}}(\psi)$ is defined inductively as follows.*

$$\begin{aligned} \mathcal{W}_{\mathcal{O}}(X) &\stackrel{\text{def}}{=} X \\ \mathcal{W}_{\mathcal{O}}(\neg\phi) &\stackrel{\text{def}}{=} \neg\mathcal{W}_{\mathcal{O}}(\phi) \\ \mathcal{W}_{\mathcal{O}}(\phi \wedge \psi) &\stackrel{\text{def}}{=} \mathcal{W}_{\mathcal{O}}(\phi) \wedge \mathcal{W}_{\mathcal{O}}(\psi) \\ \mathcal{W}_{\mathcal{O}}([L]\phi) &\stackrel{\text{def}}{=} [L]_{\mathcal{O}}\mathcal{W}_{\mathcal{O}}(\phi) \\ \mathcal{W}_{\mathcal{O}}(\nu X.\phi) &\stackrel{\text{def}}{=} \nu X.\mathcal{W}_{\mathcal{O}}(\phi) \end{aligned}$$

We write $\mu L_{\mathcal{O}}$ for the logic obtained from μL by replacing $[L]$ with $[L]_{\mathcal{O}}$.

Definition 4.3 *Let $\mathcal{O}$ be a static unary operator. Then*

$$\mu L_{\mathcal{O}} \stackrel{\text{def}}{=} \{\mathcal{W}_{\mathcal{O}}(\phi) \mid \phi \in \mu L\}$$

The logic $\mu L\Box_{\mathcal{O}}$ is defined similarly.

4.3 The Correspondence Rule

Here we show that $\mathcal{W}$ gives the intended correspondence between concrete and abstract processes. The main lemma in the proof states that interpreting ϕ on an abstract process amounts to interpreting $\mathcal{W}_{\mathcal{O}}(\phi)$ on the concrete process and then abstracting the result. The proof of the lemma is by induction on the structure of μL formulas. The following lemma is used in the modal operator case.

Lemma 4.1 *Let $\mathcal{O}$ be a static unary operator and $\mathcal{E}$ be a subset of PL. Then*

$$\mathcal{O}(\|[L]_{\mathcal{O}}\|^{\mathcal{T}}\mathcal{E}) = \|[L]\|^{\mathcal{O}(\mathcal{T})}\mathcal{O}(\mathcal{E}).$$

Proof. First we show $\mathcal{O}(\|[L]_{\mathcal{O}}\|^{\mathcal{T}}\mathcal{E}) \subseteq \|[L]\|^{\mathcal{O}(\mathcal{T})}\mathcal{O}(\mathcal{E})$. Assume $E \in \mathcal{O}(\|[L]_{\mathcal{O}}\|^{\mathcal{T}}\mathcal{E})$ so that $E \equiv \mathcal{O}(F)$ for some F, and suppose $\mathcal{O}(F) \xrightarrow{a} \mathcal{O}(F')$ and $a \in L$. Then $F' \in \mathcal{E}$ by assumption, so $\mathcal{O}(F') \in \mathcal{O}(\mathcal{E})$ by def. of $\mathcal{O}$ and therefore $E \equiv \mathcal{O}(F) \in \|[L]\|^{\mathcal{O}(\mathcal{T})}\mathcal{O}(\mathcal{E})$.

Next we show $\|[L]\|^{\mathcal{O}(\mathcal{T})}\mathcal{O}(\mathcal{E}) \subseteq \mathcal{O}(\|[L]_{\mathcal{O}}\|^{\mathcal{T}}\mathcal{E})$. Assume $E \in \|[L]\|^{\mathcal{O}(\mathcal{T})}\mathcal{O}(\mathcal{E})$ so that $E \equiv \mathcal{O}(F)$ for some F, and suppose $\mathcal{O}(F) \xrightarrow{a} \mathcal{O}(F')$ and $a \in L$. Then $\mathcal{O}(F') \in \mathcal{O}(\mathcal{E})$ by assumption, so $F' \in \mathcal{E}$ by def. of $\mathcal{O}$, and therefore $F \in \|[L]_{\mathcal{O}}\|^{\mathcal{T}}\mathcal{E}$ and $E \equiv \mathcal{O}(F) \in \mathcal{O}(\|[L]_{\mathcal{O}}\|^{\mathcal{T}}\mathcal{E})$. □

Lemma 4.2 *Let $\mathcal{O}$ be a static unary operator and ϕ be a μL formula. Then*

$$\mathcal{O}(\|\mathcal{W}_{\mathcal{O}}(\phi)\|^{\mathcal{T}}_{\mathcal{V}}) = \|\phi\|^{\mathcal{O}(\mathcal{T})}_{\mathcal{O}(\mathcal{V})}.$$

Proof. By induction on the structure of ϕ.

- Case $\phi \equiv X$. On the left we get $\mathcal{O}(\mathcal{V}(X))$, since $\mathcal{W}_{\mathcal{O}}(X)$ is X. On the right we get $\mathcal{O}(\mathcal{V})(X)$, which by definition is also $\mathcal{O}(\mathcal{V}(X))$.

- Case $\phi \equiv \phi_1 \wedge \phi_2$. We have

$$\begin{array}{lll} & \mathcal{O}(\|\mathcal{W}_{\mathcal{O}}(\phi_1 \wedge \phi_2)\|_{\mathcal{V}}^{\mathcal{T}}) & \\ = & \mathcal{O}(\|\mathcal{W}_{\mathcal{O}}(\phi_1)\|_{\mathcal{V}}^{\mathcal{T}} \cap \|\mathcal{W}_{\mathcal{O}}(\phi_2)\|_{\mathcal{V}}^{\mathcal{T}}) & \text{by def. of } \mathcal{W}_{\mathcal{O}} \text{ and } \wedge \\ = & \mathcal{O}(\|\mathcal{W}_{\mathcal{O}}(\phi_1)\|_{\mathcal{V}}^{\mathcal{T}}) \cap \mathcal{O}(\|\mathcal{W}_{\mathcal{O}}(\phi_2)\|_{\mathcal{V}}^{\mathcal{T}}) & \text{by Prop. 4.1.1} \\ = & \|\phi_1 \wedge \phi_2\|_{\mathcal{O}(\mathcal{V})}^{\mathcal{O}(\mathcal{T})} & \text{by induction and def. of } \wedge \end{array}$$

- Case $\phi \equiv \neg\psi$. Like the previous case but uses Prop. 4.1.2.

- Case $\phi \equiv [L]\psi$. We have

$$\begin{array}{lll} & \mathcal{O}(\|\mathcal{W}_{\mathcal{O}}([L]\psi)\|_{\mathcal{V}}^{\mathcal{T}}) & \\ = & \mathcal{O}(\|[L]_{\mathcal{O}}\|^{\mathcal{T}} \|\mathcal{W}_{\mathcal{O}}(\psi)\|_{\mathcal{V}}^{\mathcal{T}}) & \text{by def. of } \mathcal{O} \text{ and } [L] \\ = & \|[L]\|^{\mathcal{O}(\mathcal{T})} \mathcal{O}(\|\mathcal{W}_{\mathcal{O}}(\psi)\|_{\mathcal{V}}^{\mathcal{T}}) & \text{by Lemma 4.1} \\ = & \|[L]\psi\|_{\mathcal{O}(\mathcal{V})}^{\mathcal{O}(\mathcal{T})} & \text{by induction and the def. of } [L] \end{array}$$

- Case $\phi \equiv \nu X.\psi$. There are two cases. First we show that $\mathcal{O}(\|\mathcal{W}_{\mathcal{O}}(\nu X.\psi)\|_{\mathcal{V}}^{\mathcal{T}}) \subseteq \|\nu X.\psi\|_{\mathcal{O}(\mathcal{V})}^{\mathcal{O}(\mathcal{T})}$. Let $\mathcal{E}$ be $\|\mathcal{W}_{\mathcal{O}}(\nu X.\psi)\|_{\mathcal{V}}^{\mathcal{T}}$. Then

$$\begin{array}{lll} & \mathcal{O}(\mathcal{E}) & \\ = & \mathcal{O}(\|\nu X.\mathcal{W}_{\mathcal{O}}(\psi)\|_{\mathcal{V}}^{\mathcal{T}}) & \text{by def. of } \mathcal{W}_{\mathcal{O}} \\ = & \mathcal{O}(\|\mathcal{W}_{\mathcal{O}}(\psi)\|_{\mathcal{V}[X:=\mathcal{E}]}^{\mathcal{T}}) & \text{since } \mathcal{E} \text{ is a fixed point} \\ = & \|\psi\|_{\mathcal{O}(\mathcal{V})[X:=\mathcal{O}(\mathcal{E})]}^{\mathcal{O}(\mathcal{T})} & \text{by induction} \end{array}$$

  So by the greatest fixed-point induction rule (Prop. 2.3) $\mathcal{O}(\mathcal{E}) \subseteq \|\nu X.\psi\|_{\mathcal{O}(\mathcal{V})}^{\mathcal{O}(\mathcal{T})}$.

Next we show that $\|\nu X.\psi\|^{\mathcal{O}(\mathcal{T})}_{\mathcal{O}(\mathcal{V})} \subseteq \mathcal{O}(\|\mathcal{W}_{\mathcal{O}}(\nu X.\psi)\|^{\mathcal{T}}_{\mathcal{V}})$. Let $\mathcal{E}$ be $\|\nu X.\psi\|^{\mathcal{O}(\mathcal{T})}_{\mathcal{O}(\mathcal{V})}$. Then

$$
\begin{array}{rcll}
\mathcal{E} & \subseteq & \|\psi\|^{\mathcal{O}(\mathcal{T})}_{\mathcal{O}(\mathcal{V})[X:=\mathcal{E}]} & \text{since } \mathcal{E} \text{ is a fixed point} \\
\mathcal{O}(\mathcal{E}') & \subseteq & \|\psi\|^{\mathcal{O}(\mathcal{T})}_{\mathcal{O}(\mathcal{V})[X:=\mathcal{O}(\mathcal{E}')]} & \text{since } \mathcal{E} \text{ must be of form } \mathcal{O}(\mathcal{E}') \\
\mathcal{O}(\mathcal{E}') & \subseteq & \|\psi\|^{\mathcal{O}(\mathcal{T})}_{\mathcal{O}(\mathcal{V}[X:=\mathcal{E}'])} & \text{by def. of } \mathcal{O}(\mathcal{V}) \\
\mathcal{O}(\mathcal{E}') & \subseteq & \mathcal{O}(\|\mathcal{W}_{\mathcal{O}}(\psi)\|^{\mathcal{T}}_{\mathcal{V}[X:=\mathcal{E}']}) & \text{by induction} \\
\mathcal{E}' & \subseteq & \|\mathcal{W}_{\mathcal{O}}(\psi)\|^{\mathcal{T}}_{\mathcal{V}[X:=\mathcal{E}']} & \text{by Prop. 4.1.3} \\
\mathcal{E}' & \subseteq & \|\nu X.\mathcal{W}_{\mathcal{O}}(\psi)\|^{\mathcal{T}}_{\mathcal{V}} & \text{by greatest fixed-point induction} \\
\mathcal{O}(\mathcal{E}') & \subseteq & \mathcal{O}(\|\nu X.\mathcal{W}_{\mathcal{O}}(\psi)\|^{\mathcal{T}}_{\mathcal{V}}) & \text{by Prop. 4.1.3} \\
\mathcal{E} & \subseteq & \mathcal{O}(\|\nu X.\mathcal{W}_{\mathcal{O}}(\psi)\|^{\mathcal{T}}_{\mathcal{V}}) & \text{by def. of } \mathcal{E}'
\end{array}
$$

□

From this lemma it follows that an abstract process $\mathcal{O}(E)$ satisfies ϕ just when concrete process E satisfies $\mathcal{W}_{\mathcal{O}}(\phi)$.

Proposition 4.2 *Let $\mathcal{O}$ be a static unary process operator, ϕ be a μL formula, and E be a process of PL. Then*

$$\mathcal{O}(E) \models^{\mathcal{O}(\mathcal{T})} \phi \Leftrightarrow E \models^{\mathcal{T}} \mathcal{W}_{\mathcal{O}}(\phi).$$

Proof. By the def. of $\models$ it is enough to show that $\mathcal{O}(E) \in \|\phi\|^{\mathcal{O}(\mathcal{T})}_{()}$ just if $E \in \|\mathcal{W}_{\mathcal{O}}(\phi)\|^{\mathcal{T}}_{()}$. Suppose $\mathcal{O}(E) \in \|\phi\|^{\mathcal{O}(\mathcal{T})}_{()}$. Then equivalently $\mathcal{O}(E) \in \mathcal{O}(\|\mathcal{W}_{\mathcal{O}}(\phi)\|^{\mathcal{T}}_{()})$ by Lemma 4.2, and $E \in \|\mathcal{W}_{\mathcal{O}}(\phi)\|^{\mathcal{T}}_{()}$ by def. of $\mathcal{O}$. □

Our main theorem concerns the special case in which $\mathcal{O}$ is an operator of PL. We say that PL is *closed* under application of $\mathcal{O}$ if $E \in PL$ implies $\mathcal{O}(E) \in PL$. In this case $\mathcal{O}(PL) \subseteq PL$. The main theorem also relies on the fact that $\mathcal{O}(PL)$ is transition-closed.

Proposition 4.3 *Set $\mathcal{O}(PL)$ of processes is transition-closed.*

Proof. Suppose $E \in \mathcal{O}(PL)$ and $E \xrightarrow{a} E'$. Then $E \equiv \mathcal{O}(F)$ for some F in PL by the first assumption and the def. of $\mathcal{O}$, and $E' \equiv \mathcal{O}(F')$ for some F' in PL by the second assumption and the def. of static operator. Therefore $E' \equiv \mathcal{O}(F') \in \mathcal{O}(PL)$ by def. of $\mathcal{O}$. □

Theorem 4.1 (Correspondence Rule) *Let ϕ be a μL formula, E be a process in PL, and $\mathcal{O}$ be a static unary process operator such that PL is closed under application of $\mathcal{O}$. Then*

$$\mathcal{O}(E) \models^{\mathcal{T}} \phi \Leftrightarrow E \models^{\mathcal{T}} \mathcal{W}_{\mathcal{O}}(\phi).$$

Proof. First we show that $\mathcal{O}(\mathcal{T})$ is the submodel $\mathcal{T} \cap \mathcal{O}(PL)$ of $\mathcal{T}$ if PL is closed under application of $\mathcal{O}$. By definition $\mathcal{O}(\mathcal{T})$ is $(\mathcal{O}(PL), B, \{\xrightarrow{a}_{\mathcal{O}} \mid a \in B\})$. If PL is closed under application of $\mathcal{O}$, then $B \subseteq A$ and $\xrightarrow{a}_{\mathcal{O}} = \xrightarrow{a} \cap (\mathcal{O}(PL) \times \mathcal{O}(PL))$ for all $a \in A$. So $\mathcal{O}(\mathcal{T})$ can be written $(\mathcal{O}(PL), A, \{\xrightarrow{a} \cap (\mathcal{O}(PL) \times \mathcal{O}(PL)) \mid a \in A\})$, which is just $\mathcal{T} \cap \mathcal{O}(PL)$.

Now suppose $\mathcal{O}(E) \models^{\mathcal{T}} \phi$. Then we have, equivalently, that $\mathcal{O}(E) \in \|\phi\|^{\mathcal{T}}_{()}$ by def. of $\models$, that $\mathcal{O}(E) \in \|\phi\|^{\mathcal{T} \cap \mathcal{O}(PL)}_{() \cap \mathcal{O}(PL)}$ by the μL submodel lemma (Lemma 3.2) because $\mathcal{O}(PL)$ is a transition-closed subset of PL and $\mathcal{O}(E) \in \mathcal{O}(PL)$, that $\mathcal{O}(E) \in \|\phi\|^{\mathcal{O}(\mathcal{T})}_{()}$ because $\mathcal{O}(\mathcal{T})$ is the submodel $\mathcal{T} \cap \mathcal{O}(PL)$, that $\mathcal{O}(E) \in \|\phi\|^{\mathcal{O}(\mathcal{T})}_{\mathcal{O}(())}$ by def. of $\mathcal{O}$, that $E \in \|\mathcal{W}_{\mathcal{O}}(\phi)\|^{\mathcal{T}}_{()}$ by Prop. 4.2, and finally that $E \models^{\mathcal{T}} \mathcal{W}_{\mathcal{O}}(\phi)$ by def. of $\models$. □

The Correspondence Rule says little about the relationship between E and $\mathcal{O}(E)$. It states only that the question of whether abstract process $\mathcal{O}(E)$ satisfies formula ϕ is equivalent to the question of whether process E satisfies formula ϕ with each model operator $[L]$ of ϕ replaced by $[L]_{\mathcal{O}}$. Now we address how formulas of $\mu L_{\mathcal{O}}$ can be expressed as formulas of μL. In the following three sections we define abstraction operators $\mathcal{O}$ for every $\mu L_{\mathcal{O}}$ formula is equivalent to a μL formula.

4.4 Action Mapping

Here we extend CCS with the *action mapping* operator and show how action mapping can be used with the Correspondence Rule to verify process properties. Action mapping is a generalized relabelling operator (from [64]) that subsumes relabelling, restriction, and hiding. Action mapping is written (f), with f a partial mapping from Act to Act, similarly to the way in which relabelling is written $[f]$, with f a relabelling function.

Definition 4.4 *Let E be a process and $f : Act \rightarrow Act$ be a partial function. The* action mapping *operator (f) is defined by the following rule:*

$$\frac{E \xrightarrow{\alpha} E'}{E(f) \xrightarrow{f(\alpha)} E'(f)} \quad \alpha \in Dom(f)$$

For example, the restriction operator $\backslash L$ of CCS can be expressed as action mapping (res_L), where

$$\mathrm{res}_L(\alpha) \stackrel{\mathrm{def}}{=} \begin{cases} \alpha & \text{if } \alpha \notin L \cup \overline{L} \\ \text{undefined} & \text{otherwise} \end{cases}$$

We can also define a hiding operator $\backslash\!\backslash L$ for CCS as action mapping (hid_L), where

$$\mathrm{hid}_L(\alpha) \stackrel{\mathrm{def}}{=} \begin{cases} \tau & \text{if } \alpha \in L \\ \alpha & \text{otherwise} \end{cases}$$

Action mapping is a unary, static process operator, so by the Correspondence Rule $E(f)$ satisfies ϕ just if E satisfies $\mathcal{W}_{(f)}(\phi)$. In applying the Correspondence Rule we take CCS processes as our processes and (f) as our abstraction operator. Now we show that every $\mu L_{(f)}$ formula is equivalent to a μL formula. To do this we define a translation $T_{(f)}$ from $\mu L_{(f)}$ formulas to μL formulas.

We write $f(L)$ for $\{f(\alpha) \mid \alpha \in L \text{ and } \alpha \in \mathrm{Dom}(f)\}$ and $f^{-1}(L)$ for $\{\alpha \mid f(\alpha) \in L\}$. We further write $f(\phi)$ for the formula obtained by replacing each modal operator

$[L]$ of ϕ by $[f(L)]$, and $f^{-1}(\phi)$ for the formula obtained by replacing each modal operator $[L]$ of ϕ by $[f^{-1}(L)]$. Notice that $f(f^{-1}(L)) = L$ for every L contained in the image of f.

Definition 4.5 *Let (f) be an action mapping and ϕ be a $\mu L_{(f)}$ formula. Then $T_{(f)}(\phi)$ is defined inductively as follows.*

$$
\begin{aligned}
T_{(f)}(X) &\stackrel{\text{def}}{=} X \\
T_{(f)}(\neg\phi) &\stackrel{\text{def}}{=} \neg T_{(f)}(\phi) \\
T_{(f)}(\phi \wedge \psi) &\stackrel{\text{def}}{=} T_{(f)}(\phi) \wedge T_{(f)}(\psi) \\
T_{(f)}([L]_{(f)}\phi) &\stackrel{\text{def}}{=} [f^{-1}(L)]T_{(f)}(\phi) \\
T_{(f)}(\nu X.\phi) &\stackrel{\text{def}}{=} \nu X.T_{(f)}(\phi)
\end{aligned}
$$

Proposition 4.4 *Let ϕ be a $\mu L_{(f)}$ formula. Then*

$$\phi \Leftrightarrow T_{(f)}(\phi).$$

Proof. We show, more generally, that $\|\phi\|^{\mathcal{T}}_{\mathcal{V}} = \|T_{(f)}(\phi)\|^{\mathcal{T}}_{\mathcal{V}}$ for all modal models $(\mathcal{T}, \mathcal{V})$ by induction on the structure of ϕ. All cases except $\phi \equiv [L]_{(f)}\psi$ are routine. By def. of $T_{(f)}$ we must show in this case that $\|[L]_{(f)}\psi\|^{\mathcal{T}}_{\mathcal{V}} = \|[f^{-1}(L)]T_{(f)}(\psi)\|^{\mathcal{T}}_{\mathcal{V}}$.

$\subseteq$: Suppose $E \in \|[L]_{(f)}\psi\|^{\mathcal{T}}_{\mathcal{V}}$ and $E \xrightarrow{a} E'$ and $a \in f^{-1}(L)$. Then $E(f) \xrightarrow{f(a)} E'(f)$ and $f(a) \in L$, so by assumption $E' \in \|\psi\|^{\mathcal{T}}_{\mathcal{V}}$. By induction $E' \in \|T_{(f)}(\psi)\|^{\mathcal{T}}_{\mathcal{V}}$. Therefore $E \in \|[f^{-1}(L)]T_{(f)}(\psi)\|^{\mathcal{T}}_{\mathcal{V}}$.

$\supseteq$: Suppose $E \in \|[f^{-1}(L)]T_{(f)}(\psi)\|^{\mathcal{T}}_{\mathcal{V}}$ and $E(f) \xrightarrow{a} E'(f)$ and $a \in L$. Then $E \xrightarrow{b} E'$ and $f(b) = a$ for some b by definition of (f), so by assumption $b \in f^{-1}(L)$ and $E' \in \|T_{(f)}(\psi)\|^{\mathcal{T}}_{\mathcal{V}}$. By induction $E' \in \|\psi\|^{\mathcal{T}}_{\mathcal{V}}$. Therefore $E \in \|[L]_{(f)}\psi\|^{\mathcal{T}}_{\mathcal{V}}$. □

Notice that formula $\mathcal{W}_{(f)}(\phi)$ is equivalent to $f^{-1}(\phi)$ by Prop. 4.4 and the definition of $f(\phi)$.

4.4.1 Safe Action Mapping

To verify that process E satisfies formula ϕ with the Correspondence Rule one first finds a property ψ such that $\mathcal{W}_{\mathcal{O}}(\psi)$ implies ϕ, and then shows that $\mathcal{O}(E)$ satisfies ψ. In this section we describe a class of action mapping that simplifies verification. For such an operator (f) one can verify that E satisfies ϕ by showing that $E(f)$ satisfies formula $f(\phi)$.

An action mapping is *safe* for a formula if it maps two actions to the same action only if the two actions either both appear or both do not appear in every modal operator of the formula. If this condition holds for actions α and β in formula ϕ then we write $\alpha =_\phi \beta$.

Definition 4.6 *Let ϕ, ϕ_1, and ϕ_2 be μL formulas. Then, for all α, β in Act,*

$$\begin{aligned}
\alpha =_X \beta &\stackrel{\text{def}}{=} \text{true} \\
\alpha =_{\neg\phi} \beta &\stackrel{\text{def}}{=} \alpha =_\phi \beta \\
\alpha =_{\phi_1 \wedge \phi_2} \beta &\stackrel{\text{def}}{=} \alpha =_{\phi_1} \beta \textit{ and } \alpha =_{\phi_2} \beta \\
\alpha =_{[L]\phi} \beta &\stackrel{\text{def}}{=} (\alpha \in L \Leftrightarrow \beta \in L) \textit{ and } \alpha =_\phi \beta \\
\alpha =_{\nu X.\phi} \beta &\stackrel{\text{def}}{=} \alpha =_\phi \beta
\end{aligned}$$

Proposition 4.5 $=_\phi$ *is an equivalence relation.*

For example, consider the μL formula that expresses the property of mutual exclusion for the CCS process *MES* of Section 1.3.

$$\begin{aligned}
ME \stackrel{\text{def}}{=} \;& \nu X.[\mathtt{ex}_1, \mathtt{ex}_2]\,\mathtt{ff} \wedge [\mathtt{r}_1, \mathtt{r}_2, \tau]X \;\wedge\; [\mathtt{en}_1, \mathtt{en}_2] \\
& \nu Y.[\mathtt{en}_1, \mathtt{en}_2]\,\mathtt{ff} \wedge [\mathtt{r}_1, \mathtt{r}_2, \tau]Y \;\wedge\; [\mathtt{ex}_1, \mathtt{ex}_2]X
\end{aligned}$$

The formula has been formatted to show the similarity of the first and second parts. Informally, the first part states that no exit actions can happen initially and after any number of request or τ actions, and that after an enter action the second part of the formula must hold. The second part is similar, except with the roles of the enter and exit actions reversed. So the formula as a whole states that enter and exit actions must happen in alternation. However, it does not state that enter or exit actions must ever occur.

The actions $\mathtt{ex}_1$ and $\mathtt{ex}_2$ always appear together in ME, so $\mathtt{ex}_1 =_{ME} \mathtt{ex}_2$. As another example, deadlock freedom can be expressed in μL as $\nu X.\langle - \rangle\, \mathtt{tt} \wedge [-]X$. All the actions in the sort of the formula are equivalent relative to the formula. Notice that actions that are equivalent relative to one formula are not necessarily equivalent relative to a logically equivalent formula. For example, a and b are equivalent relative to $[a,b]\,\mathtt{ff}$, but not relative to the logically equivalent formula $[a]\,\mathtt{ff} \wedge [b]\,\mathtt{ff}$.

Definition 4.7 *An action mapping (f) is* safe *for ϕ if $f(\alpha) = f(\beta) \Rightarrow \alpha =_\phi \beta$ for all α, β in $Dom(f)$.*

Consider again the formula ME expressing mutual exclusion. The action mapping (g) is safe for ME, where

$$g(\alpha) \stackrel{\text{def}}{=} \begin{cases} \mathtt{en}_1 & \text{if } \alpha = \mathtt{en}_i\,\text{, for some } i \\ \mathtt{ex}_1 & \text{if } \alpha = \mathtt{ex}_i\,\text{, for some } i \\ \tau & \text{if } \alpha = \mathtt{r}_i\,\text{, for some } i\text{, or } \alpha = \tau \\ \alpha & \text{otherwise} \end{cases}$$

For the formula $\nu X.\langle - \rangle\, \mathtt{tt} \wedge [-]X$ that expresses deadlock freedom, the action mapping that maps all actions to τ is safe. Now we show the convenience of safe action mapping in verifying that a process satisfies a property.

Proposition 4.6 *Let ϕ be a μL formula and (f) be a safe action mapping for ϕ. Then*

$$E(f) \models f(\phi) \Leftrightarrow E \models \phi.$$

Proof. By the Correspondence Rule $E(f) \models f(\phi)$ is equivalently $E \models \mathcal{W}_{(f)}(f(\phi))$, which by Prop. 4.4 is equivalently $E \models f^{-1}(f(\phi))$. We therefore need only show that $f^{-1}(f(L)) = L$ for every subformula $[L]\psi$ of ϕ. There are two cases. If $a \in L$ then $a \in f^{-1}(f(L))$ directly by the definitions of f^{-1} and f. If $a \in f^{-1}(f(L))$ then $f(a) \in f(L)$, so there exists a b in L such that $f(a) = f(b)$. By the definition of safe action mapping we have $a =_\phi b$, and since $b \in L$ we know $a \in L$. $\square$

Thus, to see if process *MES* of Section 1.3 satisfies *ME*, we can check if $MES(g)$ satisfies formula $ME(g)$, which can be written

$$\nu X.[\mathtt{ex}_1]\,\mathtt{ff} \wedge [\tau]X \;\wedge\; [\mathtt{en}_1]\nu Y.[\mathtt{en}_1]\,\mathtt{ff} \wedge [\tau]Y \;\wedge\; [\mathtt{ex}_1]X.$$

To show that an action mapping (f) is safe for ϕ it is enough to show that f distinguishes only between the actions mentioned in a formula and other actions. In the following definition of the sort of a formula we distinguish between modal operators $[L]$ and $[-L]$. An action a is in the sort of a formula if it is found in L of an operator $[L]$ or an operator $[-L]$.

Definition 4.8 *The* sort $\mathcal{L}(\phi)$ *of a μL formula ϕ is defined by*

$$\mathcal{L}(\phi) \stackrel{\text{def}}{=} \bigcup\{L \mid [L]\psi \textit{ or } [-L]\psi \textit{ is a subformula of } \phi\}$$

Definition 4.9 *An action mapping (f) is* injective over $L \subseteq Dom(f)$ *if f restricted to L is an injection and if $f^{-1}(f(L)) \subseteq L$.*

Proposition 4.7 *If (f) is injective over $\mathcal{L}(\phi)$ then (f) is safe for ϕ.*

Proof. Assume f is injective over $\mathcal{L}(\phi)$ and $\alpha \neq_\phi \beta$. Then by definition of $=_\phi$ there exists a subformula $[L]\psi$ or subformula $[-L]\psi$ of ϕ such that $\alpha \in L$ but $\beta \notin L$. But if $\alpha \in L$ then also $\alpha \in \mathcal{L}(\phi)$, and since f is injective over $\mathcal{L}(\phi)$ there is no other action γ such that $f(\gamma) =_\phi f(\alpha)$. So $f(\beta) \neq_\phi f(\alpha)$. $\square$

Using subformulas of the form $[-L]\phi$, property *ME* can be equivalently written as

$$\nu X.[\mathtt{ex}_1, \mathtt{ex}_2]\,\mathtt{ff} \wedge [-\,\mathtt{en}_1, \mathtt{en}_2, \mathtt{ex}_1, \mathtt{ex}_2]X \;\wedge\; [\mathtt{en}_1, \mathtt{en}_2]$$
$$\nu Y.[\mathtt{en}_2, \mathtt{en}_2]\,\mathtt{ff} \wedge [-\,\mathtt{en}_1, \mathtt{en}_2, \mathtt{ex}_1, \mathtt{ex}_2]Y \;\wedge\; [\mathtt{ex}_1, \mathtt{ex}_2]X$$

Written this way, it is easy to see from Proposition 4.7 that the hiding of all but enter and exit actions is a safe action mapping for the formula.

Finally we consider two subclasses of safe action mapping. In one subclass every action is mapped to an equivalent action. In the other all equivalent actions are mapped to the same action.

Definition 4.10 *Let (f) be an action mapping that is safe for ϕ. Then (f) is* self-mapping *on ϕ if $f(\alpha) =_\phi \alpha$ for all α in $Dom(f)$, and (f) is* natural *for ϕ if $(\alpha =_\phi \beta) \Rightarrow (f(\alpha) = f(\beta))$.*

For example, the action mapping (g) used above to abstract MES is self-mapping on property ME. If (f) is self-mapping on ϕ then (f) preserves ϕ.

Proposition 4.8 *Let ϕ be a μL formula and (f) be a self-mapping on ϕ. Then*

$$E(f) \models \phi \Leftrightarrow E \models \phi.$$

Proof. By the Correspondence Rule $E(f) \models \phi$ is equivalently $E \models \mathcal{W}_{(f)}(\phi)$, which by Prop. 4.4 is equivalently $E \models f^{-1}(\phi)$. We therefore need only show that $f^{-1}(L) = L$ for every subformula $[L]\psi$ of ϕ. There are two cases. If $a \in f^{-1}(L)$ then $f(a) \in L$ by definition of $f^{-1}(L)$, and $a \in L$ since $a =_\phi f(a)$. If $a \in L$ then $f(a) \in L$ since $a =_\phi f(a)$, and $f(a) \in L$ by definition of $f^{-1}(L)$. $\Box$

For example, we know that MES satisfies ME just if $MES(g)$ satisfies ME. Proposition 4.8 also tells us that a process is deadlock-free (i.e., satisfies formula $\nu X.\langle - \rangle \mathtt{tt} \wedge [-]X$) just if it is deadlock-free after all its actions are hidden.

A natural mapping abstracts as much as safely possible. The action mapping (g) of our example is a natural one.

4.4.2 Algebraic Laws for Action Mapping

We now examine some algebraic properties of CCS plus action mapping. Strong equivalence is a congruence for the resulting notation, as CCS operators and action mapping can be defined in the *tyft/tyxt* format [47] for SOS-style semantic rules.

We now list some algebraic laws for action mapping.

Proposition 4.9 (Action Mapping laws)

$$(\alpha.E)(f) \sim \begin{cases} 0 & \textit{if } \alpha \notin \mathit{Dom}(f) \\ f(\alpha).E(f) & \textit{otherwise} \end{cases} \tag{4.1}$$

$$(E+F)(f) \sim E(f)+F(f) \tag{4.2}$$

$$E(g)(f) \sim E(f \circ g) \tag{4.3}$$

The laws for moving action mapping through the static operators of CCS have complicated side-conditions. Since relabelling and hiding are the action mappings we will use most, we now present laws for these operators. The following relabelling laws are presented but not proved in [72] as Proposition 4.8, laws 8, 10, and 11.

Proposition 4.10 (Relabelling laws)

$$(E \mid F)[f] \sim E[f] \mid F[f] \quad \textit{if } f \textit{ restricted to } L \cup \overline{L} \textit{ is injective,} \tag{4.4}$$
$$\textit{where } L = \mathcal{L}(E \mid F)$$

$$E \backslash f^{-1}(L)[f] \sim E[f] \backslash L \tag{4.5}$$

$$E[g][f] \sim E[f \circ g] \tag{4.6}$$

Proof. Each is proved by exhibiting a bisimulation relation. We will work out the details for the first law, which has the most involved proof. We will prove that the relation

$$\mathcal{R} \stackrel{\text{def}}{=} \{((E \mid F)[f], E[f] \mid F[f]) \mid E, F \in \mathcal{P} \text{ and } f \text{ a relabelling function}\}$$

is a strong bisimulation. First we consider transitions of $(E \mid F)[f]$. There are three cases, depending on whether the transition is a result of independent action by E, independent action by F, or synchronisation.

1. Suppose $(E \mid F)[f] \xrightarrow{f(\alpha)} (E' \mid F)[f]$. Then $E \mid F \xrightarrow{\alpha} E' \mid F$ and $E \xrightarrow{\alpha} E'$. From the definitions of composition and relabelling we can directly deduce that $E[f] \xrightarrow{f(\alpha)} E'[f]$ and $E[f] \mid F[f] \xrightarrow{f(\alpha)} E'[f] \mid F[f]$. Furthermore, the derived pair of processes is in $\mathcal{R}$.

2. Symmetrical to the previous case.

3. Suppose $(E \mid F)[f] \xrightarrow{\tau} (E' \mid F')[f]$ by synchronisation. Then $E \mid F \xrightarrow{\tau} E' \mid F'$ and $E \xrightarrow{l} E'$ and $F \xrightarrow{\overline{l}} F'$, for some l. Therefore $E[f] \xrightarrow{f(l)} E'[f]$ and, since $f(\overline{l}) = \overline{f(l)}$ for every relabelling function, $F[f] \xrightarrow{\overline{f(l)}} F'[f]$. So $E[f] \mid F[f] \xrightarrow{\tau} E'[f] \mid F'[f]$ and the derived processes are in $\mathcal{R}$.

Now we consider transitions of $E[f] \mid F[f]$. There are again three cases; the first two are like the first two above. For the synchronisation case, suppose $E[f] \mid F[f] \xrightarrow{\tau} E'[f] \mid F'[f]$. Then $E[f] \xrightarrow{l} E'[f]$ and $F[f] \xrightarrow{\overline{l}} F'[f]$ for some l. Since f is injective over $\mathcal{L}(E \mid F)$ and its complement, there is some l' such that $E \xrightarrow{l'} E'$ and $F \xrightarrow{\overline{l'}} F'$. So $(E \mid F)[f] \xrightarrow{\tau} (E' \mid F')[f]$, and the derived processes are in $\mathcal{R}$. □

Proposition 4.11 (Hiding laws)

$$
\begin{aligned}
(E \mid F)\backslash\!\backslash L &\sim E\backslash\!\backslash L \mid F\backslash\!\backslash L \quad \textit{if } (\mathcal{L}(E) \cap \overline{\mathcal{L}(F)}) \cap (L \cup \overline{L}) = \emptyset && (4.7)\\
E\backslash K\backslash\!\backslash L &\sim E\backslash\!\backslash L\backslash K \qquad\quad \textit{if } L \cap K = \emptyset && (4.8)\\
E[f]\backslash\!\backslash L &\sim E\backslash\!\backslash f^{-1}(L)[f] && (4.9)
\end{aligned}
$$

Proof. The proofs are similar to those for the relabelling laws. We will prove, for the first law, that the relation

$$\mathcal{R} \stackrel{\text{def}}{=} \{((E \mid F)\backslash\!\backslash L, E\backslash\!\backslash L \mid F\backslash\!\backslash L) \mid E, F \in \mathcal{P} \text{ and } \mathcal{L}(E) \cap \overline{\mathcal{L}(F)} \cap (L \cup \overline{L}) = \emptyset\}$$

is a strong bisimulation. We only show the part of the proof in which the side condition is required. Suppose $(E \mid F)\backslash\!\backslash L \xrightarrow{\tau} (E' \mid F')\backslash\!\backslash L$ by synchronisation. Then $E \xrightarrow{l} E'$ and $F \xrightarrow{\bar{l}} F'$ for some l in $\mathcal{L}(E)$ and $\overline{\mathcal{L}(F)}$. Then by the side condition l and $\bar{l}$ are not in L, so $E\backslash\!\backslash L \mid F\backslash\!\backslash L \xrightarrow{\tau} E'\backslash\!\backslash L \mid F'\backslash\!\backslash L$, and the derived processes are in $\mathcal{R}$. $\square$

4.4.3 Example

We consider Milner's task scheduler example of [72]. The scheduler controls the execution of n tasks. Action a_i signals task i to begin; action b_i signals task i to terminate. The scheduler is structured as a ring of processes, each of which controls one task. Actions c_i and $\overline{c}_{i\ominus 1}$ are used for communication between a process and its neighbours, where $\ominus$ stands for subtraction modulo n.

$$
\begin{aligned}
A_i &\stackrel{\text{def}}{=} a_i.c_i.E_i \\
E_i &\stackrel{\text{def}}{=} b_i.D_i + \overline{c}_{i\ominus 1}.b_i.A_i \\
D_i &\stackrel{\text{def}}{=} \overline{c}_{i\ominus 1}.A_i \\
\mathit{Sched} &\stackrel{\text{def}}{=} (A_0 \mid D_1 \mid D_2 \mid \cdots \mid D_{n-1})\backslash\{c_0, c_1, \ldots, c_{n-1}\}
\end{aligned}
$$

The scheduler is supposed to perform the start-task actions $\{a_i \mid 1 \leq i \leq n\}$ in cyclic order $a_1, \ldots, a_n, a_1, \ldots, a_n, \ldots$. A schema for expressing cyclic properties is the following, where $L_1, \ldots, L_k$ are disjoint sets of labels and L is $L_1 \cup \cdots \cup L_k$.

$$
\begin{aligned}
\mathit{cycle}(L_1, \ldots, L_k) \stackrel{\text{def}}{=} \; & \nu X_1.[\![L - L_1]\!]\,\mathtt{ff} \wedge [\![-L]\!]X_1 \;\wedge\; [\![L_1]\!] \\
& \nu X_2.[\![L - L_2]\!]\,\mathtt{ff} \wedge [\![-L]\!]X_2 \;\wedge\; [\![L_2]\!] \\
& \vdots \\
& \nu X_k.[\![L - L_k]\!]\,\mathtt{ff} \wedge [\![-L]\!]X_k \;\wedge\; [\![L_k]\!]X_1
\end{aligned}
$$

Thus $\mathit{cycle}(L_1, \ldots, L_k)$ expresses that an action in L_1 must occur before an action in L_2, etc. Note that this property does not require that any action ever occur. The particular property desired of the scheduler is $\mathit{cycle}(\{a_1\}, \ldots, \{a_n\})$. All b actions and τ are equal relative to this formula, so by Proposition 4.8 we know that the abstract scheduler with all b actions hidden satisfies $cycle(\{a_1\}, \ldots, \{a_k\})$ just if the concrete scheduler does. Using the hiding laws to move the hiding operator into the scheduler process, we get

$$Sched \backslash\!\!\backslash \{b_1, \ldots, b_n\} \sim (A'_1 \mid D'_2 \mid \ldots \mid D'_n) \backslash \{c_1, c_2, \ldots, c_n\}$$

where

$$\begin{aligned} A'_i &\stackrel{\text{def}}{=} a.c.E'_i \\ E'_i &\stackrel{\text{def}}{=} \tau.D'_i + d.\tau.A'_i \\ D'_i &\stackrel{\text{def}}{=} d.A'_i \end{aligned}$$

Some τ actions can be removed from this abstract version of the scheduler because $cycle(\{a_1\}, \ldots, \{a_k\})$ is a formula of $\mu I\!\!L$, the weak modal mu-calculus, and by Theorem 3.6 is preserved by observation equivalence. Therefore we can simplify $Sched \backslash\!\!\backslash \{b_1, \ldots, b_n\}$ by applying laws for observation congruence and strong equivalence to the scheduler components. First we note that $d.\tau.A' = d.A' = D'$ by Prop. 2.10 and Prop. 2.7. We also have that $\tau.D' + D' = \tau.D'$ by Prop. 2.10. The simplified scheduler component is therefore

$$\begin{aligned} A''_i &\stackrel{\text{def}}{=} a.c.E''_i \\ E''_i &\stackrel{\text{def}}{=} \tau.D''_i \\ D''_i &\stackrel{\text{def}}{=} d.A''_i \end{aligned}$$

More simplification is possible. We have $a.c.E'' = a.c.\tau.D'' = a.c.D''$ by Prop. 2.7 and Prop. 2.10, giving

$$\begin{aligned} A'''_i &\stackrel{\text{def}}{=} a.c.D'''_i \\ D'''_i &\stackrel{\text{def}}{=} d.A'''_i \end{aligned}$$

Putting all the simplification steps together, we have

$$Sched \backslash\!\!\backslash \{b_1, \ldots, b_n\} \approx (A'''_1 \mid D'''_2 \mid \ldots \mid D'''_n) \backslash \{c_1, c_2, \ldots, c_n\}$$

For n components, the abstract scheduler has $2n + 1$ states, while the concrete scheduler has more than 2^n states. For example, with seven components the two versions have 15 and 1345 states respectively. The abstract version satisfies formula $cycle(\{a_1\}, \ldots, \{a_n\})$, so the concrete scheduler does too.

4.5 Trace Mapping

Abstraction criteria [11] generalize the notion of observation captured by the weak CCS transition relation. With abstraction criteria any sequence of actions can be defined as an observation, not just those in which a label is preceded and followed by τ actions. In this section we define *trace mapping*, which is a process operator based on abstraction criteria, and show how it can be used for abstraction.

Let A and B be arbitrary sets of actions. An abstraction criterion is a set $\{e_i \mid i \in I\}$ of disjoint *abstract actions* $e_i \subseteq A^*$. Thus an abstraction criterion is a set of languages, each of which describes a type of observation. The CCS notion of observation is captured by the abstraction criterion containing only abstract actions e_l and e_ε, where e_l is strings of the form $\tau^* l \tau^*$ and e_ε is strings of the form τ^*.

We adopt a variant of abstraction criteria in which each abstract action is named by an action, as in CCS observations. We take an abstract action to be a regular language over A and an abstraction criterion to be a partial, injective function f from abstract actions to B. It is injective because no two abstract actions should have the same name. We also require that the sets in the domain of f be disjoint.

Definition 4.11 *Let f be an abstraction criterion. Then trace mapping $\mathbf{C}_f$ is defined by the following rule:*

$$\frac{E \stackrel{s}{\rightarrow} E'}{\mathbf{C}_f(E) \stackrel{f(S)}{\rightarrow} \mathbf{C}_f(E')} \quad \textit{for } s \in S \textit{ and } S \in Dom(f)$$

Trace mapping provides a way to generalize process preorders and equivalences. For example, one process f-simulates another if it simulates the other process after $\mathbf{C}_f$ is applied to both.

Definition 4.12 *F f-simulates E, written $E \leq_f F$, if $\mathbf{C}_f(E) \leq \mathbf{C}_f(F)$.*

The abstraction criteria st and wk, which capture strong and weak transitions of CCS processes respectively, are defined as follows, where we write S^* for the Kleene closure of S and $S_1\, S_2$ for the concatenation of S_1 and S_2.

$$
\begin{aligned}
st(\{\alpha\}) &\stackrel{\text{def}}{=} \alpha \\
wk(S) &\stackrel{\text{def}}{=} \begin{cases} l & \text{if } S = \tau^* l \tau^* \\ \varepsilon & \text{if } S = \tau^* \end{cases}
\end{aligned}
$$

Thus $<$ is $<_{st}$ and $\ll$ is $<_{wk}$. Notice that wk maps the set of τ strings to ε, which is not a CCS action.

Trace mapping is a unary, static process operator, so it can be used with the Correspondence Rule as an abstraction operator. In applying the rule we take as arbitrary the set of processes, and sets A and B of actions, except for the assumption that no process in the set contains trace mappings. Now we show that every $\mu L_{\mathbf{C}_f}$ formula is equivalent to a μL formula. First we show how to express in μL that ϕ holds after a string in a regular language on actions occurs.

Let S be a regular language over A and let ϕ be a μL formula. Then $SF(S, \phi)$ is the μL formula expressing that ϕ holds after a string of S occurs.

$$
\begin{aligned}
SF(\emptyset, \phi) &\stackrel{\text{def}}{=} \mathtt{tt} \\
SF(\epsilon, \phi) &\stackrel{\text{def}}{=} \phi \\
SF(\{a\}, \phi) &\stackrel{\text{def}}{=} [a]\phi \\
SF(S^*, \phi) &\stackrel{\text{def}}{=} \mu X.\phi \wedge SF(S, X) \\
SF(S_1 \cup S_2, \phi) &\stackrel{\text{def}}{=} SF(S_1, \phi) \wedge SF(S_2, \phi) \\
SF(S_1\, S_2, \phi) &\stackrel{\text{def}}{=} SF(S_1, SF(S_2, \phi))
\end{aligned}
$$

It is easy to prove by induction on the structure of S that $SF(S, \phi)$ expresses the desired property.

Proposition 4.12 *Let E and E' be processes, S be a regular language over A, and ϕ be a μL formula. Then*

$$E \models SF(S, \phi) \Leftrightarrow \forall s \in S.(E \xrightarrow{s} E' \Rightarrow E' \models \phi).$$

For f an abstraction criterion and L a subset of B, we define $f^{-1}(L)$ to be $\bigcup\{S \mid f(S) \in L\}$. If L is finite, then $f^{-1}(L)$ is a regular language, because regular languages are closed under finite union. Let ϕ be a $\mu L_{\mathbf{C}_f}$ formula in which L is finite in each modal subformula $[L]_{\mathbf{C}_f}\psi$. The equivalent μL formula is obtained by replacing each modal operator $[L]_{\mathbf{C}_f}\psi$ in ϕ by $SF(f^{-1}(L), \psi)$.

Definition 4.13 *Let f be an abstraction criterion and ϕ be a $\mu L_{\mathbf{C}_f}$ formula in which L is finite in each modal subformula $[L]_{\mathbf{C}_f}\psi$. Then $T_{\mathbf{C}_f}$ is defined inductively as for $T_{(f)}$ (see Def. 4.4) except for the following clause.*

$$T_{\mathbf{C}_f}([L]_{(f)}\phi) \stackrel{\text{def}}{=} SF(f^{-1}(L), T_{\mathbf{C}_f}(\phi))$$

Proposition 4.13 *Let ϕ be a $\mu L_{\mathbf{C}_f}$ formula in which L is finite in each modal subformula $[L]_{\mathbf{C}_f}\psi$. Then*

$$\phi \Leftrightarrow T_{\mathbf{C}_f}(\phi).$$

Proof. We show, more generally, that $\|\phi\|^{\mathcal{T}}_{\mathcal{V}} = \|T_{\mathbf{C}_f}(\phi)\|^{\mathcal{T}}_{\mathcal{V}}$ for all modal models $(\mathcal{T}, \mathcal{V})$ by induction on the structure of ϕ. All cases except $\phi \equiv [L]_{\mathbf{C}_f}\psi$ are routine. By def. of $T_{\mathbf{C}_f}$ we must show in this case that $\|[L]_{\mathbf{C}_f}\psi\|^{\mathcal{T}}_{\mathcal{V}} = \|SF(f^{-1}(L), T_{\mathbf{C}_f}(\psi))\|^{\mathcal{T}}_{\mathcal{V}}$.

$\subseteq$: Suppose $E \in \|[L]_{\mathbf{C}_f}\psi\|^{\mathcal{T}}_{\mathcal{V}}$ and $E \xrightarrow{s} E'$ and that there exists an S in $f^{-1}(L)$ such that $s \in S$. Then $\mathbf{C}_f(E) \xrightarrow{f(S)} \mathbf{C}_f(E')$ and $f(S) \in L$, so by assumption $E' \in \|\psi\|^{\mathcal{T}}_{\mathcal{V}}$. By induction $E' \in \|T_{\mathbf{C}_f}(\psi)\|^{\mathcal{T}}_{\mathcal{V}}$. Therefore $E \in \|SF(f^{-1}(L), T_{\mathbf{C}_f}(\psi))\|^{\mathcal{T}}_{\mathcal{V}}$ by Prop. 4.12.

$\supseteq$: Suppose $E \in \|SF(f^{-1}(L), T_{\mathbf{C}_f}(\psi))\|_{\mathcal{V}}^{\mathcal{T}}$ and $\mathbf{C}_f(E) \stackrel{f(S)}{\rightarrow} \mathbf{C}_f(E')$ and $f(S) \in L$. Then by def. of $\mathbf{C}_f$ there exists an s such that $E \stackrel{s}{\rightarrow} E'$ and $s \in S \subseteq f^{-1}(L)$, so $E' \in \|T_{\mathbf{C}_f}(\psi)\|_{\mathcal{V}}^{\mathcal{T}}$ by assumption and Prop. 4.12. By induction $E' \in \|\psi\|_{\mathcal{V}}^{\mathcal{T}}$. Therefore $E \in \|[L]_{\mathbf{C}_f}\psi\|_{\mathcal{V}}^{\mathcal{T}}$. □

From characterization results about μL and $\mu L\Box$ we can derive analogous results about $\mu L_{\mathbf{C}_f}$ and $\mu L\Box_{\mathbf{C}_f}$. For example, the generalized simulation relation $\leq_f$ preserves properties of $\mu L\Box_{\mathbf{C}_f}$.

Proposition 4.14 *Let E and F be processes, f be an abstraction criterion, and ϕ be a $\mu L\Box_{\mathbf{C}_f}$ formula. Then*

$$(F \models \phi \textit{ and } E \leq_f F) \Rightarrow E \models \phi.$$

Proof. Let $\mathcal{T}$ be the strong CCS transition system. Suppose $F \models^{\mathcal{T}} \phi$. Then $F \models^{\mathcal{T}} \mathcal{W}_{\mathbf{C}_f}(\psi)$ for some ψ in $\mu L\Box$ by def. of $\mu L\Box_{\mathbf{C}_f}$, and $\mathbf{C}_f(F) \models^{\mathbf{C}_f(\mathcal{T})} \psi$ by Prop. 4.2. Suppose also that $E \leq_f F$. Then $\mathbf{C}_f(F)$ simulates $\mathbf{C}_f(E)$, so $\mathbf{C}_f(E) \models^{\mathbf{C}_f(\mathcal{T})} \psi$ by Theorem 3.4. Applying Prop. 4.2 again we have that $E \models^{\mathcal{T}} \mathcal{W}_{\mathbf{C}_f}(\psi)$, so $E \models^{\mathcal{T}} \phi$. □

This result can be used to show that $\ll$ preserves properties of $\mu \mathbb{L}\Box$.

Proposition 4.15 *Let E and F be processes and ϕ be a $\mu \mathbb{L}\Box$ formula. Then*

$$(F \models \phi \textit{ and } E \ll F) \Rightarrow E \models \phi.$$

Proof. Immediately by the preceding proposition, letting f be wk. □

4.5.1 Example

Recall from Section 1.3 that *MES* is a mutual exclusion process having only $\mathtt{en}_1$ and $\mathtt{ex}_1$ as its visible actions. We show how trace mapping is helpful in verifying a property of this process. For a string s in Act^*, let $\widehat{s}$ be s with all τ actions removed. We define, following an AUTO example [55], abstraction criterion me.

$$me(S) \stackrel{\text{def}}{=} \begin{cases} \texttt{good} & \text{if } S = \{s \mid \widehat{s}\ = \texttt{en}_1\ \texttt{ex}_1\} \\ \texttt{bad} & \text{if } S = \{s \mid \widehat{s}\ = \texttt{en}_1\ \texttt{en}_1\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Suppose abstract process $\mathbf{C}_{me}(MES(g))$ satisfies $\nu X.[\texttt{bad}]\,\texttt{ff} \wedge [\texttt{good}]X$, which expresses that bad actions can never occur along paths of good actions. It might seem one can therefore infer that $\texttt{en}_1$ and $\texttt{ex}_1$ actions must occur alternately in $MES(g)$. With Prop. 4.2 we can find exactly what we know about $MES(g)$ from the fact that it satisfies $\nu X.[\texttt{bad}]\,\texttt{ff} \wedge [\texttt{good}]X$.

It is easy to show that formulas $[\texttt{good}]_{\mathbf{C}_{me}}\phi$ and $[\![\texttt{en}_1]\!][\![\texttt{ex}_1]\!]\phi$ are equivalent over CCS processes, and similarly for $[\texttt{bad}]_{\mathbf{C}_{me}}\phi$ and $[\![\texttt{en}_1]\!][\![\texttt{en}_1]\!]\phi$. Then the following are all equivalent:

$$\begin{aligned} \mathbf{C}_{me}(MES(g)) &\models \nu X.[\texttt{bad}]\,\texttt{ff} \wedge [\texttt{good}]X \\ MES(g) &\models \mathcal{W}_{\mathbf{C}_{me}}(\nu X.[\texttt{bad}]\,\texttt{ff} \wedge [\texttt{good}]X) \\ MES(g) &\models \nu X.[\texttt{bad}]_{\mathbf{C}_{me}}\,\texttt{ff} \wedge [\texttt{good}]_{\mathbf{C}_{me}}X \\ MES(g) &\models \nu X.[\![\texttt{en}_1]\!][\![\texttt{en}_1]\!]\,\texttt{ff} \wedge [\![\texttt{en}_1]\!][\![\texttt{ex}_1]\!]X \end{aligned}$$

A process that satisfies this formula might not perform $\texttt{en}_1$ and $\texttt{ex}_1$ in alternation. For example, any process that first performs $\texttt{ex}_1$ satisfies it. However, if a process performs $\texttt{en}_1$ and $\texttt{ex}_1$ alternately then it satisfies the formula.

4.6 A Priority Operator

The third abstraction operation we consider is a priority operator inspired by [26, 4]. Actions are assigned priorities, and an action can only occur if no higher-priority actions can also occur. In this section we consider the extension of CCS with this priority operator.

To define an order on actions, we assume that disjoint sets $K_1, \ldots, K_n$ of actions are defined such that $K_1 \cup \cdots \cup K_n = Act$. Then $\alpha < \beta$ holds for actions α and β if there exists i, j in $1, \ldots, n$ such that $\alpha \in K_i$ and $\beta \in K_j$ and $i < j$. Operator $P_<$ gives priority to actions greater in the $<$ order.

$$\frac{E \xrightarrow{\alpha} E'}{P_<(E) \xrightarrow{\alpha} P_<(E')} \quad E \stackrel{\beta}{\not\rightarrow} \text{ for all } \beta > \alpha$$

This operator is unary and static, so it can be used with the Correspondence Rule. In applying the rule we take CCS processes as our set of processes, take the priority operator as our abstraction operator, take Act as set A, and take $\mathcal{L} \cup \{\varepsilon\}$ as set B. We now show that every $\mu L_{P_<}$formula is equivalent to a μL formula.

Definition 4.14 *Let $P_<$ be a priority operator and ϕ be a $\mu L_{P_<}$ formula. Then $T_{P_<}$ is defined inductively as for $T_{(f)}$ (see Def. 4.4) except for the following clause.*

$$T_{P_<}([L]_{P_<}\phi) \stackrel{\text{def}}{=} \bigwedge_{1 \le i \le n} ([K_{i+1} \cup \ldots \cup K_n]\,\mathtt{ff} \Rightarrow [L \cap K_i]T_{P_<}(\phi))$$

Proposition 4.16 *Let ϕ be a $\mu L_{P_<}$ formula. Then*

$$\phi \Leftrightarrow T_{P_<}(\phi).$$

Proof. We show, more generally, that $\|\phi\|^{\mathcal{T}}_{\mathcal{V}} = \|T_{P_<}(\phi)\|^{\mathcal{T}}_{\mathcal{V}}$ for all modal models $(\mathcal{T}, \mathcal{V})$ by induction on the structure of ϕ. All cases except $\phi \equiv [L]_{(f)}\psi$ are

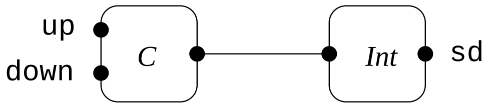


Figure 4–1: Flow Diagram for the Counting System

routine. By the definition of $T_{P_<}$ we must show in this case that $\|[L]_{P_<}\psi\|^{\mathcal{T}}_{\mathcal{V}} = \|\bigwedge_{1\le i\le n}([K_{i+1}\cup\ldots\cup K_n]\,\mathtt{ff} \Rightarrow [L\cap K_i]T_{P_<}(\psi))\|^{\mathcal{T}}_{\mathcal{V}}$.

$\subseteq$: Suppose $E \in \|[L]_{P_<}\psi\|^{\mathcal{T}}_{\mathcal{V}}$ and $E \in \|[K_{i+1}\cup\ldots\cup K_n]\,\mathtt{ff}\,\|^{\mathcal{T}}_{\mathcal{V}}$ and $E \xrightarrow{a} E'$ for some a in $L\cap K_i$. Then $P_<(E) \xrightarrow{a} P_<(E')$ by the def. of $P_<$, so by the first assumption $E' \in \|\psi\|^{\mathcal{T}}_{\mathcal{V}}$. By induction $E' \in \|T_{P_<}(\psi)\|^{\mathcal{T}}_{\mathcal{V}}$. Therefore $E \in \|\bigwedge_{1\le i\le n}([K_{i+1}\cup\ldots\cup K_n]\,\mathtt{ff} \Rightarrow [L\cap K_i]T_{P_<}(\psi))\|^{\mathcal{T}}_{\mathcal{V}}$.

$\supseteq$: Suppose $E \in \|\bigwedge_{1\le i\le n}([K_{i+1}\cup\ldots\cup K_n]\,\mathtt{ff} \Rightarrow [L\cap K_i]T_{P_<}(\psi))\|^{\mathcal{T}}_{\mathcal{V}}$ and $P_<(E) \xrightarrow{a} P_<(E')$ for some a in L. Then by def. of $P_<$ there exists an i such that $E \xrightarrow{a} E'$ and $a \in K_i$ and $E \not\xrightarrow{b}$ for all b in $K_{i+1}\cup\ldots\cup K_n$. Therefore $E' \in \|T_{P_<}(\psi)\|^{\mathcal{T}}_{\mathcal{V}}$ by assumption and $E' \in \|(\psi)\|^{\mathcal{T}}_{\mathcal{V}}$ by induction, so $E \in \|[L]_{P_<}\psi\|^{\mathcal{T}}_{\mathcal{V}}$. □

4.6.1 Example

We borrow from [26] a counting system example designed to illustrate the difficulty of describing interrupts in CCS. The flow diagram of the example is shown in Figure 4–1.

$$
\begin{aligned}
C &\stackrel{\text{def}}{=} \mathtt{up}\,.C_1 + \mathtt{i}\,.0 \\
C_{n+1} &\stackrel{\text{def}}{=} \mathtt{up}\,.C_{n+2} + \mathtt{down}\,.C_n + \mathtt{i}\,.0 \\
\mathit{Int} &\stackrel{\text{def}}{=} \mathtt{sd}\,.\overline{\mathtt{i}}\,.0 \\
\mathit{CSys} &\stackrel{\text{def}}{=} (C \mid \mathit{Int})\backslash\{\mathtt{i}\}
\end{aligned}
$$

Process *CSys* has two components: a counter and an interrupter. The counter produces `up` and `down` actions until it synchronises with action `i` of the interrupter, which is triggered by shutdown action `sd`. The intention behind *CSys* is that counting will cease after shutdown has been requested. Written in μL, this property can

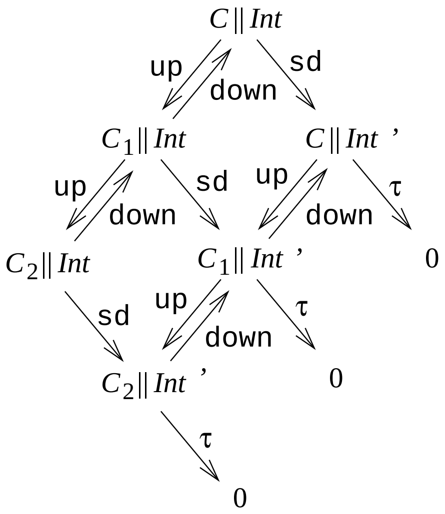


Figure 4–2: Transition Diagram for the Counting System

be expressed as the formula CS (cease after shutdown):

$$\begin{aligned} \mathit{always}(\phi) &\stackrel{\text{def}}{=} \nu X.\phi \wedge [-]X \\ CS &\stackrel{\text{def}}{=} \mathit{always}([\mathtt{sd}]\,\mathit{always}([\mathtt{up}, \mathtt{down}]\,\mathtt{ff})) \end{aligned}$$

Figure 4–2 is the transition diagram for $CSys$ with a two-step counter. In the figure we write $E \parallel F$ for $(E \mid F)\backslash\{\mathtt{i}\}$. The process clearly does not satisfy property CS.

The desired behaviour can be achieved with our priority operator. The order $<$ on CCS actions is given by the following sets:

$$\begin{aligned} K_1 &\stackrel{\text{def}}{=} \mathit{Act} - \{\mathtt{up}, \mathtt{down}, \mathtt{sd}, \tau\} \\ K_2 &\stackrel{\text{def}}{=} \{\mathtt{up}, \mathtt{down}, \mathtt{sd}\} \\ K_3 &\stackrel{\text{def}}{=} \{\tau\} \end{aligned}$$

Process $P_<(CSys)$ is then the counter with priority. Figure 4–3 shows the transition system for this process.

With the aid of the Correspondence Rule, the property CS can be checked of the prioritized counter system by checking related property $\mathcal{W}_{P_<}(CS)$ of the basic counter system. Applying Proposition 4.16, we have:

$$[L]_{P_<}\phi \;\Leftrightarrow\; ([K_2 \cup K_3]\,\mathtt{ff} \Rightarrow [L \cap K_1]\phi) \;\wedge$$

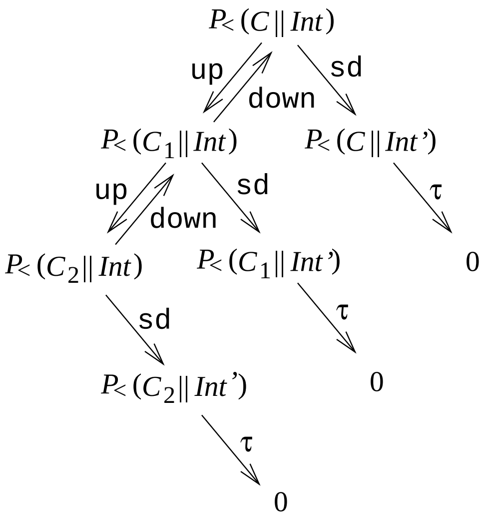


Figure 4–3: Transition Diagram for the Prioritized Counting System

$$([K_3]\,\mathtt{ff} \Rightarrow [L \cap K_2]\phi) \;\wedge$$
$$[L \cap K_3]\phi$$

Therefore, by the Correspondence Rule,

$$\mathcal{W}_{P_<}(CS) \Leftrightarrow \mathit{always}'([\tau]\,\mathtt{ff} \Rightarrow [\mathtt{sd}]\,\mathit{always}'([\tau]\,\mathtt{ff} \Rightarrow [\mathtt{up}, \mathtt{down}]\,\mathtt{ff}))$$

where

$$\mathit{always}'(\phi) \stackrel{\text{def}}{=} \nu X.\phi \;\wedge\; ([\tau]\,\mathtt{ff} \Rightarrow [\mathtt{up}, \mathtt{down}, \mathtt{sd}]X) \;\wedge\; [\tau]X.$$

This property holds of the basic counter system $CSys$, so property CS holds of the prioritized counter system $P_<(CSys)$.

Cleaveland and Hennessy [26] state that this example cannot be described correctly in any language having a semantics given by traditionally defined labelled transition systems. We have shown that it can be described in CCS once our simple priority operator is added. The use of a static operator to define priorities accomplishes what Cleaveland and Hennessy do using a two-stage operational semantics. Groote [48] uses a stratification technique in a general method for showing that a set of semantic rules using negative premises are consistent.

With this example we have shown a different way of using the Correspondence Rule. In previous examples an abstract process was used to verify a property of a concrete process. Here a basic CCS process was used to verify a property of an "abstract" process containing a priority operator. In this way the Correspondence Rule can be used to extend tools that support only basic CCS to the verification of processes in which other process operators are found.

4.7 Discussion

The main result of this chapter shows that a class of static, unary process operators can be understood as property transformers. Although process operators are often used informally as abstraction operators (e.g., the use of hiding on p. 122 of [72]), we know of no other general work on abstraction based on the use of process operators. However, the work by Larsen and Liu on contexts [65], aimed at providing a compositional proof method, is closely related. Contexts generalize the notion of process operator. An m-to-n context $\mathcal{C}$ consumes an m-tuple of actions and produces an n-tuple of actions, evolving to context $\mathcal{C}'$. Contexts are built up using composition and product operators. Composition builds an l-to-n context from an l-to-m context and an m-to-n context; product builds a $(k+m)$-to-$(l+n)$ context from a k-to-l context and an m-to-n context. The main theorem of [65] states the logical effect of context composition. If $\mathcal{C}$ is an m-to-n context, E is a 0-to-m context, and ϕ is a modal mu-calculus formula (in which n-tuples of actions may be used in modal operators) then

$$\mathcal{C} \circ E \models \phi \text{ iff } E \models \mathcal{W}(\mathcal{C}, \phi)$$

where $\circ$ is context composition and $\mathcal{W}$ is a property transformer defined in [65].

This result is similar to our Correspondence Rule. It is more general in the sense that it handles static and dynamic contexts, while our result handles only static 1-to-1 contexts. However, binary process operators are not directly handled by this result because both composition and product are required to define them. The related decomposition theorem for the product of contexts handles only formulas not containing fixed-point operators.

Our result is more general than the result for context composition in the sense that some unary process operators cannot be defined as contexts. The semantics of contexts is given operationally by a set of transduction rules, each of which has the form $\mathcal{C} \xrightarrow[\hat{a}]{\hat{b}} \mathcal{C}'$, where $\hat{a}$ is the input tuple and $\hat{b}$ is the output tuple for context $\mathcal{C}$. In this fixed format one cannot express that action a can be consumed only if b can *not* be consumed, as would be needed to define a priority operator. However, the format of transduction rules may be able to be extended.

In using the Correspondence Rule to verify that process E satisfies formula ϕ, one finds an abstraction operator $\mathcal{O}$ and a formula ψ such that $\mathcal{O}(E)$ satisfies ψ, and $\mathcal{W}_{\mathcal{O}}(\psi)$ implies ϕ. A shortcoming of this approach is that formula ψ must be guessed. It would be useful to know, for a given process E, abstraction operator $\mathcal{O}$, and property ϕ, the weakest property ψ such that E satisfies ϕ if $\mathcal{O}(E)$ satisfies ψ. We do not know the general answer to this question, but showed in Section 4.4.1 that ψ and ϕ coincide when $\mathcal{O}$ is an action mapping that is self-mapping.

Chapter 5

Applications of the Correspondence Rule

In this chapter we show how two existing process verification techniques can be derived using the Correspondence Rule. The first technique, called the Interface Rule [22], shows how to simplify components of a process according to the property to be proved of the process. The second technique, based on the property of data independence [97], shows how properties of a value-passing process over some data domain can be checked with a process over a smaller domain.

5.1 The Interface Rule

In this section we present the Interface Rule [22] in its original form and prove its soundness. In the following section we show how CCS-based versions of the rule can be derived using the Correspondence Rule and algebraic properties of CCS.

The purpose of the Interface Rule is to allow a kind of compositional reasoning about processes. It states that a property concerning the behaviour of one component can be proved by hiding the behaviour of the other components except for their behaviour at the interface with the first component. To make the rule as general as possible, it is presented in [22] for any process operators that satisfy certain simple conditions. First, it is assumed that properties of processes are built up from atomic propositions. Let E and F be processes. The set of atomic propositions of E is denoted Σ_E, the parallel composition of E and F is denoted $E \parallel F$, and the hiding of all atomic propositions of E except those in set Σ is denoted $E \downarrow \Sigma$. Parallel composition and hiding satisfy the conditions

$$\Sigma_{E \parallel F} = \Sigma_E \cup \Sigma_F \tag{5.1}$$

and

$$\Sigma_{E \downarrow \Sigma} = \Sigma_E \cap \Sigma. \tag{5.2}$$

A behavioural equivalence $\equiv$ on processes is also assumed. It is preserved by parallel composition:

$$E \equiv E' \Rightarrow (E \parallel F \equiv E' \parallel F \text{ and } F \parallel E \equiv F \parallel E'), \tag{5.3}$$

and allows hiding to be moved to one component if the hidden propositions concern only one component:

$$(E \parallel F) \downarrow \Sigma_E \equiv E \parallel (F \downarrow \Sigma_E) \text{ and } (E \parallel F) \downarrow \Sigma_F \equiv (E \downarrow \Sigma_F) \parallel F. \tag{5.4}$$

Finally, a family of process logics is assumed. The logic built from atomic propositions in Σ is $\mathcal{L}(\Sigma)$. To show that ϕ is a formula of this logic one writes $\phi \in \mathcal{L}(\Sigma)$. The satisfaction of formula ϕ by a process E is written $E \models \phi$. Equivalent processes must satisfy the same formulas:

$$\Sigma_E = \Sigma_F \text{ and } E \equiv F \Rightarrow (\forall \phi \in \mathcal{L}(\Sigma_E). E \models \phi \Leftrightarrow F \models \phi), \tag{5.5}$$

and atomic propositions not mentioned in a formula can be hidden in a process without affecting whether the process satisfies the formula[1]:

$$\phi \in \mathcal{L}(\Sigma) \Rightarrow (E \downarrow \Sigma \models \phi \Leftrightarrow E \models \phi). \tag{5.6}$$

[1]In [22] a second premise $\Sigma \subseteq \Sigma_E$ is used, but it is omitted here as the soundness of the Interface Rule does not depend on it. The paper does not explain the presence of the second premise.

The Interface Rule shows how a property ϕ can be proved of a process having parallel components E and F. If ϕ refers only to the propositions of F, then E's behaviour can be restricted to these propositions in proving that ϕ holds of the parallel composition of E and F.

$$\frac{E \downarrow \Sigma_F \equiv E' \quad \phi \in \mathcal{L}(\Sigma_F) \quad E' \parallel F \models \phi}{E \parallel F \models \phi}$$

The symmetric rule also holds. The soundness of the rule is easy to show from the conditions just listed.

Proposition 5.1 *If conditions 5.1 – 5.6 hold then the Interface Rule is sound.*

Proof. By the first hypothesis of the Interface Rule and conditions 5.3 and 5.4 we have $E' \parallel F \equiv (E \downarrow \Sigma_F) \parallel F \equiv (E \parallel F) \downarrow \Sigma_F$. Then $(E \parallel F) \downarrow \Sigma_F \models \phi$ by the third hypothesis and condition 5.5, and $E \parallel F \models \phi$ by the second hypothesis and conditions 5.1 and 5.6. □

5.2 Two Interface Rules for CCS

The Interface Rule can be instantiated for CCS by defining all the symbols of the rule in CCS such that conditions 5.1 – 5.6 hold. We present here two CCS versions of the rule. Both versions take a set of atomic propositions to be a set of actions closed under complementation: if a label l belongs to such a set then so does $\bar{l}$. The versions differ in how they interpret the symbols $\equiv$ and $\mathcal{L}(\Sigma)$.

In the rest of this section we assume that all modal operators of μL are written as $[L]$ or $[-L]$, where L is a finite set of actions. The sort $\mathcal{L}(\phi)$ of any formula ϕ (see Def. 4.8) satisfying this assumption is finite. We also apply this assumption to modal operators of the other logics we use, such as $\mu\mathbb{L}$.

5.2.1 The Strong CCS Interface Rule

In our first CCS Interface Rule the symbol $\equiv$ is taken to mean strong equivalence. Recall that $\mathcal{L}$ stands for the set $Act - \tau$ of all CCS labels.

Definition 5.1 (Strong CCS Interface Rule) *Let E and F be processes and Σ be a set of actions closed under complementation. Then*

$$\begin{aligned}
\Sigma_E &\stackrel{\text{def}}{=} \mathcal{L}(E) \cup \overline{\mathcal{L}(E)} \\
E \parallel F &\stackrel{\text{def}}{=} E \mid F \\
E \downarrow \Sigma &\stackrel{\text{def}}{=} E \backslash\!\!\backslash (\mathcal{L} - \Sigma) \\
E \equiv F &\stackrel{\text{def}}{=} E \sim F \\
\mathcal{L}(\Sigma) &\stackrel{\text{def}}{=} \{\phi \in \mu L \mid \mathcal{L}(\phi) \subseteq \Sigma - \{\tau\}\}
\end{aligned}$$

The definition is mostly straightforward. Notice that $\downarrow$ denotes the hiding of all but the given actions while $\backslash\!\!\backslash$ denotes the hiding of the given actions.

In the definition of $\mathcal{L}(\Sigma)$ we disallow formulas having τ in their sort. This is done to satisfy condition 5.6. In CCS terms, condition 5.6 states that actions not mentioned in a formula can be hidden in a process without affecting whether the process satisfies the formula. This condition is not satisfied by all μL formulas. For example,

$$a.b.0 \models [b, \tau]\,\mathtt{ff}$$

but by hiding action a,

$$\tau.b.0 \not\models [b, \tau]\,\mathtt{ff}\,.$$

However, for formulas not having τ in their sort, the condition does hold. For example,

$$a.b.0 \models [b]\,\mathtt{ff}$$

and

$$\tau.b.0 \models [b]\,\texttt{ff}\,.$$

We now prove that the conditions 5.1 – 5.6 hold of the Strong CCS Interface Rule. The last condition is the only difficult one to prove, and the only one in which the Correspondence Rule is applied.

Proposition 5.2 *Conditions 5.1 – 5.5 hold of the Strong CCS Interface Rule.*

Proof.

5.1 The proof is straightforward.

$$\begin{aligned}\Sigma_{E\|F} &= \mathcal{L}(E \mid F) \cup \overline{\mathcal{L}(E \mid F)} \\ &= \mathcal{L}(E) \cup \mathcal{L}(F) \cup \overline{\mathcal{L}(E)} \cup \overline{\mathcal{L}(F)} \\ &= (\mathcal{L}(E) \cup \overline{\mathcal{L}(E)}) \cup (\mathcal{L}(F) \cup \overline{\mathcal{L}(F)}) \\ &= \Sigma_E \cup \Sigma_F\end{aligned}$$

5.2 By definition of $E \downarrow \Sigma$ and Σ_E the set $\Sigma_{E\downarrow\Sigma}$ is the union of the set

$$\begin{aligned}& \mathcal{L}(E\backslash\!\!\backslash((\mathcal{L}(E) \cup \overline{\mathcal{L}(E)}) - \Sigma)) \\ =\; & \mathcal{L}(E) - (\mathcal{L} - \Sigma) \\ =\; & (\mathcal{L}(E) - \mathcal{L}) \cup (\mathcal{L}(E) \cap \Sigma) \\ =\; & \mathcal{L}(E) \cap \Sigma\end{aligned}$$

and its complement $\overline{\mathcal{L}(E) \cap \Sigma} = \overline{\mathcal{L}(E)} \cap \overline{\Sigma} = \overline{\mathcal{L}(E)} \cap \Sigma$. The union can therefore be written

$$(\mathcal{L}(E) \cup \overline{\mathcal{L}(E)}) \cap \Sigma$$

which is just $\Sigma_E \cap \Sigma$.

5.3 Holds because $\mid$ preserves $\simeq$ (see Section 2.2).

5.4 By definition, $(E \parallel F) \downarrow \Sigma_E$ is

$$(E \mid F) \backslash\!\backslash (\mathcal{L} - (\mathcal{L}(E) \cup \overline{\mathcal{L}(E)})).$$

We call the hidden set L. Note that L is closed under complementation and contains no action in $\mathcal{L}(E)$. By the Hiding Laws of Section 4.4.2

$$(E \mid F) \backslash\!\backslash L \simeq E \backslash\!\backslash L \mid F \backslash\!\backslash L$$

and $E \backslash\!\backslash L \simeq E$, giving

$$E \mid (F \backslash\!\backslash (\mathcal{L} - (\mathcal{L}(E) \cup \overline{\mathcal{L}(E)})))$$

which by definition is just $E \parallel (F \downarrow \Sigma_E)$.

5.5 Holds by the μL Characterisation Theorem (Theorem 3.2). $\square$

The proof of condition 5.6 is more difficult. We first show that $\mathcal{L}(\Sigma)$ is closed under subformulas.

Proposition 5.3 *If $\phi \in \mathcal{L}(\Sigma)$ and ψ is a subformula of ϕ, then $\psi \in \mathcal{L}(\Sigma)$.*

Proof. We prove that if $\phi_1 \wedge \phi_2 \in \mathcal{L}(\Sigma)$ then $\phi_1 \in \mathcal{L}(\Sigma)$ and $\phi_2 \in \mathcal{L}(\Sigma)$. The other cases are similar. Suppose $\phi_1 \wedge \phi_2 \in \mathcal{L}(\Sigma)$. Then by definition $\mathcal{L}(\phi_1 \wedge \phi_2) = \mathcal{L}(\phi_1) \cup \mathcal{L}(\phi_2) \subseteq \Sigma - \{\tau\}$, so $\mathcal{L}(\phi_1) \subseteq \Sigma - \{\tau\}$ and therefore $\phi_1 \in \mathcal{L}(\Sigma)$. By a symmetrical argument $\phi_2 \in \mathcal{L}(\Sigma)$. $\square$

Theorem 5.1 *Condition 5.6 holds of the Strong CCS Interface Rule.*

Proof. We prove something slightly more general. Let ϕ be a μL formula, $(\mathcal{T}, \mathcal{V})$ be a modal model, and L be a complementation-closed set. We prove that if

$$\mathcal{L}(\phi) \subseteq L \text{ and } \tau \notin \mathcal{L}(\phi)$$

then

$$E\backslash\!\!\backslash(\mathcal{L} - L) \in \|\phi\|^{\mathcal{T}}_{\mathcal{V}} \Leftrightarrow E \in \|\phi\|^{\mathcal{T}}_{\mathcal{V}}.$$

The proof is by induction on the structure of ϕ. Since the definition of $\mathcal{L}(\phi)$ distinguishes $[K]$ and $[-K]$, we treat these as separate cases.

- Case $\phi \equiv X$. Trivial.

- Case $\phi \equiv \phi_1 \wedge \phi_2$. By Proposition 5.3 and the induction hypothesis we have $E\backslash\!\!\backslash(\mathcal{L} - L) \in \|\phi_1\|^{\mathcal{T}}_{\mathcal{V}} \Leftrightarrow E \in \|\phi_1\|^{\mathcal{T}}_{\mathcal{V}}$ and similarly for ϕ_2, which gives us $E\backslash\!\!\backslash(\mathcal{L} - L) \in \|\phi_1 \wedge \phi_2\|^{\mathcal{T}}_{\mathcal{V}} \Leftrightarrow E \in \|\phi_1 \wedge \phi_2\|^{\mathcal{T}}_{\mathcal{V}}$.

- Case $\phi \equiv \neg\phi'$. Similar to the previous case.

- Case $\phi \equiv [K]\phi'$. We assume $\mathcal{L}([K]\phi') \subseteq L$ and $\tau \notin \mathcal{L}([K]\phi')$ and show that

  $$E\backslash\!\!\backslash(\mathcal{L} - L) \in \|[K]\phi'\|^{\mathcal{T}}_{\mathcal{V}} \Leftrightarrow E \in \|[K]\phi'\|.$$

  $\Rightarrow$: Assume $E\backslash\!\!\backslash(\mathcal{L} - L) \in \|[K]\phi'\|^{\mathcal{T}}_{\mathcal{V}}$ and $E \xrightarrow{\alpha} E'$ and $\alpha \in K$. Since $\mathcal{L}([K]\phi') \subseteq L$ we know $K \subseteq L$ and therefore $\alpha \notin \mathcal{L} - L$. So $E\backslash\!\!\backslash(\mathcal{L} - L) \xrightarrow{\alpha} E'\backslash\!\!\backslash(\mathcal{L} - L)$ and by assumption $E'\backslash\!\!\backslash(\mathcal{L} - L) \in \|\phi'\|^{\mathcal{T}}_{\mathcal{V}}$. Then $E' \in \|\phi'\|^{\mathcal{T}}_{\mathcal{V}}$ by Prop. 5.3 and the induction hypothesis, and therefore $E \in \|[K]\phi'\|^{\mathcal{T}}_{\mathcal{V}}$.

  $\Leftarrow$: Assume $E \in \|[K]\phi'\|^{\mathcal{T}}_{\mathcal{V}}$ and $E\backslash\!\!\backslash(\mathcal{L} - L) \xrightarrow{\alpha} E'\backslash\!\!\backslash(\mathcal{L} - L)$ and $\alpha \in K$. Since $\tau \notin \mathcal{L}([K]\phi')$ we know $\tau \notin K$ and therefore $\alpha \neq \tau$. So $E \xrightarrow{\alpha} E'$ and by assumption $E' \in \|\phi'\|^{\mathcal{T}}_{\mathcal{V}}$. Then $E'\backslash\!\!\backslash(\mathcal{L} - L) \in \|\phi'\|^{\mathcal{T}}_{\mathcal{V}}$ by Prop. 5.3 and the induction hypothesis, and therefore $E\backslash\!\!\backslash(\mathcal{L} - L) \in \|[K]\phi'\|^{\mathcal{T}}_{\mathcal{V}}$.

- Case $\phi \equiv [-K]\phi'$. We assume $\mathcal{L}([-K]\phi') \subseteq L$ and $\tau \notin \mathcal{L}([-K]\phi')$ and show that

  $$E\backslash\!\!\backslash(\mathcal{L} - L) \in \|[-K]\phi'\|^{\mathcal{T}}_{\mathcal{V}} \Leftrightarrow E \in \|[-K]\phi'\|.$$

$\Rightarrow$: Assume $E\backslash\!\backslash(\mathcal{L}-L) \in \|[-K]\phi'\|^{\mathcal{T}}_{\mathcal{V}}$ and $E \xrightarrow{\alpha} E'$ and $\alpha \in Act - K$. By def. of hiding $E\backslash\!\backslash(\mathcal{L}-L) \xrightarrow{\beta} E'\backslash\!\backslash(\mathcal{L}-L)$ with β either τ or α. If $\beta = \alpha$ then $\beta \in Act - K$ by the assumption that $\alpha \in Act - K$. If $\beta = \tau$ then $\beta \in Act - K$ by the assumption that $\tau \notin \mathcal{L}([-K]\phi')$. Therefore $E'\backslash\!\backslash(\mathcal{L}-L) \in \|\phi'\|^{\mathcal{T}}_{\mathcal{V}}$ by assumption. Then $E' \in \|\phi'\|^{\mathcal{T}}_{\mathcal{V}}$ by Prop. 5.3 and the induction hypothesis, and therefore $E \in \|[-K]\phi'\|^{\mathcal{T}}_{\mathcal{V}}$.

$\Leftarrow$: Assume $E \in \|[-K]\phi'\|^{\mathcal{T}}_{\mathcal{V}}$ and $E\backslash\!\backslash(\mathcal{L}-L) \xrightarrow{\alpha} E'\backslash\!\backslash(\mathcal{L}-L)$ and $\alpha \in Act - K$. We do case analysis on α. If $\alpha = \tau$ then $E \xrightarrow{\beta} E'$ with $\beta = \tau$ or $\beta \in \mathcal{L} - L$. Since $K \subseteq L$ we have $\beta \in \{\tau\} \cup (\mathcal{L} - L) = Act - L \subseteq Act - K$. If $\alpha \neq \tau$ then $E \xrightarrow{\beta} E'$ with $\beta = \alpha \in Act - K$. In both cases $E \xrightarrow{\beta} E'$ with $\beta \in Act - K$, so by assumption $E' \in \|\phi'\|^{\mathcal{T}}_{\mathcal{V}}$. Then $E'\backslash\!\backslash(\mathcal{L}-L) \in \|\phi'\|^{\mathcal{T}}_{\mathcal{V}}$ by Prop. 5.3 and the induction hypothesis, and therefore $E\backslash\!\backslash(\mathcal{L}-L) \in \|[-K]\phi'\|^{\mathcal{T}}_{\mathcal{V}}$.

- Case $\phi \equiv \nu X.\phi'$. Let $\mathcal{E} = \|\nu X.\phi'\|^{\mathcal{T}}_{\mathcal{V}}$. Suppose $E\backslash\!\backslash(\mathcal{L}-L) \in \|\nu X.\phi'\|^{\mathcal{T}}_{\mathcal{V}}$. By def. of $\|\nu X.\phi'\|^{\mathcal{T}}_{\mathcal{V}}$ we have equivalently $E\backslash\!\backslash(\mathcal{L}-L) \in \|\phi'\|^{\mathcal{T}}_{\mathcal{V}[X:=\mathcal{E}]}$, and by Prop. 5.3 and the induction hypothesis we have equivalently $E \in \|\phi'\|^{\mathcal{T}}_{\mathcal{V}[X:=\mathcal{E}]}$. Then by the def. of $\|\nu X.\phi'\|^{\mathcal{T}}_{\mathcal{V}}$ we have equivalently $E \in \|\nu X.\phi'\|^{\mathcal{T}}_{\mathcal{V}}$. $\square$

A limitation of the Strong CCS Interface Rule is that it can only be used to prove statements of the form $E \mid F \models \phi$, while we often want to prove statements of the form $(E \mid F)\backslash L \models \phi$. However, by the Correspondence Rule and Prop. 4.4 we know $(E \mid F)\backslash L \models \phi$ just if $E \mid F \models \mathcal{W}_{\backslash L}(\phi)$, which is of the form suitable for proof with the Strong CCS Interface Rule.

5.2.2 The Weak CCS Interface Rule

In our second CCS Interface Rule the symbol $\equiv$ is taken to mean observation equivalence.

Definition 5.2 (Weak CCS Interface Rule) *Let E and F be processes and let Σ be a set of labels closed under complementation. Then*

$$\begin{aligned}
\Sigma_E &\stackrel{\text{def}}{=} \mathcal{L}(E) \cup \overline{\mathcal{L}(E)} \\
E \parallel F &\stackrel{\text{def}}{=} E \mid F \\
E \downarrow \Sigma &\stackrel{\text{def}}{=} E \backslash\!\backslash (\mathcal{L} - \Sigma) \\
E \equiv F &\stackrel{\text{def}}{=} E \approx F \\
\mathcal{L}(\Sigma) &\stackrel{\text{def}}{=} \{\phi \in \mu\mathbb{L}(\Sigma) \mid \mathcal{L}(\phi) \subseteq \Sigma\}
\end{aligned}$$

The definition of $\mathcal{L}(\Sigma)$ refers to logic $\mu\mathbb{L}(K)$, which is obtained by replacing the $[\![L]\!]$ operator of $\mu\mathbb{L}$ with operator $[\![L]\!]_K$, which is defined by

$$\begin{aligned}
[\![L]\!]_K\phi &\stackrel{\text{def}}{=} [\![\varepsilon]\!]_K[L][\![\varepsilon]\!]_K\phi \\
[\![\varepsilon]\!]_K\phi &\stackrel{\text{def}}{=} \nu X.[\![-K, \varepsilon]\!]X \;\wedge\; [\![\varepsilon]\!]\phi
\end{aligned}$$

where $L, K \subseteq \mathcal{L}$. Informally, operator $[\![L]\!]_K$ is just like $[\![L]\!]$ except that in $[\![L]\!]_K$ we treat every action not in K as hidden. The sort $\mathcal{L}(\phi)$ of a $\mu\mathbb{L}(K)$ formula ϕ is defined just as for a μL formula.

Condition 5.6 does not hold if $\mu\mathbb{L}$ is used instead of $\mu\mathbb{L}(K)$. For example,

$$a.b.0 \models [\![b]\!]\,\texttt{ff}$$

but hiding action a, which does not appear in $[\![b]\!]\,\texttt{ff}$, gives

$$\tau.b.0 \not\models [\![b]\!]\,\texttt{ff}\,.$$

Most of conditions 5.1-5.6 are true here for the same reasons as in the Strong CCS Interface Rule, because Σ_E, $\parallel$, and $\downarrow$ are defined the same in both versions of the rule. Condition 5.5, which states that equivalent processes satisfy the same formulas, holds because every $\mathcal{L}(\Sigma)$ formula is equivalent to a $\mu\mathbb{L}$ formula.

Proposition 5.4 *Every $\mu\mathbb{L}(K)$ formula is equivalent to a $\mu\mathbb{L}$ formula.*

Proof. Formulas of $\mu\mathbb{L}(K)$ and $\mu\mathbb{L}$ differ only in their modal subformulas. Let ϕ be a $\mu\mathbb{L}(K)$ formula. A subformula $[\![\varepsilon]\!]_K\psi$ of ϕ is by definition equivalent to a $\mu\mathbb{L}$ formula. For a subformula $[\![L]\!]_K\psi$ of ϕ, note that $[\![\varepsilon]\!][\![\varepsilon]\!]_K\phi \Leftrightarrow [\![\varepsilon]\!]_K\phi \Leftrightarrow [\![\varepsilon]\!]_K[\![\varepsilon]\!]\phi$. Then

$$\begin{array}{ll} & [\![L]\!]_K\psi \\ \Leftrightarrow & [\![\varepsilon]\!]_K[L][\![\varepsilon]\!]_K\psi \\ \Leftrightarrow & [\![\varepsilon]\!]_K[\![\varepsilon]\!][L][\![\varepsilon]\!][\![\varepsilon]\!]_K\psi \\ \Leftrightarrow & [\![\varepsilon]\!]_K[\![L]\!][\![\varepsilon]\!]_K\psi \end{array}$$

which is also equivalent to a $\mu\mathbb{L}$ formula. □

Condition 5.6, which states that hiding actions not in a formula does not affect satisfaction of the formula, holds because every formula of $\mathcal{L}(\Sigma)$ here is equivalent to a formula of $\mathcal{L}(\Sigma)$ in the Strong CCS Interface Rule. We show this in the next two propositions.

Proposition 5.5 *Let K be a set of labels. Then*

$$[\![\varepsilon]\!]_K\phi \Leftrightarrow \nu X.[-K]X \wedge \phi.$$

Proof. $\Rightarrow$: Suppose $E \models [\![\varepsilon]\!]_K\phi$ and $E\xrightarrow{\alpha_1}\cdots\xrightarrow{\alpha_n}E'$, where $n \geq 0$ and each α_i is in $Act - K$. Let $i_1, \ldots, i_m$ be the indices of non-τ actions in $\alpha_1, \ldots, \alpha_n$. Then

$$E(\xrightarrow{\tau})^*\xrightarrow{\alpha_{i_1}}\cdots(\xrightarrow{\tau})^*\xrightarrow{\alpha_{i_m}}(\xrightarrow{\tau})^*E'.$$

Therefore $E\xRightarrow{\alpha_{i_1}}\cdots\xRightarrow{\alpha_{i_m}}\xRightarrow{\varepsilon}E'$ and by assumption and the definition of $[\![\varepsilon]\!]_K$ we know $E' \models \phi$.

$\Leftarrow$: Suppose $E \models \nu X.[-K]X \wedge \phi$ and $E\xRightarrow{\alpha_1}\cdots\xRightarrow{\alpha_n}\xRightarrow{\varepsilon}E'$, where each α_i is in $\mathcal{L} - K$. Then

$$E(\xrightarrow{\tau})^*\xrightarrow{\alpha_1}(\xrightarrow{\tau})^*\cdots(\xrightarrow{\tau})^*\xrightarrow{\alpha_n}(\xrightarrow{\tau})^*(\xrightarrow{\tau})^*E'.$$

Therefore $E \xrightarrow{\beta_1} \cdots \xrightarrow{\beta_m} E'$, where each β_i is in $Act - K$, and by assumption $E' \models \phi$.
$\square$

Proposition 5.6 *Let K be a complementation-closed set of actions. Then every formula ϕ of $\mathcal{L}(K)$ is equivalent to a formula ψ of μL such that*

$$\mathcal{L}(\psi) \subseteq K - \{\tau\}.$$

Proof. We construct ψ from ϕ using the following translation function T from $\mu\mathbb{L}(K)$ to μL:

$$\begin{aligned}
T(X) &\stackrel{\text{def}}{=} X \\
T([\![L]\!]_K\phi) &\stackrel{\text{def}}{=} \nu X.[-K]X \wedge [L](\nu Y.[-K]Y \wedge T(\phi)) \\
T([\![\varepsilon]\!]_K\phi) &\stackrel{\text{def}}{=} \nu X.[-K]X \wedge T(\phi) \\
T(\neg\phi) &\stackrel{\text{def}}{=} \neg T(\phi) \\
T(\phi \wedge \psi) &\stackrel{\text{def}}{=} T(\phi) \wedge T(\psi) \\
T(\nu X.\phi) &\stackrel{\text{def}}{=} \nu X.T(\phi)
\end{aligned}$$

Let ψ be $T(\phi)$. By Proposition 5.5 and the definition of $[\![L]\!]_K$ formulas ϕ and ψ are equivalent.

It is clear from the definition of T that $\tau \notin \mathcal{L}(\psi)$. We must also show that $\mathcal{L}(\psi) \subseteq K$. By definition of $\mathcal{L}(\psi)$, this means that $L \subseteq K$ for every subformula $[L]\psi'$ or $[-L]\psi'$ of ψ. By definition of T, a subformula of form $[L]\psi'$ must be derived from a subformula $[\![L]\!]_K\phi'$ of ϕ, and since $\mathcal{L}(\phi) \subseteq K$ it must be that $L \subseteq K$. A subformula of form $[\![-L]\!]\psi'$ must either be derived from a subformula $[\![-L]\!]_K$, in which case $L \subseteq K$ as before, or satisfy $K = L$, so that $L \subseteq K$ is also true. $\square$

Theorem 5.2 *Conditions 5.1 – 5.6 hold of the Weak CCS Interface Rule.*

5.1 Holds as in Proposition 5.2, since Σ_E and $\|$ are defined here just as in the Strong CCS Interface Rule.

5.2 Holds as in Proposition 5.2, since Σ_E and $\downarrow$ are defined here just as in the Strong CCS Interface Rule.

5.3 Holds because $|$ preserves $\approx$ (Prop. 5.11 of [72]).

5.4 Holds as in Proposition 5.2, since Σ_E, $\downarrow$, and $\|$ are defined here just as in the Strong CCS Interface Rule.

5.5 Holds by Theorem 3.6 (the $\mu\mathbb{L}$ Characterisation Theorem), since by Proposition 5.4 every $\mathcal{L}(\Sigma)$ formula is equivalent to a $\mu\mathbb{L}$ formula.

5.6 Holds as in Proposition 5.2, since by Proposition 5.6 every $\mathcal{L}(\Sigma)$ formula is equivalent to a formula of $\mathcal{L}(\Sigma)$ as defined in the Strong CCS Interface Rule, and Σ_E and $\downarrow$ are defined here just as in the Strong CCS Interface Rule. $\square$

5.2.3 Example

The Interface Rule is illustrated in [22] through an arbiter example. The verification of the arbiter is sketched there, but the Interface Rule is never directly applied. Indeed, the Interface Rule is not as useful for the arbiter as another idea also based on the conditions of the Interface Rule. In this section we present the arbiter example, explain why verification of the arbiter with the Interface Rule is inconvenient, and verify the arbiter with a related idea.

The job of an arbiter is to arbitrate requests for a single resource from multiple users. An n-user arbiter process thus sits between a single resource process and n user processes. The interface of an n-user arbiter is shown in Figure 5–1. To acquire the resource, user i performs $\mathtt{acq}_i$ and then waits for the acknowledgement signal $\mathtt{aa}_i$. To release the resource, user i performs $\mathtt{rel}_i$ and then waits for the acknowledgement signal $\mathtt{ar}_i$. The arbiter itself uses the same protocol in acquiring and releasing the resource. A critical property of the arbiter is that at most one user must be given access to the resource at a time.

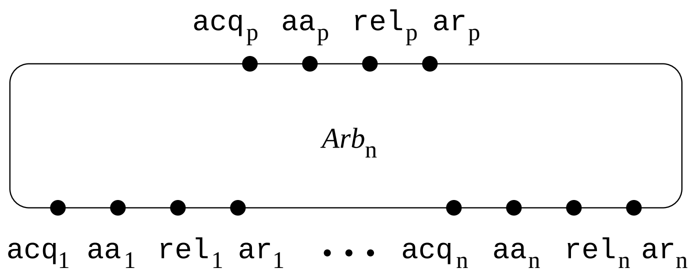


Figure 5–1: The Arbiter Interface

An arbiter can be implemented as a tree built of simple cells, each of which is a 2-user arbiter. A cell is itself built out of components A_1, A_2, B, and C.

$$
\begin{aligned}
A_i &\stackrel{\text{def}}{=} \mathtt{acq}_i \,.\, \overline{\mathtt{r}}_i \,.\, \mathtt{s}_i \,.\, \mathtt{aa}_i \,.\, \mathtt{rel}_i \,.\, \overline{\mathtt{r}}_i \,.\, \mathtt{s}_i \,.\, \mathtt{ar}_i \,.A_i \\
B &\stackrel{\text{def}}{=} \overline{\mathtt{r}}_p \,.\, \mathtt{acq}_p \,.\, \mathtt{aa}_p \,.(B_1 + B_2) \\
B_i &\stackrel{\text{def}}{=} \overline{\mathtt{s}}_i \,.\, \overline{\mathtt{r}}_p \,.\, \mathtt{rel}_p \,.\, \mathtt{ar}_p \,.\, \mathtt{s}_i \,.B \\
C &\stackrel{\text{def}}{=} \mathtt{r}_1 \,.C_1 + \mathtt{r}_2 \,.C_2 \\
C_1 &\stackrel{\text{def}}{=} \mathtt{r}_2 \,.\, \mathtt{r}_p \,.C_3 + \mathtt{r}_p \,.(\mathtt{r}_1 \,.(\mathtt{r}_p \,.C + \mathtt{r}_2 \,.\, \mathtt{r}_p \,.C2) + \mathtt{r}_2 \,.C_3) \\
C_2 &\stackrel{\text{def}}{=} \mathtt{r}_1 \,.\, \mathtt{r}_p \,.C_3 + \mathtt{r}_p \,.(\mathtt{r}_2 \,.(\mathtt{r}_p \,.C + \mathtt{r}_1 \,.\, \mathtt{r}_p \,.C2) + \mathtt{r}_1 \,.C_3) \\
C_3 &\stackrel{\text{def}}{=} \mathtt{r}_1 \,.\, \mathtt{r}_p \,.C_2 + \mathtt{r}_2 \,.\, \mathtt{r}_p \,.C_1 \\
\mathit{Cell} &\stackrel{\text{def}}{=} (A_1 \mid A_2 \mid B \mid C)\backslash\{\mathtt{r}_1, \mathtt{r}_2, \mathtt{s}_1, \mathtt{s}_2, \mathtt{r}_p\}
\end{aligned}
$$

Transition diagrams for the components are shown in Figure 5–2. Informally, processes A_1 and A_2 relay user requests to both B and C. Process B gets the resource and performs the arbitration. Process C synchronises with B after a user request, ensuring that B does not acquire the resource unless a user wants it. The flow diagram for a cell is shown in Figure 5–3.

An n-user arbiter is defined inductively, starting with the 2-user cell. To simplify relabelling we define a parametrised version of process *Cell*. In the following definition, x, y, and z are arbitrary values.

$$
\begin{aligned}
\mathit{Cell}(x,y,z) \stackrel{\text{def}}{=} \mathit{Cell}[&\mathtt{acq}_x \,/\, \mathtt{acq}_p, \mathtt{aa}_x \,/\, \mathtt{aa}_p, \mathtt{rel}_x \,/\, \mathtt{rel}_p, \mathtt{ar}_x \,/\, \mathtt{ar}_p, \\
&\mathtt{acq}_y \,/\, \mathtt{acq}_1, \mathtt{aa}_y \,/\, \mathtt{aa}_1, \mathtt{rel}_y \,/\, \mathtt{rel}_1, \mathtt{ar}_y \,/\, \mathtt{ar}_1, \\
&\mathtt{acq}_z \,/\, \mathtt{acq}_2, \mathtt{aa}_z \,/\, \mathtt{aa}_2, \mathtt{rel}_z \,/\, \mathtt{rel}_2, \mathtt{ar}_z \,/\, \mathtt{ar}_2]
\end{aligned}
$$

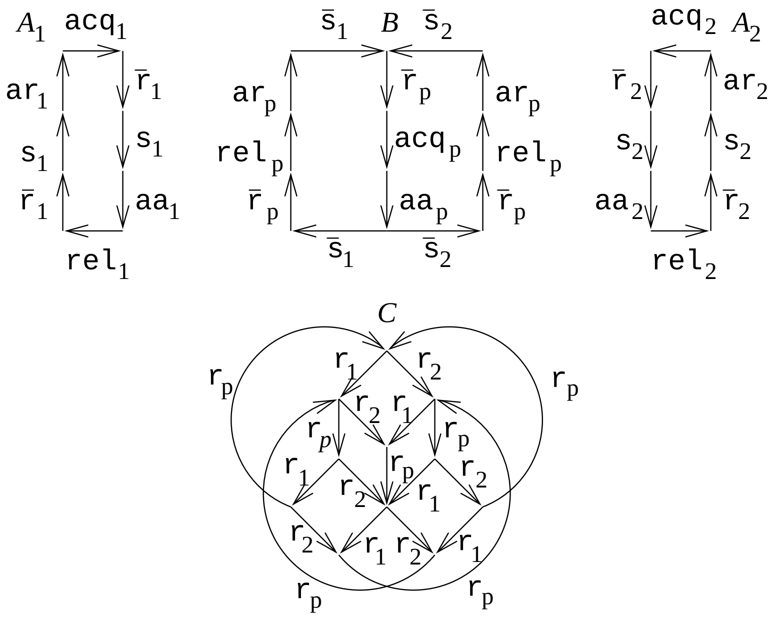


Figure 5–2: Components of an Arbiter Cell

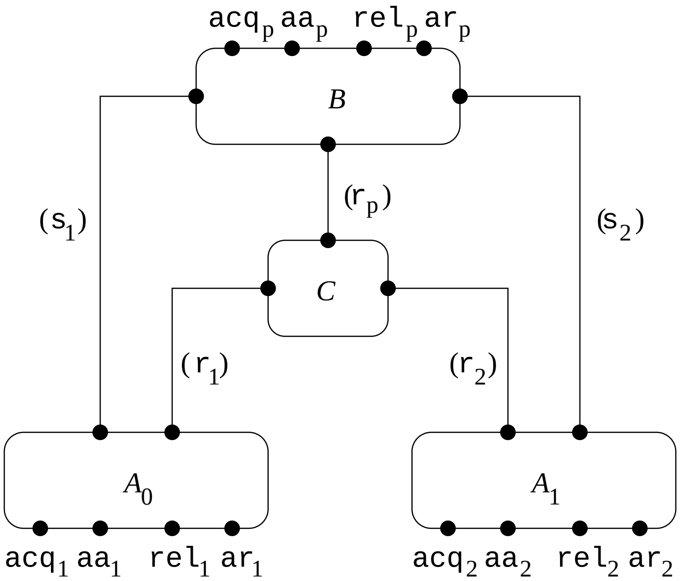


Figure 5–3: Flow Diagram for an Arbiter Cell

An $(n+1)$-user arbiter is built by composing an n-user arbiter with a single cell.

$$Arb_2(p, x, y) \stackrel{\text{def}}{=} Cell(p, x, y)$$
$$Arb_{n+3}(p, x_1, \ldots, x_{n+3}) \stackrel{\text{def}}{=} (Arb_{n+2}(p, x_1, \ldots, x_{n+1}, x) \mid Cell(x, x_{n+2}, x_{n+3})) \backslash L_x$$

where $L_x \stackrel{\text{def}}{=} \{\mathtt{acq}_x, \mathtt{aa}_x, \mathtt{rel}_x, \mathtt{ar}_x\}$ and x is distinct from p and every element of $\{x_1, \ldots, x_{n+3}\}$. A flow diagram for $Arb_{n+3}(p, 1, \ldots, n+3)$ is shown in Figure 5–4.

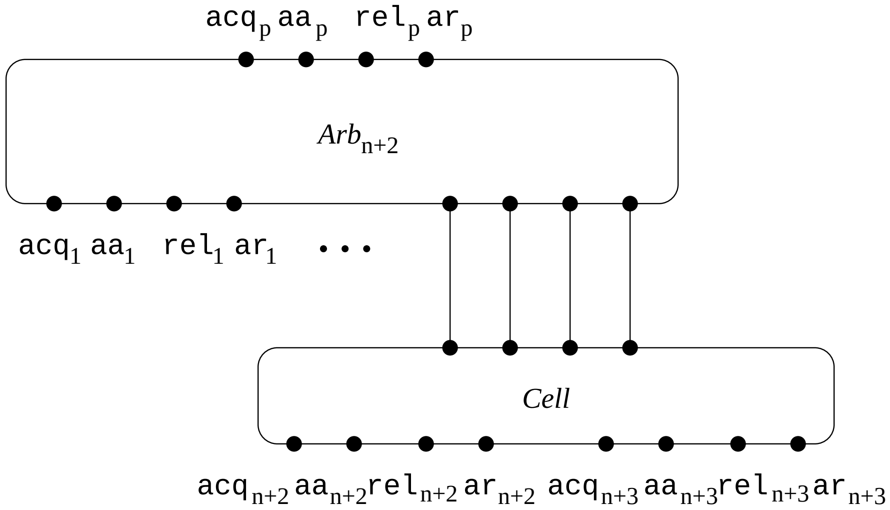


Figure 5–4: Flow Diagram for an $(n+3)$-user Arbiter

Let K be $\{\mathtt{aa}_i, \mathtt{aa}_j, \mathtt{rel}_i, \mathtt{rel}_j\}$. The following $\mu\mathbb{L}(K)$ formula expresses that user i and user j cannot simultaneously have access to the resource.

$$\begin{aligned} ME_{ij} \stackrel{\text{def}}{=} \nu X.&[\![- \mathtt{aa}_i, \mathtt{aa}_j]\!]_K X \wedge \\ &[\![\mathtt{aa}_i]\!]_K(\nu Y_i.[\![- \mathtt{rel}_i, \mathtt{aa}_j]\!]_K Y_i \wedge [\![\mathtt{rel}_i]\!]_K X \wedge [\![\mathtt{aa}_j]\!]_K \mathtt{ff}) \wedge \\ &[\![\mathtt{aa}_j]\!]_K(\nu Y_j.[\![- \mathtt{rel}_j, \mathtt{aa}_i]\!]_K Y_j \wedge [\![\mathtt{rel}_j]\!]_K X \wedge [\![\mathtt{aa}_i]\!]_K \mathtt{ff}) \end{aligned}$$

Formula ME_{ij} says that users i and j must not both be in the state in which they have acquired (and received acknowledgement for) the resource but have not yet released it. The subformula of form $\nu Y_i.\phi$ expresses the property that should hold after user i has acquired the resource. It says that user j cannot acquire the resource until after user i has released it.

Our goal is to prove that

$$Arb_n(p, 1, \ldots, n) \models ME_{ij}$$

for $n \geq 2$ and $1 \leq i < j \leq n$. Consider how the Interface Rule could be applied directly to this problem. We would write $Arb_n(p, 1, \ldots, n)$ in the form

$$E \mid F$$

where F contains only the cells that serve users i and j. Letting L be all user actions but those of users i and j, we would then show

$$E \backslash\backslash L \approx E'$$

where E' is a simpler process than $E \backslash\backslash L$. Finally we would show

$$E' \mid F \models ME_{ij}.$$

The problem with this application of the Interface Rule is that there is no natural way to simplify $E \backslash\backslash L$. However, there are elegant equations that can be used to simplify the composition of a plain cell and a cell with the actions of one user hidden. Our approach to verify the example is therefore to show, first, that

$$Arb_n(p, 1, \ldots, n) \backslash\backslash L \approx Arb_2(p, i, j)$$

where L is again the set of all user actions but those of users i and j. This is done by algebraic manipulation. Then we show

$$Arb_2(p, i, j) \models ME_{ij}.$$

Thus, showing mutual exclusion for users i and j can be done for an arbiter of size $n \geq 2$ by showing it for an arbiter of size 2.

We now do the steps of the verification in detail. Let set L be $\bigcup_{k \notin \{i,j\}} L_k$. By condition 5.6 of the Weak CCS Interface Rule, and since $ME_{ij} \in \mu I\!\!L(K)$ and $\mathcal{L}(ME_{ij}) \subseteq K$,

$$Arb_n(p, 1, \ldots, n) \models ME_{ij} \Leftrightarrow Arb_n(p, 1, \ldots, n)\backslash\!\!\backslash L \models ME_{ij}.$$

We now show that $Arb_n(p, 1, \ldots, n)\backslash\!\!\backslash L$ is observation equivalent to $Arb_2(p, i, j)$.

Proposition 5.7 *Suppose that p, i, j, k, and x are all distinct. Then the following equations hold.*

$$\begin{aligned}
(Cell(p,i,x) \mid Cell(x,j,k))\backslash L_x \backslash\!\!\backslash L_i &\approx Cell(p,j,k) \\
(Cell(p,i,x) \mid Cell(x,j,k))\backslash L_x \backslash\!\!\backslash L_j &\approx Cell(p,i,k) \\
(Cell(p,i,x) \mid Cell(x,j,k))\backslash L_x \backslash\!\!\backslash L_k &\approx Cell(p,i,j)
\end{aligned}$$

Proof. The three equations are proved in a similar way, so we show prove only the first. Let p, i, j, k,and x be any distinct values such that none of them lie between 0 to 4. Let f_{ij} be the relabelling function that maps α_i to α_j and α_j to α_i for α in $\{\texttt{acq}, \texttt{aa}, \texttt{rel}, \texttt{ar}\}$ and is the identity elsewhere. Let f be $f_{0p} \circ f_{1i}$, let g be f_{2x}, and let h be $f_{3j} \circ f_{4k}$.

The Concurrency Workbench [28] verifies that

$$(Cell(0,1,2) \mid Cell(2,3,4))\backslash L_2 \backslash\!\!\backslash L_4 \approx Cell(0,1,3).$$

Observation equivalence preserves relabelling, so

$$(Cell(0,1,2) \mid Cell(2,3,4))\backslash L_2 \backslash\!\!\backslash L_4[f][g][h] \approx Cell(0,1,3)[f][g][h].$$

By moving the relabelling inwards we get

$$(Cell(0,1,2)[f][g] \mid Cell(2,3,4)[g][h])\backslash L_x \backslash\!\!\backslash L_k \approx Cell(p,i,j).$$

Moving the relabelling inwards further we get

$$(Cell(p,i,x) \mid Cell(x,j,k)[g][h])\backslash L_x \backslash\!\!\backslash L_k \approx Cell(p,i,j).$$

□

Proposition 5.8 *Suppose $n \geq 3$ and $1 \leq i \leq n$. Then*

$$Arb_n(p, 1, \ldots, n)\backslash\!\!\backslash L_i \approx Arb_{n-1}(p, 1, \ldots, i-1, i+1, \ldots, n).$$

Proof. The proof is by induction on n. In the base case $n = 3$. There are three subcases for the possible values of i. Suppose $i = 1$. Then

$$(Cell(p, 1, x) \mid Cell(x, 2, 3))\backslash L_2\backslash\!\!\backslash L_1 \approx Cell(p, 2, 3)$$

by the first equation of Prop. 5.7 and

$$Arb_3(p, 1, 2, 3)\backslash\!\!\backslash L_1 \approx Arb_2(p, 2, 3)$$

by the definition of $Arb_2(p, x, y)$. The cases for $i = 2$ and $i = 3$ are the same except that they use the second and third equations of Prop. 5.7.

In the induction case $n > 3$. There are four subcases for the possible values of i. But first note that by the definition of Arb_n we have

$$\begin{aligned}
& Arb_n(p, 1, \ldots, n) \\
\approx\ & (Arb_{n-1}(p, 1, \ldots, n-2, x) \mid Cell(x, n-1, n))\backslash L_x \\
\approx\ & ((Arb_{n-2}(p, 1, \ldots, n-3, y) \mid Cell(y, n-2, x))\backslash L_y \mid Cell(x, n-1, n))\backslash L_x \\
\approx\ & (Arb_{n-2}(p, 1, \ldots, n-3, y) \mid (Cell(y, n-2, x) \mid Cell(x, n-1, n))\backslash L_x)\backslash L_y
\end{aligned}$$

Now suppose $1 \leq i < n-2$. Then using the induction hypothesis we have

$$\begin{aligned}
& (Arb_n(p, 1, \ldots, n-2, x) \mid Cell(x, n-1, n))\backslash L_x\backslash\!\!\backslash L_i \\
\approx\ & (Arb_{n-1}(p, 1, \ldots, n-2, x)\backslash\!\!\backslash L_i \mid Cell(x, n-1, n))\backslash L_x \\
\approx\ & (Arb_{n-2}(p, 1, \ldots, i-1, i+1, \ldots, n-2, x) \mid Cell(x, n-1, n))\backslash L_x \\
\approx\ & Arb_{n-1}(p, 1, \ldots, i-1, i+1, \ldots, n)
\end{aligned}$$

Next suppose $i = n - 2$. Then by the first equation of Proposition 5.7

$$\begin{array}{rl} & (Arb_{n-2}(p, 1, \ldots, i-1, y) \mid (Cell(y, i, x) \mid Cell(x, i+1, n))\backslash L_x)\backslash L_y \backslash\!\backslash L_i \\ \approx & (Arb_{n-2}(p, 1, \ldots, i-1, y) \mid (Cell(y, i, x) \mid Cell(x, i+1, n))\backslash L_x \backslash\!\backslash L_i)\backslash L_y \\ \approx & (Arb_{n-2}(p, 1, \ldots, i-1, y) \mid Cell(y, i+1, n))\backslash L_y \\ \approx & Arb_{n-1}(p, 1, \ldots, i-1, i+1, n) \end{array}$$

The cases for $i = n - 1$ and $i = n$ are similar except that they use the second and third equations of Prop. 5.7. □

Theorem 5.3 *Suppose* $n \geq 2$ *and* $p \notin \{1, \ldots, n\}$ *and* $1 \leq i < j \leq n$. *Then*

$$Arb_2(p, i, j) \models ME_{ij} \Rightarrow Arb_n(p, 1, \ldots, n) \models ME_{ij}.$$

Proof. Assume that

$$Arb_2(p, i, j) \models ME_{ij}.$$

By repeated application of Proposition 5.8 we then obtain, for $L \stackrel{\text{def}}{=} \bigcup\{L_k \mid k \in (\{1, \ldots, n\} - \{i, j\})\}$, that

$$Arb_n(p, 1, \ldots, n)\backslash\!\backslash L \models ME_{ij},$$

and then by condition 5.6 of the Weak CCS Interface Rule

$$Arb_n(p, 1, \ldots, n) \models ME_{ij}.$$

□

5.3 Another Abstraction Principle for Hiding

The most interesting part of developing an interface rule for CCS is defining a logic $\mathcal{L}(\Sigma)$ that satisfies condition 5.6. This condition is useful on its own as an abstraction principle for hiding. In the Weak CCS Interface Rule, condition 5.6 states that, for all complementation-closed sets K of labels and all ϕ in $\mu\mathbb{L}(K)$,

$$\text{if } \mathcal{L}(\phi) \subseteq K \text{ then } (E\backslash\!\backslash(\mathcal{L} - K) \models \phi \Leftrightarrow E \models \phi).$$

In other words, when using a $\mu\mathbb{L}(K)$ formula having a sort contained in K, all actions not in K can be hidden in a process without affecting whether the process satisfies the formula. We call a formula that satisfies this condition an *explicit* formula. For the purpose of process abstraction with hiding, the condition is stronger than necessary. We require only that

$$\text{if } \mathcal{L}(\phi) \subseteq K \text{ then } (E\backslash\!\backslash(\mathcal{L} - K) \models \phi \Rightarrow E \models \phi).$$

Indeed, if condition 5.6 is changed in this way the Interface Rule is still sound.

Here we define a different logic and show that it supports a similar abstraction principle. The logic consists of the formulas of $\mu\mathbb{L}\Box$ without ε in their sorts. Suppose ϕ is such a formula with sort L. We show that if ϕ holds of a process in which non-L actions are hidden, then ϕ holds of the process itself. We first present some facts about the strong and weak box operators that are simple to prove.

Proposition 5.9 *For all $K, L \subseteq Act$ and $M, N \subseteq \mathcal{L} \cup \{\varepsilon\}$,*

1. $[K \cup L]\phi \Leftrightarrow [K]\phi \wedge [L]\phi$

2. $[\![M \cup N]\!]\phi \Leftrightarrow [\![M]\!]\phi \wedge [\![N]\!]\phi$

3. $[\![M]\!](\phi \wedge \psi) \Leftrightarrow [\![M]\!]\phi \wedge [\![M]\!]\psi$

We also need some facts about $\mathcal{W}$.

Proposition 5.10 *For all $K \subseteq Act$,*

1. $\mathcal{W}_{\setminus\setminus L}([\![\varepsilon]\!]\phi) \Rightarrow [\![\varepsilon]\!]\mathcal{W}_{\setminus\setminus L}(\phi)$

2. $\mathcal{W}_{\setminus\setminus L}([\![\varepsilon]\!][K][\![\varepsilon]\!]\phi) \Rightarrow \begin{cases} [\![\varepsilon]\!][K \cup L][\![\varepsilon]\!]\mathcal{W}_{\setminus\setminus L}(\phi) & \textit{if } \tau \in K \\ [\![\varepsilon]\!][K - L][\![\varepsilon]\!]\mathcal{W}_{\setminus\setminus L}(\phi) & \textit{otherwise} \end{cases}$

Proof.

1.
$$\begin{array}{cll} & \mathcal{W}_{\setminus\setminus L}([\![\varepsilon]\!]\phi) & \\ \Leftrightarrow & \mathcal{W}_{\setminus\setminus L}(\nu X.\phi \wedge [\tau]X) & \text{by def. of } [\![\varepsilon]\!] \\ \Leftrightarrow & \nu X.\mathcal{W}_{\setminus\setminus L}(\phi) \wedge [L \cup \{\tau\}]X & \text{by def. of } \mathcal{W}_{\setminus\setminus L} \text{ and Prop. 4.4} \\ \Rightarrow & \nu X.\mathcal{W}_{\setminus\setminus L}(\phi) \wedge [\tau]X & \text{by Prop. 5.9.1} \\ \Leftrightarrow & [\![\varepsilon]\!]\mathcal{W}_{\setminus\setminus L}(\phi) & \text{by def. of } [\![\varepsilon]\!] \end{array}$$

2. Suppose $\tau \in K$. Then

$$\begin{array}{cll} & \mathcal{W}_{\setminus\setminus L}([\![\varepsilon]\!][K][\![\varepsilon]\!]\psi) & \\ \Rightarrow & [\![\varepsilon]\!]\mathcal{W}_{\setminus\setminus L}([K][\![\varepsilon]\!]\psi) & \text{by Prop. 5.10.1} \\ \Leftrightarrow & [\![\varepsilon]\!][K \cup L]\mathcal{W}_{\setminus\setminus L}([\![\varepsilon]\!]\psi) & \text{by def. of } \mathcal{W}_{\setminus\setminus L} \text{ and Prop. 4.4} \\ \Rightarrow & [\![\varepsilon]\!][K \cup L][\![\varepsilon]\!]\mathcal{W}_{\setminus\setminus L}(\psi) & \text{by Prop. 5.10.1} \end{array}$$

The case for $\tau \notin K$ is similar. $\square$

Proposition 5.11 *Let L be a set of labels and ϕ be a $\mu\mathbb{L}\Box$ formula such that $\mathcal{L}(\phi) \cap (L \cup \{\varepsilon\}) = \emptyset$. Then*

$$E \setminus\setminus L \models \phi \Rightarrow E \models \phi.$$

Proof. By the Correspondence Rule the left-hand side is equivalent to $E \models \mathcal{W}_{\setminus\setminus L}(\phi)$, so it is enough to prove that

$$\mathcal{W}_{\setminus\setminus L}(\phi) \Rightarrow \phi.$$

We do so by induction on the structure of ϕ. All cases are routine except for the case of the weak box operator. There are two subcases to consider.

1. Suppose ϕ is of the form $[\![K]\!]\psi$. Note that $K \subseteq \mathcal{L}(\phi)$ so $K \cap L = \emptyset$. Then

$$\begin{array}{lll} & \mathcal{W}_{\setminus\setminus L}([\![K]\!]\psi) & \\ \Leftrightarrow & \mathcal{W}_{\setminus\setminus L}([\![\varepsilon]\!][K][\![\varepsilon]\!]\psi) & \text{by def. of } [\![K]\!] \\ \Rightarrow & [\![\varepsilon]\!][K-L][\![\varepsilon]\!]\mathcal{W}_{\setminus\setminus L}(\psi) & \text{by Prop. 5.10.2} \\ \Rightarrow & [\![\varepsilon]\!][K][\![\varepsilon]\!]\mathcal{W}_{\setminus\setminus L}(\psi) & \text{since } K \cap L = \emptyset \\ \Leftrightarrow & [\![K]\!]\mathcal{W}_{\setminus\setminus L}(\psi) & \text{by def. of } [\![K]\!] \\ \Rightarrow & [\![K]\!]\psi & \text{by the induction hypothesis} \end{array}$$

2. Suppose ϕ is of the form $[\![-K]\!]\psi$. Note that $K \subseteq \mathcal{L}(\phi)$ so $\varepsilon \notin K$. Then

$$\begin{array}{lll} & [\![-K]\!]\psi & \\ \Leftrightarrow & [\![(\mathcal{L} \cup \{\varepsilon\}) - K]\!]\psi \wedge [\![\varepsilon]\!]\psi & \text{by def. of } [\![-K]\!] \\ \Leftrightarrow & [\![\varepsilon]\!][\mathcal{L}-K][\![\varepsilon]\!]\psi \wedge [\![\varepsilon]\!]\psi & \text{by Prop. 3.4} \\ \Rightarrow & [\![\varepsilon]\!][\mathcal{L}-K][\![\varepsilon]\!]\psi \wedge [\![\varepsilon]\!]\psi \wedge [\![\varepsilon]\!][\tau][\![\varepsilon]\!]\psi & \text{since } [\![\varepsilon]\!]\psi \Rightarrow [\![\varepsilon]\!][\tau][\![\varepsilon]\!]\psi \\ \Leftrightarrow & [\![\varepsilon]\!][Act-K][\![\varepsilon]\!]\psi \wedge [\![\varepsilon]\!]\psi & \text{by Prop. 5.9.1 and 5.9.3.} \end{array}$$

   So $\mathcal{W}_{\setminus\setminus L}([\![-K]\!]\psi)$ is

$$\begin{array}{lll} & \mathcal{W}_{\setminus\setminus L}([\![\varepsilon]\!][Act-K][\![\varepsilon]\!]\psi \wedge [\![\varepsilon]\!]\psi) & \\ \Leftrightarrow & \mathcal{W}_{\setminus\setminus L}([\![\varepsilon]\!][Act-K][\![\varepsilon]\!]\psi) \wedge \mathcal{W}_{\setminus\setminus L}([\![\varepsilon]\!]\psi) & \text{by def. of } \mathcal{W}_{\setminus\setminus L} \end{array}$$

   For the first conjunct we have

$$\begin{array}{lll} & \mathcal{W}_{\setminus\setminus L}([\![\varepsilon]\!][Act-K][\![\varepsilon]\!]\psi) & \\ \Rightarrow & [\![\varepsilon]\!][(Act-K) \cup L][\![\varepsilon]\!]\mathcal{W}_{\setminus\setminus L}(\psi) & \text{by Prop. 5.10.2} \\ \Rightarrow & [\![\varepsilon]\!][\mathcal{L}-K][\![\varepsilon]\!]\mathcal{W}_{\setminus\setminus L}(\psi) & \text{by Prop. 5.9.1} \\ \Leftrightarrow & [\![-(K \cup \{\varepsilon\})]\!]\mathcal{W}_{\setminus\setminus L}(\psi) & \text{by def. of } [\![\ ]\!] \\ \Rightarrow & [\![-(K \cup \{\varepsilon\})]\!]\psi & \text{by the induction hypothesis} \end{array}$$

   For the second conjunct we have

$$\mathcal{W}_{\setminus\setminus L}([\![\varepsilon]\!]\psi) \Rightarrow [\![\varepsilon]\!]\mathcal{W}_{\setminus\setminus L}(\psi) \Rightarrow [\![\varepsilon]\!]\psi$$

by Prop. 5.10.1 and the induction hypothesis. Therefore $\mathcal{W}_{\setminus L}([\![-K]\!]\psi)$ implies $[\![-(K \cup \{\varepsilon\})]\!]\psi \wedge [\![\varepsilon]\!]\psi$, which is equivalent to $[\![-K]\!]\psi$ by Prop. 5.9.2. □

5.4 Data Independence

In [97] Wolper shows that temporal properties involving large or infinite sets of actions can be expressed using small sets of actions if one assumes that the process to be verified is data-independent. This result can be regarded as an application of the Correspondence Rule to a class of value-passing processes. In this section we generalize the notion of data independence from the linear-time case to the branching-time case, and prove the theorems of [97] using the Correspondence Rule.

Informally, data-independent processes are value-passing processes whose behaviour does not depend on the specific data values they operate upon. An example data-independent process is the simple buffer B.

$$B \stackrel{\text{def}}{=} \mathtt{in}(x).\,\overline{\mathtt{out}}(x).B$$

The arbitrary nature of values in a data-independent process means that if a property is proved of such a process for a particular data domain, then it also holds for all other data domains. An important application is in showing that an infinite-state process has a property because a finite-state version does.

Recall from Section 2.4 that (E, D) stands for the basic process obtained from value-passing process E by letting the parameter of a label l in E range over values in $D(l)$. In this section we require that the parameters of all labels in the sort of a process range over the same set. Therefore we use (E, A) as shorthand for (E, D), where D maps the parameters of all labels in E to set A. For example, (B, V) is the buffer B that takes its inputs from value set V. As we define value-passing processes by translation to basic CCS, we can regard process (E, D) as a basic CCS process, to which all basic CCS operators can be applied. The sort $\mathcal{L}(E, D)$ of a process

(E, D) contains only actions of the form $l(v)$ or τ, where v is a data value in D. For a μL formula on value-passing processes, the *data sort* is the set of values explicitly mentioned in the formula:

$$\mathcal{V}(\phi) \stackrel{\text{def}}{=} \{v \mid l(v) \in \mathcal{L}(\phi)\}$$

For example, the data sort of $\nu Z.[-b(1)]\,\mathtt{ff} \wedge [a(1)]Z$ is $\{1\}$.

A mapping on a data domain induces a relabelling function on value-passing processes. If a data mapping maps v to v', then the induced relabelling function maps $a(v)$ to $a(v')$.

Definition 5.3 *Let $f : D \to D'$ be a data mapping. Then the induced relabelling function f_r is:*

$$f_r(\alpha) \stackrel{\text{def}}{=} \begin{cases} l(f(v)) & \text{if } \alpha = l(v) \\ \tau & \text{otherwise} \end{cases}$$

Given a data mapping f, we write $E[f]$ to mean $E[f_r]$.

Wolper defines the property of data independence on the traces of value-passing processes. Such traces consist of actions parametrised by values. A process (E, D) is data independent if its set of traces, transformed by applying a mapping f to every value of every action, is just the set of traces of $(E, f(D))$. We define data independence on the branching behaviour of processes using strong equivalence. As in earlier chapters, if $f : A \to B$ is a function and $X \subseteq A$ then we write $f(X)$ for $\{f(x) \mid x \in X\}$. Also, if $Y \subseteq B$ then we write $f^{-1}(Y)$ for $\{x \mid f(x) \in Y\}$.

Definition 5.4 *A value-passing process E is* data-independent *if, for every data mapping $f : D \to D'$,*

$$(E, f(D)) \sim (E, D)[f].$$

It is easy to see that buffer process B is data-independent. By the definition of value-passing CCS we know $(B, f(D)) \sim B_1$, where $B_1 \stackrel{\text{def}}{=} \sum_{v \in f(D)} \mathtt{in}(v).\,\overline{\mathtt{out}}(v).B_1$, and $(B,D)[f] \sim B_2[f]$, where $B_2 \stackrel{\text{def}}{=} \sum_{v\in D} \mathtt{in}(v).\overline{\mathtt{out}}(v).B_2$. By moving the relabelling into B_2 and using the definition of $f(D)$ we have that $B_1 \sim B_2[f]$. Wolper mentions in [97] that showing a particular process to be data-independent is often easy, although the general problem of checking data-independence is undecidable.

We apply data mappings to μL formulas on value-passing processes similarly to the way functions are applied to formulas in Section 4.4. If ϕ is a μL formula then $f(\phi)$ is the formula obtained by replacing each modal operator $[L]$ of ϕ by $[f_r(L)]$. By Prop. 4.4 and the def. of $f^{-1}(\phi)$ we know formula $\mathcal{W}_{[f]}(\phi)$ is equivalent to $f^{-1}(\phi)$.

We now state Wolper's results on data-independent processes and prove them using the Correspondence Rule. Proposition 5.1 of [97] is a special case of the Correspondence Rule for surjective data mappings on data-independent processes. It states that, in some circumstances, a property holds of a process on some data domain just if a related property holds of a process on a smaller data domain.

Proposition 5.12 *If E is a data-independent process and $f : D \to D'$ is a surjection, then*

$$(E,D) \models f^{-1}(\phi) \Leftrightarrow (E,D') \models \phi.$$

Proof.

$$\begin{array}{rlll}
 & (E,D) & \models & f^{-1}(\phi) & \\
\Leftrightarrow & (E,D) & \models & \mathcal{W}_{[f]}(\phi) & \text{by Prop. 4.4 and the def. of } f^{-1}(\phi) \\
\Leftrightarrow & (E,D)[f] & \models & \phi & \text{by the Correspondence Rule} \\
\Leftrightarrow & (E,f(D)) & \models & \phi & \text{by data-independence and Theorem 3.2} \\
\Leftrightarrow & (E,D') & \models & \phi & \text{since } f \text{ is a surjection}
\end{array}$$

□

Suppose we want to prove that buffer process $(B,\mathbb{N})$ can initially input some value. Using Proposition 5.12, with f the surjection that maps all natural numbers to 0, we know that $(B,\mathbb{N})$ satisfies $\langle \mathtt{in}(0), \mathtt{in}(1), \ldots\rangle\, \mathtt{tt}$ just if $(B,\{0\})$ satisfies $\langle \mathtt{in}(0)\rangle\, \mathtt{tt}$.

Proposition 5.2 of [97] states that if f is a mapping that does not confuse values mentioned in a property, then the property holds of a process on a domain just if the property transformed by f holds of the process on the domain transformed by f. We define a mapping $f : D \to D'$ to be an *injection over* $D_0 \subseteq D$ if f restricted to D_0 is an injection and if $f^{-1}(f(D_0)) \subseteq D_0$.

Proposition 5.13 *If $f : D \to D'$ is an injection over $\mathcal{V}(\phi)$ then $[f]$ is safe for ϕ.*

Proof. Suppose $f : D \to D'$ is an injection over $\mathcal{V}(\phi)$. By Proposition 4.7, to show that $[f]$ is safe for ϕ it is enough to show that f_r is an injection over $\mathcal{L}(\phi)$. First we show that f_r restricted to $\mathcal{L}(\phi)$ is an injection. Suppose $a(v)$ and $b(v')$ are in $\mathcal{L}(\phi)$ and that $f_r(a(v)) = f_r(b(v'))$. Then $a = b$ and $f(v) = f(v')$. Since f is an injection over $\mathcal{V}(\phi)$ we know that $v = v'$, and therefore $a(v) = b(v')$.

Secondly we show that $f_r^{-1}(f_r(\mathcal{L}(\phi))) \subseteq \mathcal{L}(\phi)$. Suppose $a(v) \in \mathcal{L}(\phi)$. Then $v \in \mathcal{V}(\phi)$ by def. of $\mathcal{V}(\phi)$, $f_r^{-1}(f_r(a(v))) = a(f^{-1}(f(v)))$ by def. of f_r, and $a(f^{-1}(f(v))) = a(v)$ because f is an injection over $\mathcal{V}(\phi)$. □

Proposition 5.14 *If E is a data-independent process and $f : D \to D'$ is an injection over $\mathcal{V}(\phi)$, then*

$$(E, D) \models \phi \Leftrightarrow (E, f(D)) \models f(\phi).$$

Proof.

$$\begin{array}{llll} & (E,D) & \models & \phi \\ \Leftrightarrow & (E,D)[f] & \models & f(\phi) \quad \text{by Props. 4.7 and 5.13} \\ \Leftrightarrow & (E,f(D)) & \models & f(\phi) \quad \text{by data independence and Theorem 3.2} \end{array}$$

□

Suppose we want to prove of the buffer process B that after value 1 is input, value 2 cannot be immediately output. This theorem tells us that $(B, \mathbb{N})$ satisfies $[\mathtt{in}(1)][\overline{\mathtt{out}}(2)]\,\mathtt{ff}$ just when $(B, \{1, 2, 3\})$ satisfies $[\mathtt{in}(1)]\,[\overline{\mathtt{out}}(2)]\,\mathtt{ff}$, because we can map all values of $I\!N$ that are neither 1 nor 2 to 3 by a mapping that is an injection over $\{1, 2\}$.

Recall that a permutation of a set D is a bijection from D to itself, and let $\mathrm{Per}(D)$ be the permutations of D. Corollary 5.3 of [97] states that a data-independent process over data domain D satisfies a property ϕ just if it satisfies all properties $f(\phi)$, where f is a permutation of D. To express the result formally we must extend μL to allow infinitary conjunction. We write $\bigwedge_{i \in I} \phi_i$ for possibly infinite conjunction, defined by $E \models \bigwedge_{i \in I} \phi_i \stackrel{\text{def}}{=} E \models \phi_i$ for all $i \in I$.

Proposition 5.15 *If E is a data-independent process, then*

$$(E, D) \models \phi \Rightarrow (E, D) \models \bigwedge_{f \in \mathrm{Per}(D)} f(\phi)$$

Proof. Suppose $(E, D) \models \phi$. For an arbitrary permutation f of D we have by Theorem 5.14 that $(E, f(D)) \models f(\phi)$, and since $f(D) = D$ we know $(E, D) \models f(\phi)$. Therefore

$$\bigwedge_{f \in \mathrm{Per}(D)} (E, D) \models f(\phi)$$

and the result follows by the definition of infinitary conjunction. $\square$

This result allows us to prove of the buffer process that if $(B, I\!N)$ satisfies $\langle \mathtt{in}(1) \rangle\, \mathtt{tt}$, then it also satisfies

$$\bigwedge_{i \in I\!N} \langle \mathtt{in}(i) \rangle\, \mathtt{tt}$$

because for every i we can find a permutation of $I\!N$ that maps 1 to i. This result is stronger than that of our first example, where we showed only that the buffer must be able to input *some* value. We can also prove that if $(B, I\!N)$ satisfies $[\mathtt{in}(1)][\overline{\mathtt{out}}(2)]\, \mathtt{ff}$, then it satisfies

$$\bigwedge_{(i,j) \in (I\!N \times I\!N) - I} [\mathtt{in}(i)][\overline{\mathtt{out}}(j)]\, \mathtt{ff}$$

where I is the identity relation on natural numbers. Here we use every permutation of $I\!N$ that maps 1 and 2 to a pair of distinct naturals.

Theorem 5.4 of [97] states that a property can be shown to hold of all finite subsets of a data domain by proving that it holds of a single subset.

Proposition 5.16 *If E is a data-independent process and $f : D \to D'$ is an injection over $\mathcal{V}(\phi)$, then*

$$(E, f(D)) \models f(\phi) \Rightarrow (E, D) \models \bigwedge_{g \in \mathrm{Per}(D)} g(\phi).$$

Proof. The left-hand side of the implication is equivalent to $(E, D) \models \phi$ by Proposition 5.14, and the result then follows directly by Proposition 5.15. $\square$

Thus, if $(B, \{1, 2, 3\})$ satisfies $[\mathtt{in}(1)][\overline{\mathtt{out}}(2)]\,\mathtt{ff}$, then $(B, I\!N)$ satisfies

$$\bigwedge_{(i,j) \in (I\!N \times I\!N) - I} [\mathtt{in}(i)][\overline{\mathtt{out}}(j)]\,\mathtt{ff}\,.$$

In other words, we can prove that the buffer cannot input one value and then output a different value, for any values in $I\!N$, by showing that the buffer operating on values 1,2, and 3 cannot input 1 and then output 2.

5.5 Discussion

The main purpose of this chapter is to suggest that the Correspondence Rule is a basic verification idea by showing how two other well-known verification ideas can be derived from it. The Interface Rule [22] was chosen for study because it was proposed as a general technique for compositional model checking. The results on data-independent processes were chosen for study because data-indendent, value-passing processes are commonly used to model systems such as protocols (e.g., the sliding-window protocol modelled in [72]).

Some of the results in this chapter are of independent interest. In [22] it is claimed that the Interface Rule holds for CCS and HML. We showed in Section 5.2.1 that it does not generally hold for HML. It is also claimed in [22] that the Interface Rule can be used to prove that a CSP-like model of an arbiter satisfies a safety property. We showed in Section 5.2.3 the Interface Rule is not suitable for verifying the arbiter's safety property. However, we verify the property using the Interface Rule and algebraic properties of the arbiter. This suggests that the flexibility of the Correspondence Rule is important in applications.

Studying the Interface Rule also led to the notion of an *explicit* μL formula, which is a formula ϕ that is satisfied by a process E just if it is satisfied by $E \backslash\backslash L$, where labels in L are not in $\mathcal{L}(\phi)$ (or its complement). Equivalently, if processes $E \backslash\backslash L$ and $F \backslash\backslash L$ are bisimilar and the labels in L are not in $\mathcal{L}(\phi)$ (or its complement), then E satisfies explicit formula ϕ just if F does. In Section 5.2.1 we showed that μL formulas not containing τ are explicit formulas. In Section 5.2.2 we showed that $\mu \mathbb{L}(\mathcal{L}(\phi))$ formulas are explicit formulas, where $\mu \mathbb{L}(K)$ is like $\mu \mathbb{L}$ except that in $\mu \mathbb{L}(K)$ the box operator treats non-K actions like τ. Section 5.3 describes a class of formulas that satisfy a slightly weaker condition than that satisfied by explicit formulas.

We have proved slightly different results than Wolper on data independent processes. Wolper defines data independence using process traces, while we define it using strong equivalence. Wolper expresses properties of processes as formulas of propositional, linear-time temporal logic, while we we use μL formulas. We have also proved our results differently than Wolper. Our proofs of Propositions 5.12 and 5.14 use mainly the Correspondence Rule and the principle of data independence, and do not use induction. Wolper, lacking a general principle like the Correspondence Rule, used induction on the structure of logical formulas to prove his results.

Chapter 6

Abstraction with Process Preorders

In this chapter we examine how processes can be abstracted with process preorders. Here algebraic laws for the weakly simulates relation are applied to a process to obtain an abstract version of the process. Results of previous chapters guarantee that every property of the abstract process is also a property of the concrete process, at least for properties expressed in the diamond-free mu-calculus. The most important algebraic laws we present allow the merging of states and the loosening of synchrony. We demonstrate the technique using Milner's scheduler example.

6.1 Abstraction with Preorders

The basic theory of CCS allows a kind of process abstraction. For example, using the laws for observation congruence we can simplify process $a.\tau.0 + (b.0)[a/b]$ as follows:

$$
\begin{array}{lll}
 & a.\tau.0 + (b.0)[a/b] & \\
= & a.0 + (b.0)[a/b] & \text{since } \alpha.\tau.E = \alpha.E \\
= & a.0 + a.0 & \text{since } (\alpha.E)[f] = f(\alpha).E[f] \text{ and } 0[f] = 0 \\
= & a.0 & \text{since } E + E = E
\end{array}
$$

Abstraction here is the application of an algebraic law that simplifies the process term. Two observation congruent processes satisfy the same set of $\mu\mathbb{L}$ formulas, so if $a.0$ satisfies such a formula then so does $a.\tau.0 + (b.0)[a/b]$.

However, observation congruence is very limited as a tool for abstraction. The relation is so strong that many desirable abstraction operations cannot be used. For example, we cannot freely remove τ actions, since $\tau.E \neq E$. Also, observation congruence is symmetrical, so an abstract and concrete process will satisfy the same set of $\mu\mathbb{L}$ formulas. We require only that a formula satisfied by an abstract process is also satisfied by the concrete process, not vice versa.

In this chapter we explore the use of the weakly simulates relation (written $\ll$) in abstraction. Most of the results that we need for abstraction with $\ll$ have already been presented. First, $\ll$ is a precongruence (Prop. 2.15), so abstraction can be performed by applying algebraic laws. Secondly, $\ll$ *preserves* formulas of $\mu\mathbb{L}\Box$ (Prop. 4.15), for if F satisfies a formula ϕ of $\mu\mathbb{L}\Box$, and $E \ll F$, then E satisfies ϕ. All that is needed are algebraic laws for $\ll$ that can be used as abstraction operations.

The two main abstraction operations defined in this chapter allow the "merging" of process constants, the syntactic analogue of the merging of states, and the weakening of synchrony between process components through hiding and relabelling. Also defined are operations for the removal of τ actions, and for the introduction and elimination of process constants.

The chapter is organised as follows. In the following section we define some operations on families of definitions and show how families can be abstracted. Algebraic laws for $\ll$ are then presented. Then we define some derived laws and show how Milner's scheduler process can be abstracted.

6.2 Abstracting Families of Definitions

In the example above algebraic laws were used to abstract a simple process expression. More complex processes are usually described using a family of process definitions. Some of the abstraction operations we use are defined not on simple process expressions, but on families themselves. In this section we define basic operations on families that will be used later to define abstraction operations.

In what follows we write $E\{F_i/A_i \mid i \in I\}$ for the process obtained by simultaneously replacing every A_i in E by F_i.

Definition 6.1 *Let* $\mathcal{F} = \{A_i \stackrel{\text{def}}{=} E_i \mid i \in I\}$ *be a family of process definitions, let* j, k *be indices in* I*, and let* $S = \{B_j \stackrel{\text{def}}{=} F_j \mid j \in J\}$ *be a determinate set of process definitions. Then*

$$\begin{aligned}
\mathcal{F}\{A/B\} &\stackrel{\text{def}}{=} \{A_i\{A/B\} \stackrel{\text{def}}{=} E_i\{A/B\} \mid i \in I\} \\
\mathcal{F}S &\stackrel{\text{def}}{=} \{A_i \stackrel{\text{def}}{=} E_i \mid i \in I \text{ and } \forall j \in J.A_i \not\equiv B_j\} \cup S \\
\mathrm{merge}(A_j, A_k, \mathcal{F}) &\stackrel{\text{def}}{=} (\mathcal{F}\{A_j \stackrel{\text{def}}{=} E_j + E_k\}\{A_k \stackrel{\text{def}}{=} E_j + E_k\})\{A_j/A_k\}
\end{aligned}$$

The first operation is the extension of substitution for one constant by another from processes to families of definitions. Notice that the substitution is applied not only in the bodies of the definitions but also to the constants being defined.

The second operation is the *updating* of $\mathcal{F}$ by S, in which constant definitions in S are added to, or replace, the definitions in $\mathcal{F}$. Notice that S need not be a family and that updating $\mathcal{F}$ by S may not result in a family. Updating $\{A \stackrel{\text{def}}{=} a.\tau.A\}$ by $\{A \stackrel{\text{def}}{=} a.A\}$ yields $\{A \stackrel{\text{def}}{=} a.A\}$.

The third operation is the *merging* of two constants in a family. The definition of the first of the two given constants is extended by adding the definition of the second as a choice, and renaming the second constant to the first. For example, $\mathrm{merge}(A, B, \{A \stackrel{\text{def}}{=} a.B, B \stackrel{\text{def}}{=} b.A\})$ yields $\{A \stackrel{\text{def}}{=} a.A + b.A\}$.

Next we extend $\ll$ from processes to families of process definitions.

Definition 6.2 *Let* $\mathcal{F} \stackrel{\text{def}}{=} \{A_i \stackrel{\text{def}}{=} E_i \mid i \in I\}$ *and* $\mathcal{F}' \stackrel{\text{def}}{=} \{B_j \stackrel{\text{def}}{=} F_j \mid j \in J\}$ *be families of process definitions. Then* $\mathcal{F} \ll \mathcal{F}'$ *if* $(A_i \textit{ of } \mathcal{F}) \ll (B_j \textit{ of } \mathcal{F}')$ *for all* $i \in I$ *and* $j \in J$ *such that* $A_i \equiv B_j$.

A family $\mathcal{F}'$ of definitions weakly simulates a family $\mathcal{F}$ if, for all constants A common to the two families, A of $\mathcal{F}'$ weakly simulates A of $\mathcal{F}$. Since the weakly simulates relation is a precongruence (Prop. 2.15), applying a law for $\ll$ anywhere within a family $\mathcal{F}$ results in a family related to $\mathcal{F}$ by $\ll$.

To simplify the presentation of a series of abstraction steps, we often reuse constants instead of defining new ones. For example, suppose we have family

$$\{A \stackrel{\text{def}}{=} a.B, B \stackrel{\text{def}}{=} b.A\}$$

and then abstract process A by hiding action b. We write the resulting family of definitions as

$$\{A \stackrel{\text{def}}{=} a.B, B \stackrel{\text{def}}{=} \tau.A\}$$

in which the new A is strongly equivalent to the old A with action b hidden. This reuse of constants is supported by the following proposition.

Proposition 6.1 *Let* $\mathcal{F} \stackrel{\text{def}}{=} \{A_i \stackrel{\text{def}}{=} E_i \mid i \in I\}$ *be a family of definitions, let* $\{E'_i \mid i \in I\}$ *be a set of processes, and let* O *be a unary process operator that preserves* $\sim$. *If*

$$O(E_i) \sim E'_i\{O(A_i)/A_i \mid i \in I\}$$

for all i *in* I, *with* A_i $(i \in I)$ *weakly guarded in* E'_j $(j \in I)$, *then*

$$O(A_i \textit{ of } \mathcal{F}) \sim A_i \textit{ of } \{A_i \stackrel{\text{def}}{=} E'_i \mid i \in I\}$$

for all i *in* I.

Proof. We show that $O(A_i \text{ of } \mathcal{F})$ and A_i of $\{A_i \stackrel{\text{def}}{=} E_i' \mid i \in I\}$ are both solutions to equation

$$X_i \sim E_i'\{X_i/A_i \mid i \in I\}.$$

and are therefore strongly equivalent by unique solutions of equations (Prop. 2.8). First, for an A_i of $\mathcal{F}$,

$$\begin{aligned} A_i \stackrel{\text{def}}{=} E_i \quad &\Rightarrow \quad A_i \sim E_i \\ &\Rightarrow \quad O(A_i) \sim O(E_i) \\ &\Rightarrow \quad O(A_i) \sim E_i'\{O(A_i)/A_i \mid i \in I\} \end{aligned}$$

so $O(A_i)$ satisfies the equation. Secondly, for an A_i of $\{A_i \stackrel{\text{def}}{=} E_i' \mid i \in I\}$,

$$A_i \stackrel{\text{def}}{=} E_i' \Rightarrow A_i \sim E_i'$$

so A_i satisfies the equation. $\square$

6.3 Abstraction Operations

We now present algebraic laws for $\ll$ that will be used as process abstraction operations. Since $\ll$ is a weaker relation than strong equivalence, observation equivalence, and observation congruence, any equation over these equivalences can be used as an algebraic law for $\ll$. We list in this section the laws needed in examples of this and the following chapter. The laws are organised according to the outermost operator of the left-hand process.

The following laws allow restricted actions to be hidden and relabelled. They are used mainly to hide and relabel actions through which components synchronise.

Proposition 6.2 (restriction laws)

$$E\backslash L \quad \ll \quad E\backslash\!\!\backslash K\backslash L \quad \textit{if } K \subseteq L \cup \overline{L} \tag{6.1}$$

$$E\backslash L \quad \ll \quad E[f]\backslash L \quad \textit{if } f(\alpha) = \alpha \textit{ for } \alpha \notin L \cup \overline{L} \tag{6.2}$$

Proof. The proofs of the two laws are similar, so we prove only the first. We will show that

$$\{(E\backslash L, E\backslash\!\backslash K\backslash L) \mid E \in \mathcal{P}, K \subseteq L \cup \overline{L}\}$$

is a simulation. Suppose $E\backslash L \xrightarrow{\alpha} E'\backslash L$. Then $E \xrightarrow{\alpha} E'$ and $\alpha \notin L \cup \overline{L}$. So also $\alpha \notin K$ and we can infer $E\backslash\!\backslash K \xrightarrow{\alpha} E'\backslash\!\backslash K$ and $E\backslash\!\backslash K\backslash L \xrightarrow{\alpha} E'\backslash\!\backslash K\backslash L$. □

The operators introduced by applying these laws can be removed using the laws for relabelling and hiding.

Proposition 6.3 (hiding laws)

$$(\alpha.E)\backslash\!\backslash L \;\ll\; \tau.(E\backslash\!\backslash L) \qquad \textit{if } \alpha \in L \tag{6.3}$$

$$(E + F)\backslash\!\backslash L \;\ll\; E\backslash\!\backslash L + F\backslash\!\backslash L \tag{6.4}$$

$$(E \mid F)\backslash\!\backslash L \;\ll\; E\backslash\!\backslash L \mid F\backslash\!\backslash L \qquad \textit{if } L = \overline{L} \tag{6.5}$$

Proof.

6.3 Holds because $(\alpha.E)\backslash\!\backslash L \sim \tau.(E\backslash\!\backslash L)$ if $\alpha \in L$ (Prop. 4.9).

6.4 Holds because $(E + F)\backslash\!\backslash L \sim E\backslash\!\backslash L + F\backslash\!\backslash L$ (Prop. 4.9).

6.5 We prove $(E \mid F)\backslash\!\backslash L \ll E\backslash\!\backslash L \mid F\backslash\!\backslash L$ by showing that

$$\{((E \mid F)\backslash\!\backslash L, (E\backslash\!\backslash L \mid F\backslash\!\backslash L)) \mid E \in \mathcal{P}, L = \overline{L}\}$$

is a weak simulation. Suppose $(E \mid F)\backslash\!\backslash L \xrightarrow{\alpha} (E' \mid F')\backslash\!\backslash L$. There are three cases to consider. Suppose the transition is the result of independent action by E. Then $E \xrightarrow{\alpha} E'$ and $\alpha \notin L$ and $F \equiv F'$, so we can infer $E\backslash\!\backslash L \xrightarrow{\alpha} E'\backslash\!\backslash L$ and $E\backslash\!\backslash L \mid F\backslash\!\backslash L \xrightarrow{\alpha} E'\backslash\!\backslash L \mid F\backslash\!\backslash L$. The case for independent action by F is symmetrical.

Suppose the transition is the result of synchronisation. Then $E \xrightarrow{l} E'$ and $F \xrightarrow{\overline{l}} F'$. There are two sub-cases, depending on whether $l \in L$ or not. Suppose

$l \in L$. Then $E \backslash\backslash L \xrightarrow{\tau} E' \backslash\backslash L$ and $F \backslash\backslash L \xrightarrow{\tau} F' \backslash\backslash L$, so $E \backslash\backslash L \mid F \backslash\backslash L \stackrel{\varepsilon}{\Rightarrow} E' \backslash\backslash L \mid F' \backslash\backslash L$. In the second sub-case, suppose $l \notin L$. Then $E \backslash\backslash L \xrightarrow{l} E' \backslash\backslash L$ and $F \backslash\backslash L \xrightarrow{\bar{l}} F' \backslash\backslash L$, so again $E \backslash\backslash L \mid F \backslash\backslash L \stackrel{\varepsilon}{\Rightarrow} E' \backslash\backslash L \mid F' \backslash\backslash L$. □

The hiding laws allow the hiding operator to be moved inwards in a process. The first two laws follow directly from laws of strong equivalence. The third law allows the movement of hiding inwards through composition. It may seem that this law is useless for abstraction, since it reduces synchronisation and therefore increases the state space of a process. However, it can enable later simplification by making the behaviour of a process more regular.

The following laws for relabelling are similar to the hiding laws, allowing relabelling operators to be moved inwards in a process.

Proposition 6.4 (relabelling laws)

$$(\alpha.E)[f] \ll f(\alpha).(E[f]) \tag{6.6}$$

$$(E + F)[f] \ll E[f] + F[f] \tag{6.7}$$

$$(E \mid F)[f] \ll E[f] \mid F[f] \tag{6.8}$$

Proof.

6.6 Holds because $(\alpha.E)[f] \sim f(\alpha).(E[f])$ (Prop. 4.9).

6.7 Holds because $(E + F)[f] \sim E[f] + F[f]$ (Prop. 4.9).

6.8 We prove that $(E \mid F)[f] \ll E[f] \mid F[f]$ by showing that

$$\{((E \mid F)[f], (E[f] \mid F[f])) \mid E \in \mathcal{P}\}$$

is a weak simulation. Suppose $(E \mid F)[f] \xrightarrow{\alpha} (E' \mid F')[f]$. There are three cases to consider. Suppose the transition is the result of independent action by E. Then $E \xrightarrow{\beta} E'$ and $f(\beta) = \alpha$ and $F \equiv F'$, so we can infer $E[f] \xrightarrow{\alpha} E'[f]$

and $E[f] \mid F[f] \xrightarrow{\alpha} E'[f] \mid F'[f]$. The case for independent action by F is symmetrical.

Suppose the transition is the result of synchronisation between E and F. Then $E \xrightarrow{l} E'$ and $F \xrightarrow{\bar{l}} F'$. Since relabelling respects complementation we can infer $E[f] \xrightarrow{f(l)} E'[f]$ and $F[f] \xrightarrow{\overline{f(l)}} F'[f]$ and $E[f] \mid F[f] \xrightarrow{\tau} E'[f] \mid F'[f]$. $\square$

Next we present two laws that operate directly on families of definitions.

Proposition 6.5 (family laws)

$$\mathcal{F} \ll \text{merge}(A_j, A_k, \mathcal{F}) \tag{6.9}$$

$$\mathcal{F}\{A \stackrel{\text{def}}{=} A + E\} \ll \mathcal{F}\{A \stackrel{\text{def}}{=} E\} \tag{6.10}$$

Proof.

6.9 Since $E_j < E_j + E_k$ and $<$ is a precongruence,

$$\mathcal{F} < \mathcal{F}\{A_j \stackrel{\text{def}}{=} E_j + E_k\}.$$

Furthermore, if $A_j \sim A_k$ then

$$\mathcal{F} \sim \mathcal{F}\{A_j/A_k\}$$

because $\sim$ is a congruence. Therefore

$$\begin{array}{ll} & \mathcal{F} \\ < & \mathcal{F}\{A_j \stackrel{\text{def}}{=} E_j + E_k\} \\ < & \mathcal{F}\{A_j \stackrel{\text{def}}{=} E_j + E_k\}\{A_k \stackrel{\text{def}}{=} E_k + (E_j + E_k)\} \\ \sim & \mathcal{F}\{A_j \stackrel{\text{def}}{=} E_j + E_k\}\{A_k \stackrel{\text{def}}{=} E_j + E_k\} \\ \sim & \mathcal{F}\{A_j \stackrel{\text{def}}{=} E_j + E_k\}\{A_k \stackrel{\text{def}}{=} E_j + E_k\}\{A_j/A_k\} \\ \sim & \text{merge}(A_j, A_k, \mathcal{F}). \end{array}$$

6.10 We show that relation $\mathcal{R}$, defined as all pairs of the form

$$(E\{A_1/X\}, E\{A_2/X\})$$

where E is a process containing at most variable X, and $A_1 \stackrel{\text{def}}{=} A_1 + E\{A_1/X\}$ and $A_2 \stackrel{\text{def}}{=} E\{A_2/X\}$, is a simulation. To do this we prove the following condition on $\mathcal{R}$ by induction on the length of inference of transition.

> Whenever $E\{A_1/X\} \xrightarrow{\alpha} E'$ then $E\{A_2/X\} \xrightarrow{\alpha} E''$ for some E'' such that $(E', E'') \in \mathcal{R}$,

We perform case analysis on the structure of E. Let S_1 be A_1/X and S_2 be A_2/X in all cases.

Case $E \equiv X$**.** Suppose $X\{S_1\} \equiv A_1 \xrightarrow{\alpha} E'$. Then $(A_1 + E)\{S_1\} \equiv A_1 + E\{A_1\} \equiv A_1 + A_1 \xrightarrow{\alpha} E'$ by a shorter transition and $A_1 \equiv E\{S_1\} \xrightarrow{\alpha} E'$ by an even shorter transition. By induction $E\{S2\} \xrightarrow{\alpha} E''$ for some E_2 such that $(E', E'') \in \mathcal{R}$.

Case $E \equiv \alpha.E'$**.** Transitions $\alpha.E'\{S_1\} \xrightarrow{\alpha} E'\{S_1\}$ and $\alpha.E'\{S_2\} \xrightarrow{\alpha} E'\{S_2\}$ are both possible, with the derivatives in $\mathcal{R}$.

Case $E \equiv E_1 + E_2$**.** Suppose $(E_1 + E_2)\{S_1\} \equiv E_1\{S_1\} + E_2\{S_1\} \xrightarrow{\alpha} E'$. There are two cases to consider. If $E_1\{S_1\} \xrightarrow{\alpha} E'$ then $E_1\{S_2\} \xrightarrow{\alpha} E''$ and $(E', E'') \in \mathcal{R}$ by the induction hypothesis. We can then infer $E_1\{S_2\} + E_2\{S_2\} \equiv (E_1 + E_2)\{S_2\} \xrightarrow{\alpha} E''$. The other case is symmetrical.

Case $E \equiv E_1 \mid E_2$**.** Suppose $(E_1 \mid E_2)\{S_1\} \equiv E_1\{S_1\} \mid E_2\{S_1\} \xrightarrow{\alpha} E_1' \mid E_2'$. There are three cases to consider. The first two cases apply if this transition arises from independent action of $E_1\{S_1\}$ or $E_2\{S_1\}$. The argument here is similar to that for the previous case. The third case applies if the transition arises from synchronisation. Suppose $E_1\{S_1\} \xrightarrow{l} E_1'$ and $E_2\{S_1\} \xrightarrow{\bar{l}} E_2'$. Then $E_1\{S_2\} \xrightarrow{l} E_1''$ and $E_2\{S_2\} \xrightarrow{\bar{l}} E_2''$ and $(E_1', E_1'') \in \mathcal{R}$ and $(E_2', E_2'') \in \mathcal{R}$ by the induction hypothesis. We can then infer $E_1\{S_2\} \mid E_2\{S_2\} \equiv (E_1 \mid E_2)\{S_2\} \xrightarrow{\alpha} E_1'' \mid E_2''$. Furthermore, since

$(E_1', E_1'') \in \mathcal{R}$ we know $E_1' \equiv F_1\{S_1\}$ and $E_1'' \equiv F_1\{S_2\}$ for some F_1. Similarly we know $E_2' \equiv F_2\{S_1\}$ and $E_2'' \equiv F_2\{S_2\}$ for some F_2. Therefore $(E_1' \mid E_2', E_1'' \mid E_2'') \equiv (F_1\{S_1\} \mid F_2\{S_1\}, F_1\{S_2\} \mid F_2\{S_2\}) \equiv ((F_1 \mid F_2)\{S_1\}, (F_1 \mid F_2)\{S_2\}) \in \mathcal{R}$.

Case $E \equiv E'\backslash L$**.** Suppose $E'\backslash L\{S_1\} \equiv E'\{S_1\}\backslash L \xrightarrow{\alpha} E''\backslash L$. Then $E'\{S_1\} \xrightarrow{\alpha} E''$ by a shorter inference so $E'\{S_2\} \xrightarrow{\alpha} E'''$ and $(E'', E''') \in \mathcal{R}$ by induction. Since $\alpha \notin L \cup \overline{L}$ we can infer $E'\{S_2\}\backslash L \equiv E'\backslash L\{S_2\} \xrightarrow{\alpha} E'''\backslash L$. Furthermore, since $(E'', E''') \in \mathcal{R}$ we know $E'' \equiv F\{S_1\}$ and $E''' \equiv F\{S_2\}$ for some F, and then $(E''\backslash L, E'''\backslash L) \equiv (F\{S_1\}\backslash L, F\{S_2\}\backslash L) \equiv (F\backslash L\{S_1\}, F\backslash L\{S_2\}) \in \mathcal{R}$.

Case $E \equiv E'[f]$**.** By reasoning similar to the previous case.

Case $E \equiv B$**.** Here B is a constant defined by $B \stackrel{\text{def}}{=} F$, with X not appearing in F. Suppose $B\{S_1\} \xrightarrow{\alpha} E'$. Since B is a constant, $B\{S_1\}$ and $B\{S_2\}$ are identical, so both have E' as an α-derivative. Furthermore, $E' \equiv E'\{S_1\} \equiv E'\{S_2\}$, so $(E'\{S_1\}, E'\{S_2\}) \in \mathcal{R}$. □

Both laws are used to eliminate constants. The first law allows a constant definition and all references to it to be eliminated. The second law allows a single unguarded constant to be eliminated. An algebraic law similar to the second law is presented (but not proved) for finite-state agents in Section 7.5 of [72].

Proposition 6.6 (basic laws)

$$E \mid 0 \ll E \tag{6.11}$$

$$\tau.E \ll E \tag{6.12}$$

$$A \ll E \quad \textit{if } A \stackrel{\text{def}}{=} E \tag{6.13}$$

$$E \ll A \quad \textit{if } A \stackrel{\text{def}}{=} E \tag{6.14}$$

Proof. Law 6.11 holds because $E \mid 0 \sim E$ (Prop. 2.7). Law 6.12 holds because $\tau.E \approx E$ (Prop. 5.7 of [72]). Laws 6.13 and 6.14 hold because $A \stackrel{\text{def}}{=} E$ implies $A \sim E$ (Prop. 2.7). □

The first of the basic laws allows inactive components to be eliminated. The second law allows τ actions to be removed wherever they occur. The third and fourth laws allow constants to be eliminated and introduced.

6.4 Derived Operations

The laws of the previous section are commonly applied in certain combinations. Here we present derived laws that allow abstraction to proceed in coarser steps.

Proposition 6.7 (derived laws) *Let K and L be sets of labels such that $K = \overline{K} \subseteq L \cup \overline{L}$, and let f be a relabelling function satisfying $f(\alpha) = \alpha$ ($\alpha \notin L \cup \overline{L}$). Then*

$$(E_1 \mid \cdots \mid E_n)\backslash L \ll (E_1 \backslash\!\backslash K \mid \cdots \mid E_n \backslash\!\backslash K)\backslash L \tag{6.15}$$

$$(E_1 \mid \cdots \mid E_n)\backslash L \ll (E_1[f] \mid \cdots \mid E_n[f])\backslash L \tag{6.16}$$

Proof.

6.15
$$\begin{aligned} & (E_1 \mid \ldots \mid E_n)\backslash L \\ \ll\ & (E_1 \mid \ldots \mid E_n)\backslash\!\backslash K\backslash L && \text{(by restriction law 6.1)} \\ \ll\ & (E_1\backslash\!\backslash K \mid \ldots \mid E_n\backslash\!\backslash K)\backslash L && \text{(by hiding law 6.5)} \end{aligned}$$

6.16
$$\begin{aligned} & (E_1 \mid \ldots \mid E_n)\backslash L \\ \ll\ & (E_1 \mid \ldots \mid E_n)[f]\backslash L && \text{(by restriction law 6.2)} \\ \ll\ & (E_1[f] \mid \ldots \mid E_n[f])\backslash L && \text{(by relabelling law 6.8)} \end{aligned}$$

□

The first law supports the hiding of synchronisation actions. The law is derived by first hiding a restricted action and then distributing the hiding over composition. Notice that hiding only some occurrences of a synchronisation action does not necessarily result in a weaker process. For example, $(a.b.0 \mid \overline{a}.0)\backslash a \not\ll (a.b.0 \mid (\overline{a}.0)\backslash\!\backslash a)\backslash a$.

The second law supports the relabelling of synchronisation actions.

6.5 Example – Milner's Scheduler

Here we illustrate our technique by abstracting Milner's task scheduling, which was presented in Section 4.4.3. The scheduler is supposed to perform begin and end signals for each task in alternation. This property can be expressed as a formula of $\mu\mathbb{L}\Box$.

$$\begin{aligned}
Cycle(L_1, L_2) &\stackrel{\text{def}}{=} \nu X_1.[\![L_2]\!]\,\mathtt{ff} \wedge [\![-L_1, L_2]\!]X_1 \;\wedge\; [\![L_1]\!] \\
&\quad\;\; \nu X_2.[\![L_1]\!]\,\mathtt{ff} \wedge [\![-L_1, L_2]\!]X_2 \;\wedge\; [\![L_2]\!]X_1 \\
AltSig &\stackrel{\text{def}}{=} \bigwedge_{0 \leq i < n} Cycle(\{a_i\}, \{b_i\})
\end{aligned}$$

The scheduler satisfies property *AltSig* because the a_i and b_i actions are performed only by process i, and occur in alternation there. We now apply abstraction operations to remove other details from the algorithm. We begin by hiding all c_i actions and their complements using derived law 6.15, and move the hiding inward using the hiding laws.

$$\begin{aligned}
A_i &\stackrel{\text{def}}{=} a_i.\tau.E_i \\
E_i &\stackrel{\text{def}}{=} b_i.D_i + \tau.b_i.A_i \\
D_i &\stackrel{\text{def}}{=} \tau.A_i
\end{aligned}$$

To make the presentation concise we have reused the constants of the algorithm (as allowed by Proposition 6.1) rather than invent new ones. Also, we have shown only the constant definitions that were affected by the abstraction step.

We now remove all τ actions by repeatedly applying basic law 6.12.

$$\begin{aligned}
A_i &\stackrel{\text{def}}{=} a_i.E_i \\
E_i &\stackrel{\text{def}}{=} b_i.D_i + b_i.A_i \\
D_i &\stackrel{\text{def}}{=} A_i
\end{aligned}$$

Family law 6.9 is now applied to merge constants A_i, and D_i.

$$\begin{aligned} A_i &\stackrel{\text{def}}{=} a_i.E_i + A_i \\ E_i &\stackrel{\text{def}}{=} b_i.A_i + b_i.A_i \\ \mathit{Sched} &\stackrel{\text{def}}{=} (A_0 \mid A_1 \mid A_2 \mid \cdots \mid A_{n-1})\backslash\{c_0, c_1, \ldots, c_{n-1}\} \end{aligned}$$

We remove the unguarded occurrence of constant A_i using family law 6.10.

$$\begin{aligned} A_i &\stackrel{\text{def}}{=} a_i.E_i \\ E_i &\stackrel{\text{def}}{=} b_i.A_i \end{aligned}$$

Finally, we eliminate the constant E_i using basic law 6.13, giving the complete abstract algorithm.

$$\begin{aligned} A_i &\stackrel{\text{def}}{=} a_i.b_i.A_i \\ \mathit{Sched} &\stackrel{\text{def}}{=} (A_0 \mid A_1 \mid A_2 \mid \cdots \mid A_{n-1})\backslash\{c_0, c_1, \ldots, c_{n-1}\} \end{aligned}$$

The abstract scheduler weakly simulates the original scheduler so by Prop. 4.15 we know that the original scheduler satisfies formula *AltSig* if the abstract one does. With the Concurrency Workbench we confirm that the abstract one does indeed satisfy *AltSig*. This example shows the idea of abstraction with the weakly simulates relation but does not show how useful it can be. In the next chapter the technique is applied to a more serious example.

6.6 Discussion

There is existing work on process abstraction using the simulates relation, but none that gives algebraic abstraction operations. Lynch [69] abstracts I/O automata using the simulates relation, but requires that a simulation relation be invented and checked. The logical effect of abstraction with simulation is not given; it is only stated that simulation is a stronger relation than trace inclusion.

Clarke *et al* [19] abstract finite-state, sequential, procedural programs by mappings on program inputs and outputs. The abstraction operations are specific operations on program inputs and outputs. For example, they consider congruence modulo an integer for abstraction with arithmetic operations. No abstraction operations on control structure are given. The only low-level abstraction operation supported is the merging of states in a transition system. The main theorem of [19] states that the merging of states preserves formulas of CTL* not containing existential path quantifiers. Formulas are interpreted over infinite computation paths, so only weak eventuality properties can be expressed.

Other process preorders are also used for abstraction. Cleaveland and Riely [29] abstract value-passing CCS processes by mappings on data domains. They show that the abstract process is greater in the specification preorder than the concrete process. The abstraction operations do not operate on process structure, and their logical effects are not given.

McIsaac [70] abstracts transition systems to modal transition systems [64]. A modal transition system has two transition relations: $\xrightarrow{\alpha}_{\Box}$ and $\xrightarrow{\alpha}_{\Diamond}$. Modal operators are interpreted on modal transition systems as follows: formula $[L]\phi$ holds of a state s if all $\Diamond$-derivatives of s satisfy ϕ; $\langle L\rangle\phi$ holds of s if some $\Box$-derivative of s satisfies ϕ. McIsaac uses state merging as an abstraction operation and defines modal transitions between the abstract states created by merging basic states. If *every* basic state in an abstract state s can perform an action a leading to a derivative in abstract state s', then $s \xrightarrow{a}_{\Box} s'$. If *some* basic state in an abstract state s can perform an action a leading to a derivative in abstract state s', then $s \xrightarrow{a}_{\Diamond} s'$. McIsaac showed that all μL formulas satisfied by the abstract modal transition system also hold of the concrete transition system. However, state merging is the only abstraction operation investigated by McIsaac and it is carried out purely semantically; no operations on process expressions are defined. A related preservation result by Larsen [66] shows that HML formulas are preserved by the *refinement* relation on modal transition systems, which is slightly stronger than the relation induced by state merging operations.

Chapter 7

Abstracting Dekker's Mutual Exclusion Algorithm

In this chapter we illustrate our abstraction techniques by abstracting Dekker's mutual exclusion algorithm [75] relative to a safety property and a liveness property. In each case, we show that if the abstract algorithm satisfies the property, then so does the concrete algorithm. The resulting abstract algorithms have smaller process expressions and state spaces than the concrete algorithm.

7.1 Dekker's Algorithm

The purpose of a mutual exclusion algorithm is to control access by a set of processes to a shared resource so that at most one process has access at any time. A process is said to be in its *critical section* when it has access to the resource. The key safety property of a mutual exclusion algorithm is that at most one process is in its critical section at any time. The key liveness property is that a process wishing to enter its critical section will eventually be able to do so.

We now present a two-process version of Dekker's mutual exclusion algorithm. The top level of the algorithm, written in a concurrent while language, is as follows.

begin
    var b_1,b_2,k;
    b_1 := **false**; b_2 := **false**;
    $k := 1$;
    P_1 **par** P_2
end;

Processes P_1 and P_2 coordinate via shared variables b_1, b_2, and k. Process P_1 is defined as follows; P_2 is obtained from P_1 by interchanging 1 with 2 everywhere. The non-critical section of the process is where it does its useful work.

while true do
begin
    ⟨ non-critical section ⟩;
    b_1 := **true**;
    while b_2 **do**
        if $k = 2$ **then begin**
                        b_1 := **false**;
                        while $k = 2$ **do skip**;
                        b_1 := **true**
                    end;
    ⟨ critical section ⟩;
    $k := 2$;
    b_1 := **false**
end

Informally, the b variables are "request" variables and the k variable is a "turn" variable. Variable b_i is true if P_i is requesting entry to its critical section; variable k is i if it is P_i's turn to enter its critical section. Only P_i writes on variable b_i, but both processes read b_i.

We translate the algorithm to CCS following Walker's translation of [95]. Two main ideas lie behind the translation. First, the shared variables are translated to new concurrent components. Each new component has actions for the reading and writing of the variable it represents. For example, shared variable b_1 is translated to process $B1$ with read actions $\overline{\mathtt{b_1rt}}$ and $\overline{\mathtt{b_1rf}}$, and write actions $\mathtt{b_1wt}$ and $\mathtt{b_1wf}$. Action $\overline{\mathtt{b_1rt}}$ represents a read of value *true*, and action $\mathtt{b_1wf}$ represents a write of value *false*. In restriction operators we often write $\mathtt{b_1}$ to stand for the set $\{\mathtt{b_1rt}, \mathtt{b_1wt}, \mathtt{b_1rf}, \mathtt{b_1wf}\}$ of actions, and similarly for other shared variables.

The second idea in the translation is the addition of *probe* actions [95] to allow the behaviour of P_1 and P_2 to be observed. Action $\mathtt{req}_i$ represents a request by process i to enter its critical section; actions $\mathtt{enter}_i$ and $\mathtt{exit}_i$ represent entrance to and exit from the critical section by process i.

We omit the definitions for *B2f* and *P2* in the family presented below as they are symmetrical to *B1f* and *P1*.

$$
\begin{aligned}
\mathit{K1} &\stackrel{\text{def}}{=} \overline{\mathtt{kr1}}\,.\mathit{K1} + \mathtt{kw1}\,.\mathit{K1} + \mathtt{kw2}\,.\mathit{K2}\\
\mathit{K2} &\stackrel{\text{def}}{=} \overline{\mathtt{kr2}}\,.\mathit{K2} + \mathtt{kw1}\,.\mathit{K1} + \mathtt{kw2}\,.\mathit{K2}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{B1f} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1rf}}\,.\mathit{B1f} + \mathtt{b_1wf}\,.\mathit{B1f} + \mathtt{b_1wt}\,.\mathit{B1t}\\
\mathit{B1t} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1rt}}\,.\mathit{B1t} + \mathtt{b_1wf}\,.\mathit{B1f} + \mathtt{b_1wt}\,.\mathit{B1t}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{P1} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}\,.\,\mathtt{req_1}\,.\mathit{P11}\\
\mathit{P11} &\stackrel{\text{def}}{=} \mathtt{b_2rf}\,.\mathit{P13} + \mathtt{b_2rt}\,.(\mathtt{kr1}\,.\mathit{P11} + \mathtt{kr2}\,.\,\overline{\mathtt{b_1wf}}\,.\mathit{P12})\\
\mathit{P12} &\stackrel{\text{def}}{=} \mathtt{kr1}\,.\,\overline{\mathtt{b_1wt}}\,.\mathit{P11} + \mathtt{kr2}\,.\tau.\mathit{P12}\\
\mathit{P13} &\stackrel{\text{def}}{=} \mathtt{enter_1}\,.\,\mathtt{exit_1}\,.\,\overline{\mathtt{kw2}}\,.\,\overline{\mathtt{b_1wf}}\,.\mathit{P1}
\end{aligned}
$$

$$
\mathit{Dekker} \stackrel{\text{def}}{=} (\mathit{P1} \mid \mathit{P2} \mid \mathit{B1f} \mid \mathit{B2f} \mid K1)\backslash\{\mathtt{b_1}, \mathtt{b_2}, \mathtt{k}\}
$$

Figure 7–1 is the flow diagram for process *Dekker*. To simplify the diagram, ports $\mathtt{b_1wt}$ and $\mathtt{b_1wf}$ of process *B1* have been drawn as a single port. Similar simplifications have been made for *B2* and *K*.

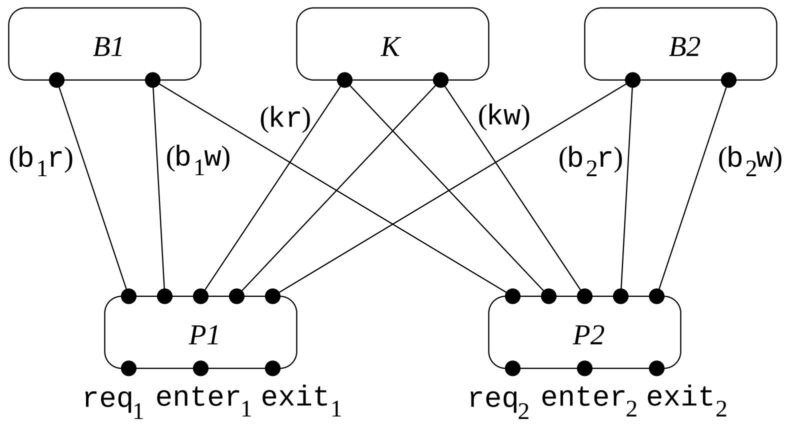


Figure 7–1: Flow Diagram for CCS Model of Dekker's Algorithm

7.1.1 Safety

The property of *mutual exclusion* is that at most one process can be in its critical section at any time. The property can be expressed as formula *ME* of $\mu\mathbb{L}\Box$ requiring that `enter` and `exit` actions alternate:

$$
\begin{aligned}
Cycle(L_1, L_2) &\stackrel{\text{def}}{=} \nu X_1 . \llbracket L_2 \rrbracket\, \mathtt{ff} \wedge \llbracket -L_1, L_2 \rrbracket X_1 \;\wedge\; \llbracket L_1 \rrbracket \\
&\qquad \nu X_2 . \llbracket L_1 \rrbracket\, \mathtt{ff} \wedge \llbracket -L_1, L_2 \rrbracket X_2 \;\wedge\; \llbracket L_2 \rrbracket X_1 \\
ME &\stackrel{\text{def}}{=} Cycle(\{\mathtt{enter}_1, \mathtt{enter}_2\}, \{\mathtt{exit}_1, \mathtt{exit}_2\})
\end{aligned}
$$

Formula $Cycle(L_1, L_2)$ expresses that actions in set L_1 must occur in alternation with actions in set L_2. Note that $Cycle(L_1, L_2)$ does not require that any action of L_1 or L_2 ever be performed.

Dekker's algorithm satisfies the property of mutual exclusion because a process sets its request variable to true before attempting to enter its critical section, and waits for the request variable of the other process to be false before actually entering it. Dekker's algorithm can be abstracted greatly with respect to mutual exclusion because much of the algorithm's design deals with the problem of ensuring liveness.

We follow the same conventions in presenting the abstraction steps as used in the scheduler example of the previous chapter. We reuse the constants of the algorithm at each step as allowed by Proposition 6.1 and present only the constant definitions that were affected by the abstraction step.

We abstract Dekker's algorithm in two stages. In the first stage we apply the Correspondence Rule to remove the indices of the $\texttt{enter}_i$ and $\texttt{exit}_i$ actions and to hide the $\texttt{req}_i$ actions. If the resulting process satisfies $Cycle(\{\texttt{enter}\}, \{\texttt{exit}\})$, then *Dekker* satisfies *ME*. Let set L be $\{\texttt{req}_1, \texttt{req}_2, \overline{\texttt{req}_1}, \overline{\texttt{req}_2}\}$ and let relabelling function f be $(\texttt{enter}/\texttt{enter}_1, \texttt{enter}/\texttt{enter}_2, \texttt{exit}/\texttt{exit}_1, \texttt{exit}/\texttt{exit}_2)$. Then

$$\begin{array}{llll}
 & (Dekker \backslash\!\backslash L)[f] & \models & Cycle(\{\texttt{enter}\}, \{\texttt{exit}\}) & \\
\Leftrightarrow & Dekker \backslash\!\backslash L & \models & \mathcal{W}_{[f]}(Cycle(\{\texttt{enter}\}, \{\texttt{exit}\})) & \text{by Corres. Rule} \\
\Leftrightarrow & Dekker \backslash\!\backslash L & \models & ME & \text{by Prop. 4.4} \\
\Rightarrow & Dekker & \models & ME & \text{by Prop. 5.11}
\end{array}$$

Simplifying $(Dekker \backslash\!\backslash L)[f]$ by the hiding and relabelling laws of the previous chapter gives the following family, in which constants have been reused according to Proposition 6.1. We refer to *Dekker* of this family as $Dekker_1$.

$$\begin{array}{rcl}
P1 & \stackrel{\text{def}}{=} & \overline{\texttt{b}_1\texttt{wt}}\,.P11 \\
P11 & \stackrel{\text{def}}{=} & \texttt{b}_2\texttt{rf}\,.P13 + \texttt{b}_2\texttt{rt}\,.(\texttt{kr1}\,.P11 + \texttt{kr2}\,.\overline{\texttt{b}_1\texttt{wf}}\,.P12) \\
P12 & \stackrel{\text{def}}{=} & \texttt{kr1}\,.\overline{\texttt{b}_1\texttt{wt}}\,.P11 + \texttt{kr2}\,.\tau.P12 \\
P13 & \stackrel{\text{def}}{=} & \texttt{enter}\,.\texttt{exit}\,.\overline{\texttt{kw2}}\,.\overline{\texttt{b}_1\texttt{wf}}\,.P1
\end{array}$$

In the second stage of abstraction we use only the abstraction operations of the previous chapter, which all preserve formula $Cycle(\{\texttt{enter}\}, \{\texttt{exit}\})$.

Our informal explanation of why Dekker's algorithm satisfies mutual exclusion mentions only the request variables, not the turn variable k. We therefore hide k-related actions $\{\texttt{kr1}, \texttt{kw1}, \texttt{kr2}, \texttt{kw2}\}$ and their complements using derived law 6.15, and move the hiding inward using the hiding laws. We refer to *Dekker* of the resulting family as $Dekker_2$.

$$\begin{array}{rcl}
K1 & \stackrel{\text{def}}{=} & \tau.K1 + \tau.K1 + \tau.K2 \\
K2 & \stackrel{\text{def}}{=} & \tau.K2 + \tau.K1 + \tau.K2
\end{array}$$

$$
\begin{aligned}
\mathit{P1} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}\,.\mathit{P11}\\
\mathit{P11} &\stackrel{\text{def}}{=} \mathtt{b_2rf}\,.\mathit{P13} + \mathtt{b_2rt}\,.(\tau.\mathit{P11} + \tau.\overline{\mathtt{b_1wf}}\,.\mathit{P12})\\
\mathit{P12} &\stackrel{\text{def}}{=} \tau.\overline{\mathtt{b_1wt}}\,.\mathit{P11} + \tau.\tau.\mathit{P12}\\
\mathit{P13} &\stackrel{\text{def}}{=} \mathtt{enter}\,.\,\mathtt{exit}\,.\tau.\overline{\mathtt{b_1wf}}\,.\mathit{P1}
\end{aligned}
$$

The actions related to variables b_1 and b_2 cannot all be hidden, as variables b_1 and b_2 play a part in ensuring mutual exclusion. However, only some of the b-related actions are involved. We can hide the actions that represent when a b variable is read with value true. The effect is to allow P_1 to proceed as if b_2 is true whether b_2 is true or not. Thus P_1 can elect not to enter its critical section even if b_2 is false. Similarly P_2 can wait instead of entering its critical section.

Applying derived law 6.15 with actions $\{\mathtt{b_1rt}, \mathtt{b_2rt}\}$ and their complements, and moving hiding inward with the hiding laws gives the following.

$$
\begin{aligned}
\mathit{B1f} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1rf}}\,.\mathit{B1f} + \mathtt{b_1wf}\,.\mathit{B1f} + \mathtt{b_1wt}\,.\mathit{B1t}\\
\mathit{B1t} &\stackrel{\text{def}}{=} \tau.\mathit{B1t} + \mathtt{b_1wf}\,.\mathit{B1f} + \mathtt{b_1wt}\,.\mathit{B1t}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{P1} &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}\,.\mathit{P11}\\
\mathit{P11} &\stackrel{\text{def}}{=} \mathtt{b_2rf}\,.\mathit{P13} + \tau.(\tau.\mathit{P11} + \tau.\overline{\mathtt{b_1wf}}\,.\mathit{P12})\\
\mathit{P12} &\stackrel{\text{def}}{=} \tau.\overline{\mathtt{b_1wt}}\,.\mathit{P11} + \tau.\tau.\mathit{P12}\\
\mathit{P13} &\stackrel{\text{def}}{=} \mathtt{enter}\,.\,\mathtt{exit}\,.\tau.\overline{\mathtt{b_1wf}}\,.\mathit{P1}
\end{aligned}
$$

We now remove all τ actions by repeatedly applying basic law 6.12.

$$
\begin{aligned}
\mathit{K1} &\stackrel{\text{def}}{=} \mathit{K1} + \mathit{K1} + \mathit{K2}\\
\mathit{K2} &\stackrel{\text{def}}{=} \mathit{K2} + \mathit{K1} + \mathit{K2}
\end{aligned}
$$

$$
\mathit{B1f} \stackrel{\text{def}}{=} \overline{\mathtt{b_1rf}}\,.\mathit{B1f} + \mathtt{b_1wf}\,.\mathit{B1f} + \mathtt{b_1wt}\,.\mathit{B1t}
$$

$$B1t \stackrel{\text{def}}{=} B1t + \mathtt{b_1wf}\,.B1f + \mathtt{b_1wt}\,.B1t$$

$$\begin{aligned}
P1 &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}\,.P11 \\
P11 &\stackrel{\text{def}}{=} \mathtt{b_2rf}\,.P13 + (P11 + \overline{\mathtt{b_1wf}}\,.P12) \\
P12 &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}\,.P11 + P12 \\
P13 &\stackrel{\text{def}}{=} \mathtt{enter}\,.\,\mathtt{exit}\,.\,\overline{\mathtt{b_1wf}}\,.P1
\end{aligned}$$

We apply family law 6.10 to remove all unguarded occurrences of constants.

$$\begin{aligned}
K1 &\stackrel{\text{def}}{=} 0 \\
K2 &\stackrel{\text{def}}{=} 0
\end{aligned}$$

$$\begin{aligned}
B1f &\stackrel{\text{def}}{=} \overline{\mathtt{b_1rf}}\,.B1f + \mathtt{b_1wf}\,.B1f + \mathtt{b_1wt}\,.B1t \\
B1t &\stackrel{\text{def}}{=} \mathtt{b_1wf}\,.B1f + \mathtt{b_1wt}\,.B1t
\end{aligned}$$

$$\begin{aligned}
P1 &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}\,.P11 \\
P11 &\stackrel{\text{def}}{=} \mathtt{b_2rf}\,.P13 + \overline{\mathtt{b_1wf}}\,.P12 \\
P12 &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}\,.P11 \\
P13 &\stackrel{\text{def}}{=} \mathtt{enter}\,.\,\mathtt{exit}\,.\,\overline{\mathtt{b_1wf}}\,.P1
\end{aligned}$$

The transition system for process *P1* as abstracted so far is shown in Figure 7–2. The states enclosed by the dotted line are outside the critical section but have not yet requested entry to the critical section. To prepare for the merging of these states we apply basic law 6.14 to introduce constants *P14* and *P24*.

$$\begin{aligned}
P1 &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}\,.P11 \\
P11 &\stackrel{\text{def}}{=} \mathtt{b_2rf}\,.P13 + \overline{\mathtt{b_1wf}}\,.P12 \\
P12 &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}\,.P11 \\
P13 &\stackrel{\text{def}}{=} \mathtt{enter}\,.\,\mathtt{exit}\,.P14 \\
P14 &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wf}}\,.P1
\end{aligned}$$

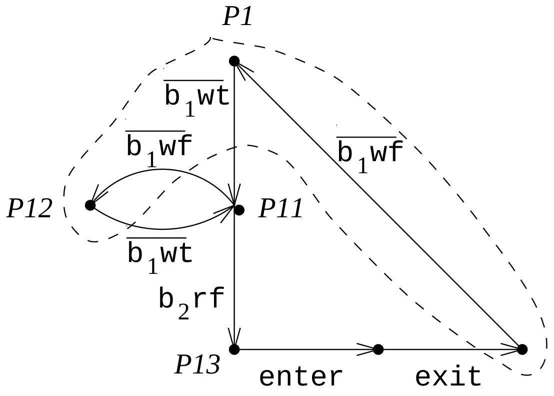


Figure 7–2: Transition Diagram for Partly-abstracted Process *P1*

Family law 6.9 is now applied to merge constants *P1*, *P12*, and *P14*. Similarly, constants *P2*, *P22*, and *P24* of *P2* are merged. Constants *K1* and *K2* are then removed using basic law 6.11 to obtain the following family. We refer to *Dekker* of this family as $Dekker_3$.

$$\begin{aligned}
B1f &\stackrel{\text{def}}{=} \overline{\mathtt{b_1rf}}.B1f + \mathtt{b_1wf}.B1f + \mathtt{b_1wt}.B1t \\
B1t &\stackrel{\text{def}}{=} \mathtt{b_1wf}.B1f + \mathtt{b_1wt}.B1t
\end{aligned}$$

$$\begin{aligned}
P1 &\stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}.P11 + \overline{\mathtt{b_1wf}}.P1 \\
P11 &\stackrel{\text{def}}{=} \mathtt{b_2rf}.P13 + \overline{\mathtt{b_1wf}}.P1 \\
P13 &\stackrel{\text{def}}{=} \mathtt{enter}.\mathtt{exit}.P1
\end{aligned}$$

$$Dekker \stackrel{\text{def}}{=} (P1 \mid P2 \mid B1f \mid B2f)\backslash\{\mathtt{b_1}, \mathtt{b_2}\}$$

The abstract algorithm contains little more than the protocol for handling the request variables to ensure mutual exclusion. While process *Dekker* has 153 states, $Dekker_3$ has only 16 states. With a tool like the Concurrency Workbench [28] it is easy to show that $Dekker_3 \models Cycle(\{\mathtt{enter}\}, \{\mathtt{exit}\})$. Since $Dekker_3$ was obtained by abstraction operations that preserve $Cycle(\{\texttt{enter}\}, \{\texttt{exit}\})$ we know $Dekker_1$ also satisfies $Cycle(\{\texttt{enter}\}, \{\texttt{exit}\})$. Then, since

$$Dekker_1 \models Cycle(\{\texttt{enter}\}, \{\texttt{exit}\}) \Rightarrow Dekker \models ME$$

we know $Dekker \models ME$.

Several points about the abstraction of the algorithm should be emphasised. First, original algorithm *Dekker* and abstract algorithm $Dekker_3$ are not observation equivalent. Therefore the abstract algorithm cannot be derived solely by application of the algebraic laws for observation congruence.

Secondly, $Dekker_3$ was derived from *Dekker* without the use of the *expansion law* (Prop. 5 of [72]). This algebraic law of CCS reduces concurrency to choice, thus changing the static structure of a process. It is commonly used when proving observation equivalence using algebraic laws. However, the theorem is cumbersome to apply as one must examine all possible synchronisation between a set of components. The algebraic laws for $\ll$ of Chapter 6 all preserve the static structure of processes.

Finally, the soundness of our abstraction of Dekker's algorithm does not depend on the assumptions we made about the shared variables in it. For example, our decision to hide actions of variable k was based on the assumption that k plays no part in ensuring mutual exclusion. Regardless of the truth of this assumption, the algorithm will satisfy the formula expressing mutual exclusion if the abstract version of it does. Since mutual exclusion does hold in the abstract algorithm, we know not only that it also holds in the concrete algorithm, but also that our assumption about variable k's role in mutual exclusion is correct.

7.1.2 Liveness

An important liveness property for mutual exclusion algorithms is that if one process requests entry to its critical section, then it will not wait forever while the other processes continue to enter their critical sections. Dekker's algorithm was designed to satisfy this property only under the fairness assumption that the requesting process continues to execute after it requests entry to its critical section.

Here we use abstraction to show that Dekker's algorithm satisfies the liveness property, with process 2 as the requesting process. We treat the fairness assumption by incorporating it into the formula we use to express the liveness property, not by revising the algorithm to make process 2 satisfy the assumption. However, process 2 does need to be revised, because the formula for liveness will need to express that process 2 is continuing to execute. We revise process 2 by adding probe action `p2`, which continues to occur while process P_2 is waiting to enter its critical section.

$$
\begin{aligned}
\mathit{P2} &\stackrel{\text{def}}{=} \overline{\mathtt{b_2wt}}\,.\,\mathtt{req_2}\,.\mathit{P21}\\
\mathit{P21} &\stackrel{\text{def}}{=} \mathtt{b_1rf}\,.\mathit{P23} + \mathtt{b_1rt}\,.\,\mathtt{p2}\,.(\mathtt{kr2}\,.\mathit{P21} + \mathtt{kr1}\,.\,\overline{\mathtt{b_2wf}}\,.\mathit{P22})\\
\mathit{P22} &\stackrel{\text{def}}{=} \mathtt{kr2}\,.\,\overline{\mathtt{b_2wt}}\,.\mathit{P21} + \mathtt{kr1}\,.\,\mathtt{p2}\,.\tau.\mathit{P22}\\
\mathit{P23} &\stackrel{\text{def}}{=} \mathtt{enter_2}\,.\,\mathtt{exit_2}\,.\,\overline{\mathtt{kw1}}\,.\,\overline{\mathtt{b_2wf}}\,.\mathit{P2}
\end{aligned}
$$

The following $\mu\mathbb{L}\Box$ formula expresses the property that process 2 will not wait forever to enter its critical section while process 1 continues to enter its critical section, provided that process 2 continues to execute while waiting.

$$
\begin{aligned}
\mathit{Live} \stackrel{\text{def}}{=} \nu X.[\![-\,\mathtt{req}_2]\!]X \wedge\\
&[\![\mathtt{req}_2]\!]\mu Y.\nu X_1.[\![-\,\mathtt{enter}_1,\mathtt{enter}_2,\mathtt{p2}]\!]X_1 \wedge [\![\mathtt{enter}_2]\!]X \wedge [\![\mathtt{enter}_1]\!]\\
&\qquad \nu X_2.[\![-\,\mathtt{enter}_1,\mathtt{enter}_2,\mathtt{p2}]\!]X_2 \wedge [\![\mathtt{enter}_2]\!]X \wedge [\![\mathtt{p2}]\!]Y
\end{aligned}
$$

Paraphrasing the formula gives: "after $\mathtt{req}_2$ occurs, no path containing infinitely many $\mathtt{enter}_1$ and `p2` actions but no $\mathtt{enter}_2$ actions, can occur". The informal argument that Dekker's algorithm satisfies this property is intricate. Process 1 will only enter its critical section if process 2's request variable is false, and will set the turn variable to 2 upon leaving its critical section. If Process 2 has requested entry but not yet entered its critical section, and the turn variable is set to 2, then it will eventually set its request variable to true and keep it true until after it exits its critical section.

Formula *Live*, unlike formula *ME*, is asymmetric with respect to process indices. Consequently, processes 1 and 2 will be abstracted in different ways. We first hide both exit actions and action $\mathtt{req}_1$. We simplify $\mathit{Dekker}\backslash\!\backslash\{\mathtt{exit}_1, \mathtt{exit}_2, \mathtt{req}_1\}$ by hiding laws 6.3 and 6.4 to obtain the following family of definitions. We refer to *Dekker* of this family as Dekker_1.

$$
\begin{array}{rcl}
\mathit{P1} & \stackrel{\text{def}}{=} & \overline{\mathtt{b_1wt}}\,.\tau.\mathit{P11} \\
\mathit{P11} & \stackrel{\text{def}}{=} & \mathtt{b_2rf}\,.\mathit{P13} + \mathtt{b_2rt}\,.(\mathtt{kr1}\,.\mathit{P11} + \mathtt{kr2}\,.\overline{\mathtt{b_1wf}}\,.\mathit{P12}) \\
\mathit{P12} & \stackrel{\text{def}}{=} & \mathtt{kr1}\,.\overline{\mathtt{b_1wt}}\,.\mathit{P11} + \mathtt{kr2}\,.\tau.\mathit{P12} \\
\mathit{P13} & \stackrel{\text{def}}{=} & \mathtt{enter_1}\,.\tau.\overline{\mathtt{kw2}}\,.\overline{\mathtt{b_1wf}}\,.\mathit{P1}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{P2} & \stackrel{\text{def}}{=} & \overline{\mathtt{b_2wt}}\,.\mathtt{req_2}\,.\mathit{P21} \\
\mathit{P21} & \stackrel{\text{def}}{=} & \mathtt{b_1rf}\,.\mathit{P23} + \mathtt{b_1rt}\,.\mathtt{p2}\,.(\mathtt{kr2}\,.\mathit{P21} + \mathtt{kr1}\,.\overline{\mathtt{b_2wf}}\,.\mathit{P22}) \\
\mathit{P22} & \stackrel{\text{def}}{=} & \mathtt{kr2}\,.\overline{\mathtt{b_2wt}}\,.\mathit{P21} + \mathtt{kr1}\,.\mathtt{p2}\,.\tau.\mathit{P22} \\
\mathit{P23} & \stackrel{\text{def}}{=} & \mathtt{enter_2}\,.\tau.\overline{\mathtt{kw1}}\,.\overline{\mathtt{b_2wf}}\,.\mathit{P2}
\end{array}
$$

It will be enough to show that $\mathit{Dekker}_1 \models \mathit{Live}$, because

$$
\begin{array}{lll}
 & \mathit{Dekker}_1 \models \mathit{Live} & \\
\Leftrightarrow & \mathit{Dekker}\backslash\!\backslash\{\mathtt{exit}_1, \mathtt{exit}_2, \mathtt{req}_1\} \models \mathit{Live} & \text{by def. of } \mathit{Dekker}_1 \\
\Rightarrow & \mathit{Dekker} \models \mathit{Live} & \text{by Prop. 5.11}
\end{array}
$$

Next we hide the reads of variables b_1 and b_2 that give value *true*. The result of hiding these actions is that the processes can continue to loop after making a request, even if the other process has not made a request. We apply derived law 6.15 with actions $\mathtt{b_1rt}$ and $\mathtt{b_2rt}$ (and their complements), and then move hiding inward with the hiding laws.

$$
\mathit{B1f} \stackrel{\text{def}}{=} \overline{\mathtt{b_1rf}}\,.\mathit{B1f} + \mathtt{b_1wf}\,.\mathit{B1f} + \mathtt{b_1wt}\,.\mathit{B1t}
$$

$$B1t \stackrel{\text{def}}{=} \tau.B1t + \mathtt{b_1wf}\,.B1f + \mathtt{b_1wt}\,.B1t$$

$$B2f \stackrel{\text{def}}{=} \overline{\mathtt{b_2rf}}\,.B2f + \mathtt{b_2wf}\,.B2f + \mathtt{b_2wt}\,.B2t$$
$$B2t \stackrel{\text{def}}{=} \tau.B2t + \mathtt{b_2wf}\,.B2f + \mathtt{b_2wt}\,.B2t$$

$$P1 \stackrel{\text{def}}{=} \overline{\mathtt{b_1wt}}\,.\tau.P11$$
$$P11 \stackrel{\text{def}}{=} \mathtt{b_2rf}\,.P13 + \tau.(\mathtt{kr1}\,.P11 + \mathtt{kr2}\,.\overline{\mathtt{b_1wf}}\,.P12)$$
$$P12 \stackrel{\text{def}}{=} \mathtt{kr1}\,.\overline{\mathtt{b_1wt}}\,.P11 + \mathtt{kr2}\,.\tau.P12$$
$$P13 \stackrel{\text{def}}{=} \mathtt{enter_1}\,.\tau.\overline{\mathtt{kw2}}\,.\overline{\mathtt{b_1wf}}\,.P1$$

$$P2 \stackrel{\text{def}}{=} \overline{\mathtt{b_2wt}}\,.\mathtt{req_2}\,.P21$$
$$P21 \stackrel{\text{def}}{=} \mathtt{b_1rf}\,.P23 + \tau.\mathtt{p2}\,.(\mathtt{kr2}\,.P21 + \mathtt{kr1}\,.\overline{\mathtt{b_2wf}}\,.P22)$$
$$P22 \stackrel{\text{def}}{=} \mathtt{kr2}\,.\overline{\mathtt{b_2wt}}\,.P21 + \mathtt{kr1}\,.\mathtt{p2}\,.\tau.P22$$
$$P23 \stackrel{\text{def}}{=} \mathtt{enter_2}\,.\tau.\overline{\mathtt{kw1}}\,.\overline{\mathtt{b_2wf}}\,.P2$$

Now we hide all actions of variable b_1. The effect is to allow process 2 to enter its critical section even if process 1 has requested entrance to its own critical section. Applying derived law 6.15 with actions $\mathtt{b_1rf}$, $\mathtt{b_1wt}$, and $\mathtt{b_1wf}$ (and their complements), and then moving hiding inwards with the hiding laws gives:

$$B1f \stackrel{\text{def}}{=} \tau.B1f + \tau.B1f + \tau.B1t$$
$$B1t \stackrel{\text{def}}{=} \tau.B1t + \tau.B1f + \tau.B1t$$

$$P1 \stackrel{\text{def}}{=} \tau.\tau.P11$$
$$P11 \stackrel{\text{def}}{=} \mathtt{b_2rf}\,.P13 + \tau.(\mathtt{kr1}\,.P11 + \mathtt{kr2}\,.\tau.P12)$$
$$P12 \stackrel{\text{def}}{=} \mathtt{kr1}\,.\tau.P11 + \mathtt{kr2}\,.\tau.P12$$
$$P13 \stackrel{\text{def}}{=} \mathtt{enter_1}\,.\tau.\overline{\mathtt{kw2}}\,.\tau.P1$$

$$P2 \stackrel{\text{def}}{=} \overline{\mathtt{b_2wt}}\,.\mathtt{req_2}\,.P21$$

$$
\begin{aligned}
P21 &\stackrel{\text{def}}{=} \tau.P23 + \tau.\,\mathtt{p2}\,.(\mathtt{kr2}\,.P21 + \mathtt{kr1}\,.\overline{\mathtt{b_2wf}}\,.P22)\\
P22 &\stackrel{\text{def}}{=} \mathtt{kr2}\,.\overline{\mathtt{b_2wt}}\,.P21 + \mathtt{kr1}\,.\,\mathtt{p2}\,.\tau.P22\\
P23 &\stackrel{\text{def}}{=} \mathtt{enter_2}\,.\tau.\overline{\mathtt{kw1}}\,.\overline{\mathtt{b_2wf}}\,.P2
\end{aligned}
$$

Next reads of variable `k` of value 2 are hidden. The significance of this step is that the processes will be able to proceed as if it is process 2's turn even if it process 1's turn. Applying derived law 6.15 with actions `kr2` and $\overline{\mathtt{kr2}}$, and then moving hiding inward with the hiding laws gives:

$$
\begin{aligned}
K1 &\stackrel{\text{def}}{=} \overline{\mathtt{kr1}}\,.K1 + \mathtt{kw1}\,.K1 + \mathtt{kw2}\,.K2\\
K2 &\stackrel{\text{def}}{=} \tau.K2 + \mathtt{kw1}\,.K1 + \mathtt{kw2}\,.K2
\end{aligned}
$$

$$
\begin{aligned}
P1 &\stackrel{\text{def}}{=} \tau.\tau.P11\\
P11 &\stackrel{\text{def}}{=} \mathtt{b_2rf}\,.P13 + \tau.(\mathtt{kr1}\,.P11 + \tau.\tau.P12)\\
P12 &\stackrel{\text{def}}{=} \mathtt{kr1}\,.\tau.P11 + \tau.\tau.P12\\
P13 &\stackrel{\text{def}}{=} \mathtt{enter_1}\,.\tau.\overline{\mathtt{kw2}}\,.\tau.P1
\end{aligned}
$$

$$
\begin{aligned}
P2 &\stackrel{\text{def}}{=} \overline{\mathtt{b_2wt}}\,.\,\mathtt{req_2}\,.P21\\
P21 &\stackrel{\text{def}}{=} \tau.P23 + \tau.\,\mathtt{p2}\,.(\tau.P21 + \mathtt{kr1}\,.\overline{\mathtt{b_2wf}}\,.P22)\\
P22 &\stackrel{\text{def}}{=} \tau.\overline{\mathtt{b_2wt}}\,.P21 + \mathtt{kr1}\,.\,\mathtt{p2}\,.\tau.P22\\
P23 &\stackrel{\text{def}}{=} \mathtt{enter_2}\,.\tau.\overline{\mathtt{kw1}}\,.\overline{\mathtt{b_2wf}}\,.P2
\end{aligned}
$$

We remove all τ actions by repeatedly applying basic law 6.12, and remove unguarded constants by applying family law 6.10.

$$
\begin{aligned}
B2f &\stackrel{\text{def}}{=} \overline{\mathtt{b_2rf}}\,.B2f + \mathtt{b_2wf}\,.B2f + \mathtt{b_2wt}\,.B2t\\
B2t &\stackrel{\text{def}}{=} \mathtt{b_2wf}\,.B2f + \mathtt{b_2wt}\,.B2t
\end{aligned}
$$

$$
K1 \stackrel{\text{def}}{=} \overline{\mathtt{kr1}}\,.K1 + \mathtt{kw1}\,.K1 + \mathtt{kw2}\,.K2
$$

$$K2 \stackrel{\text{def}}{=} \texttt{kw1}\,.K1 + \texttt{kw2}\,.K2$$

$$\begin{aligned}
P1 &\stackrel{\text{def}}{=} P11\\
P11 &\stackrel{\text{def}}{=} \texttt{b}_2\texttt{rf}\,.P13 + \texttt{kr1}\,.P11 + P12\\
P12 &\stackrel{\text{def}}{=} \texttt{kr1}\,.P11\\
P13 &\stackrel{\text{def}}{=} \texttt{enter}_1\,.\overline{\texttt{kw2}}\,.P1
\end{aligned}$$

$$\begin{aligned}
P2 &\stackrel{\text{def}}{=} \overline{\texttt{b}_2\texttt{wt}}\,.\texttt{req}_2\,.P21\\
P21 &\stackrel{\text{def}}{=} P23 + \texttt{p2}\,.(P21 + \texttt{kr1}\,.\overline{\texttt{b}_2\texttt{wf}}\,.P22)\\
P22 &\stackrel{\text{def}}{=} \overline{\texttt{b}_2\texttt{wt}}\,.P21 + \texttt{kr1}\,.\texttt{p2}\,.P22\\
P23 &\stackrel{\text{def}}{=} \texttt{enter}_2\,.\overline{\texttt{kw1}}\,.\overline{\texttt{b}_2\texttt{wf}}\,.P2
\end{aligned}$$

Figure 7–3 shows the partly-abstracted $P1$. In line with our informal explanation of why Dekker's algorithm satisfies the liveness property, the process will not enter its critical section unless process 2's request variable is false, and sets the turn variable to 2 after exiting its critical section. By merging constants $P1$, $P11$, and $P12$ through the application of family law 6.9 we can simplify the process expression for $P1$. This operation does not reduce the state space of the process, however.

$$\begin{aligned}
P1 &\stackrel{\text{def}}{=} \texttt{b}_2\texttt{rf}\,.P13 + \texttt{kr1}\,.P1 + P1\\
P13 &\stackrel{\text{def}}{=} \texttt{enter}_1\,.\overline{\texttt{kw2}}\,.P1
\end{aligned}$$

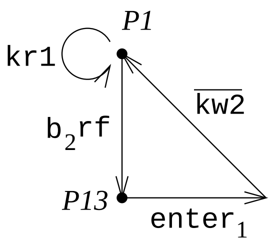


Figure 7–3: Transition Diagram for Partly-Abstracted Process *P1*

Figure 7–4 shows the partly-abstracted $P2$. The dotted lines indicate the states to be merged. Our informal understanding of $P2$'s role in liveness is that, after requesting entry to its critical section, and once the turn variable is set to 2, it will eventually set its request variable to true and keep it there until it enters its critical section. The set of states including $P2$ can be merged because $P2$'s behaviour between entering its critical section and requesting its next entry is unimportant except that its request variable must be set to true before requesting. The set of states including $P21$ can be merged because there is no difference between them that affects the request variable or turn variable.

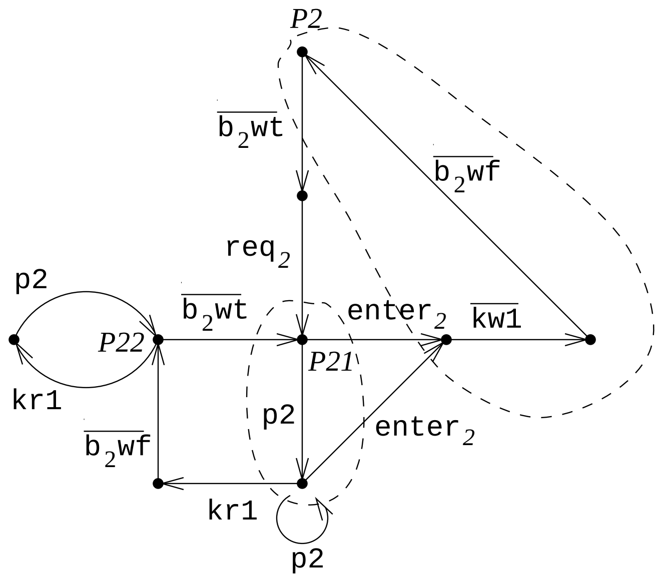


Figure 7–4: Transition Diagram for Partly-Abstracted Process *P2*

To prepare for the merging we introduce new constants *P24*, *P25*, and *P26* by applying basic law 6.14.

$$
\begin{aligned}
P2 &\stackrel{\text{def}}{=} \overline{\mathtt{b_2wt}}\,.\,\mathtt{req_2}\,.P21 \\
P21 &\stackrel{\text{def}}{=} P23 + \mathtt{p2}\,.P26 \\
P22 &\stackrel{\text{def}}{=} \overline{\mathtt{b_2wt}}\,.P21 + \mathtt{kr1}\,.\,\mathtt{p2}\,.P22 \\
P23 &\stackrel{\text{def}}{=} \mathtt{enter_2}\,.P24 \\
P24 &\stackrel{\text{def}}{=} \overline{\mathtt{kw1}}\,.P25 \\
P25 &\stackrel{\text{def}}{=} \overline{\mathtt{b_2wf}}\,.P2 \\
P26 &\stackrel{\text{def}}{=} P21 + \mathtt{kr1}\,.\,\overline{\mathtt{b_2wf}}\,.P22
\end{aligned}
$$

We apply family law 6.9 to merge *P21* and *P26*, and to merge *P2*, *P24*, and *P25*.

$$
\begin{array}{rcl}
\mathit{P2} & \stackrel{\mathrm{def}}{=} & \overline{\mathtt{b_2wt}}\,.\,\mathtt{req_2}\,.\mathit{P21} + \overline{\mathtt{kw1}}\,.\mathit{P2} + \overline{\mathtt{b_2wf}}\,.\mathit{P2} \\
\mathit{P21} & \stackrel{\mathrm{def}}{=} & \mathit{P23} + \mathtt{p2}\,.\mathit{P21} + \mathit{P21} + \mathtt{kr1}\,.\,\overline{\mathtt{b_2wf}}\,.\mathit{P22} \\
\mathit{P22} & \stackrel{\mathrm{def}}{=} & \overline{\mathtt{b_2wt}}\,.\mathit{P21} + \mathtt{kr1}\,.\,\mathtt{p2}\,.\mathit{P22} \\
\mathit{P23} & \stackrel{\mathrm{def}}{=} & \mathtt{enter_2}\,.\mathit{P2}
\end{array}
$$

Finally, unguarded constants *B1f*, *B1t*, *P1*, and *P21* are removed by applying family law 6.10. We also eliminate constant *P23* by basic law 6.13. We refer to *Dekker* of the resulting family as Dekker_2.

$$
\begin{array}{rcl}
\mathit{B2f} & \stackrel{\mathrm{def}}{=} & \overline{\mathtt{b_2rf}}\,.\mathit{B2f} + \mathtt{b_2wf}\,.\mathit{B2f} + \mathtt{b_2wt}\,.\mathit{B2t} \\
\mathit{B2t} & \stackrel{\mathrm{def}}{=} & \mathtt{b_2wf}\,.\mathit{B2f} + \mathtt{b_2wt}\,.\mathit{B2t}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{K1} & \stackrel{\mathrm{def}}{=} & \overline{\mathtt{kr1}}\,.\mathit{K1} + \mathtt{kw1}\,.\mathit{K1} + \mathtt{kw2}\,.\mathit{K2} \\
\mathit{K2} & \stackrel{\mathrm{def}}{=} & \mathtt{kw1}\,.\mathit{K1} + \mathtt{kw2}\,.\mathit{K2}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{P1} & \stackrel{\mathrm{def}}{=} & \mathtt{b_2rf}\,.\mathit{P13} + \mathtt{kr1}\,.\mathit{P1} \\
\mathit{P13} & \stackrel{\mathrm{def}}{=} & \mathtt{enter_1}\,.\,\overline{\mathtt{kw2}}\,.\mathit{P1}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{P2} & \stackrel{\mathrm{def}}{=} & \overline{\mathtt{b_2wt}}\,.\,\mathtt{req_2}\,.\mathit{P21} + \overline{\mathtt{kw1}}\,.\mathit{P2} + \overline{\mathtt{b_2wf}}\,.\mathit{P2} \\
\mathit{P21} & \stackrel{\mathrm{def}}{=} & \mathtt{enter_2}\,.\mathit{P2} + \mathtt{p2}\,.\mathit{P21} + \mathtt{kr1}\,.\,\overline{\mathtt{b_2wf}}\,.\mathit{P22} \\
\mathit{P22} & \stackrel{\mathrm{def}}{=} & \overline{\mathtt{b_2wt}}\,.\mathit{P21} + \mathtt{kr1}\,.\,\mathtt{p2}\,.\mathit{P22}
\end{array}
$$

$$
\mathit{Dekker}_2 \stackrel{\mathrm{def}}{=} (\mathit{P1} \mid \mathit{P2} \mid \mathit{B2f} \mid \mathit{K1})\backslash\{\mathtt{b_2},\mathtt{k}\}
$$

The CWB verifies that Dekker_2 satisfies property *Live*, and therefore *Dekker* satisfies it as well. The abstraction gives a modest reduction in state space from

177 to 41 states. Also important is that the process expressions in the definition of $Dekker_2$ are smaller than in $Dekker$.

7.2 Discussion

Similar patterns were followed in abstracting Dekker's algorithm relative to the safety and to the liveness property. First actions in the sort of the algorithm were hidden and relabelled using the Correspondence Rule. Then particular synchronisation actions were hidden using a derived law for $\ll$. The hiding was moved inwards using the hiding laws. Next τ actions were removed, and the resulting unguarded constants were eliminated. Finally constants were introduced and merging was performed.

We have also abstracted Dijkstra's [34] and Knuth's [58] mutual exclusion algorithms. These algorithms are more complicated that Dekker's algorithm and have larger state spaces. For example, while the two-process version of Dekker's algorithm has 153 states, a two-process version of Dijkstra's algorithm has about 3600 states. However, the same general pattern was followed in abstracting these algorithms. Furthermore, abstraction makes it clear that the algorithms use the same idea as Dekker's algorithm to achieve mutual exclusion. In each case, the abstract algorithm (relative to the property of mutual exclusion) has just 16 states.

To describe these algorithms in their n-process form requires the use of value-passing CCS. We have written algebraic laws for $\ll$ and value-passing processes, and have abstracted an n-process form of Dekker's algorithm using them. However, the work is not satisfactory because the abstraction operations on value-passing processes are clumsy. For example, in abstracting Dekker's algorithm we saw that it is sometimes useful to hide reads of a variable that have a certain value. We have not found an elegant way to define such an operation on value-passing CCS processes.

A related goal is to define abstraction operations for processes written in a concurrent while language. The problem here is that the language does not have the constructs needed to write the algebraic laws that we want. For example, it is not clear how one could hide true reads of variable b_2 in the concurrent while form of Dekker's algorithm. This problem can be overcome by adding new language constructs, but we have not been able to find extensions that seem simple and natural. This is a serious problem, as we would like algorithm designers to use abstraction directly in their design language.

Abstraction with the Correspondence Rule and the weakly simulates relation are complementary. The Correspondence Rule does not support any direct algebraic manipulation of a process, but does allow the sort of the process to be changed. With the technique of the previous chapter the reverse is true. In abstracting Dekker's algorithm we used first the Correspondence Rule to hide or relabel actions in the sort of the process, and then used the technique of the previous chapter to hide or relabel synchronisation actions.

Chapter 8

Abstraction with Priorities

In this chapter we present another abstraction technique based on a process preorder. The preorder used here is the ready simulates preorder, which is slightly stronger than the simulates preorder. We show that this preorder is characterized by the logic obtained by extending $\mu L\Box$ with $\langle - \rangle\, \mathtt{tt}$ formulas. The sole abstraction operation we define here is to add priorities to the actions of a process. We also present algebraic laws that allow prioritized processes to be simplified. We illustrate the technique by proving that a version of Ben-Ari's concurrent garbage collection algorithm [5] is unsafe, and by proving that Dekker's mutual exclusion algorithm fails to satisfy a liveness property.

8.1 Abstracting to Demonstrate Errors

The technique of Chapter 6 allows one to prove that a process satisfies a $\mu\mathbb{L}\Box$ formula by proving that an abstract version of the process satisfies it. A limitation of the technique is that it works only when the concrete process satisfies the formula. If the concrete process does not satisfy the formula then neither will any abstract version of the process. Furthermore, if the abstract version does not satisfy the formula one cannot conclude the that concrete version does not satisfy it. It may be that relevant features of the concrete version were abstracted away. This is a serious limitation because we commonly discover during verification that a process does not satisfy a formula we believed it did satisfy.

To show that a concrete process fails to satisfy a $\mu\mathbb{L}\Box$ formula we can use the contrapositive of Theorem 4.15. We take as abstract any process that is weakly simulated by the concrete process. If the abstract process fails to satisfy a $\mu\mathbb{L}\Box$ formula then the concrete process will also fail to satisfy it. It is easy to invent abstraction operations that produce a process weakly simulated by the concrete process. Replacing an arbitrary process term by 0 is such an operation.

In this chapter we use this approach but with the anonymous ready simulates preorder, which is slightly stronger than the simulates preorder. To abstract a process we add priorities to its actions. We first present a version of CCS with priorities. We then define anonymous ready simulation and show that it is characterized by a logic slightly more expressive than $\mu L\Box$. We show that adding priorities to a process yields a process that is anonymous ready-simulated by the original process. We also present algebraic laws for processes with priorities.

We demonstrate the technique using Ben-Ari's algorithm for concurrent garbage collection. Ben-Ari claimed that his algorithm satisfied its main safety property even if the two actions of a certain process of the algorithm were transposed. The claim has been shown to be false [81,76], but we were not able to verify this because models of the algorithm with enough detail to expose the problem have so many states that they cannot be checked with the Concurrency Workbench. By adding priorities we were able to obtain a model with about 6500 states, which was readily checked.

For a second example we use Dekker's mutual exclusion algorithm. A basic liveness property we would like the algorithm to satisfy is that if a process requests entry to its critical section then it eventually does enter it. Using priorities we show that the algorithm fails to satisfy this property. We include this example because, unlike Ben-Ari's algorithm, it could not have been abstracted using priorities and the simulates preorder.

8.2 CCS with Priorities

8.2.1 Basic PCCS

The version of CCS with priorities (PCCS) presented here is based on [26] and [27]. Like the priority operator of Section 4.6, the idea is that each action has an associated priority. However, here each action is tagged explicitly with a natural number, with a lower number indicating a higher priority. Recall that in basic CCS the set of all labels is written $\mathcal{L}$ and the set of all actions is written Act. In PCCS the set $\{l:k \mid l \in \mathcal{L} \text{ and } k \in I\!\!N\}$ of all prioritized labels is written $\mathcal{L}_p$ and the set $\{\alpha:k \mid \alpha \in Act \text{ and } k \in I\!\!N\}$ of all prioritized actions is written Act_p.

The relabelling functions of CCS are extended to PCCS by the rule $f(\alpha:k) = f(\alpha):k$. Also, as for CCS we extend complementation to sets of prioritized labels: if $L \subseteq \mathcal{L}_p$ then $\overline{L} = \{\bar{l}:k \mid l:k \in L\}$.

The abstract syntax of PCCS processes is defined as follows:

$$E ::= \alpha:k.E \;\Big|\; \sum_{i \in I} E_i \;\Big|\; E \mid F \;\Big|\; E\backslash L \;\Big|\; E[f] \;\Big|\; A$$

where k ranges over $I\!\!N$, I is an indexing set, L ranges over sets of prioritized labels, f ranges over relabelling functions, and A ranges over process constants. We write the set of all PCCS processes as $\mathcal{P}_p$ and let $E, F, \ldots$ range over it.

The meaning of PCCS processes is given as the LTS

$$(\mathcal{P}_p, Act_p, \{\xrightarrow{\alpha:k}_p \mid \alpha:k \in Act_p\})$$

where $\xrightarrow{\alpha:k}_p \subseteq \mathcal{P}_p \times \mathcal{P}_p$ for all $\alpha:k$ in Act_p. The transition relations are defined inductively as in CCS. However, the semantics of PCCS is given in two layers. The first layer defines the initial transitions of a PCCS process under the assumption that priorities have no significance. The second level defines the PCCS transition relation taking priorities into account. The general use of multi-layered semantics for process algebras with negative side conditions is described in [48]. A non-layered semantics for PCCS is given in [26], along with the two-layered version adopted here.

The following rules define the initial transitions $I_k(E)$ of a PCCS process E with priorities not taken into account:

$$
\begin{array}{lcll}
. & : & j = k & \Rightarrow \ \alpha{:}j \in I_k(\alpha{:}j.E) \\
\sum & : & \alpha{:}j \in I_k(E_i) \text{ for some } i \text{ in } I & \Rightarrow \ \alpha{:}j \in I_k(\sum_{i\in I} E_i) \\
\mid & : & \alpha{:}j \in I_k(E) & \Rightarrow \ \alpha{:}j \in I_k(E \mid F) \\
 & & \alpha{:}j \in I_k(F) & \Rightarrow \ \alpha{:}j \in I_k(E \mid F) \\
 & & l{:}j \in I_k(E) \text{ and } \overline{l}{:}j \in I_k(F) & \Rightarrow \ \tau{:}j \in I_k(E \mid F) \\
\backslash & : & \alpha{:}j \in I_k(E) - (L \cup \overline{L}) & \Rightarrow \ \alpha{:}j \in I_k(E\backslash L) \\
f & : & \alpha{:}j \in I_k(E) & \Rightarrow \ f(\alpha{:}j) \in I_k(E[f]) \\
\text{def} & : & \alpha{:}j \in I_k(E) \text{ and } A \stackrel{\text{def}}{=} E & \Rightarrow \ \alpha{:}j \in I_k(A)
\end{array}
$$

Note that $\alpha : j \in I_k(E)$ implies $j = k$, and that $\tau{:}k \in I_k(E)$ does not imply that E or some process reachable from E contains a τ action with priority k. We write $I_{<k}(E)$ for $\bigcup_{j<k} I_j(E)$ and write $\tau \notin I_{<k}(E)$ if $\tau{:}j \notin I_{<k}(E)$ for all j.

The transition rules for PCCS are as follows:

$$. \ : \ \alpha{:}k.E \xrightarrow{\alpha:k}_p E$$

$$\sum \ : \ \frac{E_j \xrightarrow{\alpha:k}_p E'_j}{\sum_{i\in I} E_i \xrightarrow{\alpha:k}_p E'_j} \ j \in I \text{ and } \tau \notin I_{<k}(E_i) \text{ for } i \in (I - \{j\})$$

$$\mid \ : \ \frac{E \xrightarrow{\alpha:k}_p E'}{E \mid F \xrightarrow{\alpha:k}_p E' \mid F} \ \tau \notin I_{<k}(E \mid F) \quad \frac{F \xrightarrow{\alpha:k}_p F'}{E \mid F \xrightarrow{\alpha:k}_p E \mid F'} \ \tau \notin I_{<k}(E \mid F)$$

$$\frac{E \xrightarrow{l:k}_p E' \quad F \xrightarrow{\overline{l}:k}_p F'}{E \mid F \xrightarrow{\tau:k}_p E' \mid F'} \ \tau \notin I_{<k}(E \mid F)$$

$$\backslash \ : \ \frac{E \xrightarrow{\alpha:k}_p E'}{E\backslash L \xrightarrow{\alpha:k}_p E'\backslash L} \ \alpha{:}k \notin L \cup \overline{L}$$

$$f \quad : \quad \frac{E \xrightarrow{\alpha:k}_p E'}{E[f] \xrightarrow{f(\alpha:k)}_p E'[f]}$$

$$\text{def} \quad : \quad \frac{E \xrightarrow{\alpha:k}_p E'}{A \xrightarrow{\alpha:k}_p E'} \; A \stackrel{\text{def}}{=} E$$

These rules are like the rules for CCS except for the negative side conditions, which prohibit a transition of priority k if a higher-priority τ action is enabled. Note that a non-τ transition cannot prevent a transition with a lower priority action from occurring. For example, process $a\!:\!1.E + b\!:\!2.F$ has a $b\!:\!2$ transition to F.

The syntactic sort $\mathcal{L}(E)$ of a PCCS process E is defined as in CCS, except that $\mathcal{L}(l\!:\!k.E) = \{l\!:\!k\} \cup \mathcal{L}(E)$ and $\mathcal{L}(\tau\!:\!k.E) = \mathcal{L}(E)$.

Two PCCS processes are strongly equivalent if they are strongly equivalent relative to the PCCS labelled transition system. Consequently an action of one must be matched by an action of the other having the same priority.

Definition 8.1 *Two PCCS processes are* strongly equivalent*, written $E \sim_p F$, if they are bisimulation equivalent relative to the LTS of PCCS.*

Theorem 8.1 *The strong equivalence of PCCS processes is a congruence.*

Proof. See Theorem 4.2 of [26].

8.2.2 Value-Passing PCCS

A *value-passing PCCS* process is a value-passing CCS process that is interpreted with respect to a mapping P from actions to priority values, with $P(l) = P(\bar{l})$ for all labels l. The syntax of value-passing PCCS is just the syntax of value-passing CCS. The semantics is defined by translation to basic PCCS. We write (E, D, P) for the basic PCCS process obtained from a closed value-passing PCCS process E relative to data domain D and priority mapping P. We define (E, D, P) as follows.

E	(E, D, P)
$\tau.F$	$\tau\!:\!P(\tau).(F, D, P)$
$a(x).F$	$\sum_{v \in D(a)} a(v)\!:\!P(a).(F\{v/x\}, D, P)$
$\overline{a}(e).F$	$\overline{a}(\llbracket e \rrbracket)\!:\!P(a).(F, D, P)$
$\sum_{i \in I} F_i$	$\sum_{i \in I}(F_i, D, P)$
$F_1 \mid F_2$	$(F_1, D, P) \mid (F_2, D, P)$
$F \backslash L$	$(F, D, P)\backslash\{l(v)\!:\!P(l) \mid v \in D(l) \text{ and } l \in L\}$
$F[f]$	$(F, D, P)[\hat{f}]$, where $\hat{f}(l(v)\!:\!k) \stackrel{\text{def}}{=} (f(l))(v)\!:\!k$
$A(e_1, \ldots, e_n)$	$A(v_1, \ldots, v_n)$, where $\llbracket e_i \rrbracket = v_i$ $(1 \leq i \leq n)$
if b then F_1 else F_2	$\begin{cases} (F_1, D, P) & \text{if } \llbracket b \rrbracket = \text{true} \\ (F_2, D, P) & \text{otherwise} \end{cases}$

Process definitions are translated just as in value-passing CCS. Notice that for any value-passing PCCS process E, data domain D, and priority mapping P, all instances of an action α in (E, D, P) have the same priority, even if α is τ.

When data domain D and priority mapping P are understood, we often write E as shorthand for (E, D, P) when considering the behaviour of a value-passing PCCS process E. For example, we write $E \xrightarrow{\alpha:i}_p$ instead of $(E, D, P) \xrightarrow{\alpha:i}_p$ and write $E \sim_p F$ instead of $(E, D_E, P_E) \sim_p (F, D_F, P_F)$.

We allow unparametrised actions in value-passing processes. Let a be an action and let $D(a)$ be a set with a single arbitrary value v. Then we write $a.E$ as an abbreviation for $a(x).E$, where x is not free in E. Similarly we write $\overline{a}.E$ as an abbreviation for $\overline{a}(v).E$.

8.3 Anonymous Ready Simulation

Definition 8.2 *Let $(S, A, \{\xrightarrow{a} \mid a \in A\})$ be an LTS. A binary relation $\mathcal{R}$ on S is an* anonymous ready simulation *if $(E, F) \in \mathcal{R}$ implies, for all a in A:*

1. *Whenever $E \xrightarrow{a} E'$, then $F \xrightarrow{a} F'$ for some F' such that $(E', F') \in \mathcal{R}$,*

2. *Whenever* $E \not\rightarrow$ *then* $F \not\rightarrow$.

Anonymous ready simulation is closely related to the ready simulation relation of [8] and the 2/3 bisimulation relation of [63]. In these relations clause 2 of Def. 8.2 is replaced with

2. *Whenever* $E \stackrel{a}{\not\rightarrow}$ *then* $F \stackrel{a}{\not\rightarrow}$

Definition 8.3 *Process* F ready-simulates E*, written* $E \preceq F$*, if there exists an anonymous ready simulation* $\mathcal{R}$ *such that* $(E, F) \in \mathcal{R}$*.*

The ready-simulates relation is reflexive and transitive, and hence a preorder. It is logically characterized by the fragment of μL^+ having the following abstract syntax:

$$\phi ::= X \;\Big|\; \phi_1 \wedge \phi_2 \;\Big|\; \phi_1 \vee \phi_2 \;\Big|\; [L]\phi \;\Big|\; \langle - \rangle\, \mathtt{tt} \;\Big|\; \nu X.\phi \;\Big|\; \mu X.\phi$$

where, as before, X ranges over propositional variables and L ranges over sets of actions. We write μLr for this logic because of its connection with the ready-simulates preorder. We first show that the ready-simulates preorder preserves formulas of μLr.

Theorem 8.2 *Let* $\mathcal{T} = (S, A, \{\xrightarrow{a} \mid a \in A\})$ *be an LTS, let* E *and* F *be processes in* S*, and let* ϕ *be a* μLr *formula. Then*

$$(F \models^{\mathcal{T}} \phi \text{ and } E \preceq F) \Rightarrow E \models^{\mathcal{T}} \phi.$$

Proof. The proof closely follows the proof of Theorem 3.4, which states that the simulation preorder preserves formulas of $\mu L\Box$. We describe only the changes that would have to be made to the proof of Theorem 3.4 and its preceding lemmas to obtain a proof here.

In Definition 3.2 the simulation preorder must be replaced by the ready-simulates preorder.

Lemma 3.3 must be extended with a fourth item stating that $\|\langle - \rangle\| S$ is a down-set. (Recall that S is the set of states in the LTS of the lemma.) This is easily proved. Suppose $E \in \|\langle - \rangle\| S$ and $F \preceq E$. Then $F \in \|\langle - \rangle\| S$ directly by definition of down set and the ready-simulates preorder.

Lemmas 3.4 – 3.6 are independent of the simulates preorder and need no modification.

Lemma 3.7 must be extended with case $\phi \equiv \langle - \rangle\, \mathtt{tt}$. This holds for item 1 of the lemma directly by the extension above to Lemma 3.3. It holds for items 2 and 3 by the same argument used in case $\phi \equiv \phi_1 \wedge \phi_2$.

In Theorem 3.4 the simulation preorder must be replaced by the ready-simulates preorder. $\square$

Next we show that process F ready-simulates process E if F satisfies more μLr formulas than E.

Proposition 8.1 *Let $\mathcal{T} = (S, A, \{\xrightarrow{a} \mid a \in A\})$ be an LTS and let E and F be image-finite processes in S. Then*

$$(\forall \phi \in \mu Lr . F \models^{\mathcal{T}} \phi \Rightarrow E \models^{\mathcal{T}} \phi) \Rightarrow (E \preceq F)$$

Proof. The proof here also closely follows the corresponding result for the simulation preorder. The proof of this proposition can be obtained from Theorem 3.5 by adding a new case 3: $E \not\rightarrow$ and $F \rightarrow$. If some pair (E, F) of $\mathcal{R}$ existed, then $F \models^{\mathcal{T}} \langle - \rangle\, \mathtt{tt}$ but $E \not\models^{\mathcal{T}} \langle - \rangle\, \mathtt{tt}$, contrary to the definition of $\mathcal{R}$ in that proof. $\square$

Thus the ready-simulates preorder is logically characterized by μLr for image-finite processes. Similar characterization results are given in [8] and [63] for ready simulation and 2/3 bisimulation. However, in both cases the logic used is essentially μL^{+} without fixed-point operators or propositional variables.

8.4 Abstraction Operations

The sole abstraction operation of this chapter is to add priorities to a process. In this section we show that a CCS process ready simulates a prioritized version of it in which all instances of a label have the same priority. We also present algebraic laws for PCCS processes.

Let $\mathcal{F} \stackrel{\text{def}}{=} \{A_i \stackrel{\text{def}}{=} E_i \mid i \in I\}$ be a family of PCCS processes. $\mathcal{F}$ is *uniformly prioritized* if $\{l\!:\!i, l\!:\!j\} \subseteq \bigcup_{i \in I} \mathcal{L}(E_i)$ implies that $i = j$. Furthermore, every A_i and E_i of $\mathcal{F}$ is uniformly prioritized if $\mathcal{F}$ is. We write $\mathrm{pri}(l)$ for the priority of label l in a uniformly prioritized process. The *unprioritized version* $\mathcal{F}' = \{A_i' \stackrel{\text{def}}{=} E_i' \mid i \in I\}$ of $\mathcal{F}$ is the family of CCS processes obtained from $\mathcal{F}$ by replacing each $\alpha\!:\!k$ occurring as a prefix or in a restriction in an E_i of $\mathcal{F}$ by α. Furthermore, we say that A_i' is the unprioritized version of A_i, written $\overline{A_i}$, and similarly for E_i' and $\overline{E_i}$.

By definition all value-passing PCCS processes are uniformly prioritized. Indeed, in a value-passing PCCS process all τ actions have the same priority even though this is not required for a process to be uniformly prioritized.

To show that a CCS process ready simulates any uniformly prioritized version of it, we show two things. First, we show that $E \xrightarrow{\alpha:k}_p E$ implies $\overline{E} \xrightarrow{\alpha} \overline{E'}$. This can be proved by a simple induction on the length of the inference of the transition $E \xrightarrow{\alpha:k}_p E'$. Notice that here E need not be uniformly prioritized.

Lemma 8.1 *Let E and E' be PCCS processes and k be a natural number. Then*

$$E \xrightarrow{\alpha:k}_p E' \Rightarrow \overline{E} \xrightarrow{\alpha} \overline{E'}.$$

Secondly, we show that if $\overline{E}$ can perform some action then so can E. The difficulty in showing this is in finding a suitable induction hypothesis. We obtain the result in two steps. The following lemma relates transitions of the CCS process $\overline{E}$ to the initial actions $I_k(E)$ of E. The subsequent lemma relates $I_k(E)$ to the transitions of the PCCS process E.

Lemma 8.2 *Let E be a uniformly prioritized PCCS process and α be an action. Then $\overline{E} \xrightarrow{\alpha}$ implies*

1. $\alpha{:}k \in I_k(E)$ *if α is a label with pri(α) $= k$, and*

2. $\tau{:}k \in I_k(E)$ *for some $k \in I\!N$ if $\alpha = \tau$.*

Proof. By straightforward induction on the depth of inference of $\overline{E} \xrightarrow{\alpha}$. We do case analysis on the structure of E, because E determines how transitions of $\overline{E}$ can be inferred.

Case $E \equiv A$**.** Here we assume $A \stackrel{\text{def}}{=} F$. Suppose $\overline{A} \xrightarrow{\alpha}$. Then $\overline{F} \xrightarrow{\alpha}$, and by induction $\alpha : k \in I_k(F)$ with $k = \text{pri}(\alpha)$ or $\tau : k \in I_k(F)$ for some k. Therefore $\alpha : k \in I_k(A)$ with $k = \text{pri}(\alpha)$ or $\tau{:}k \in I_k(A)$ for some k.

Case $E \equiv \alpha{:}k.E'$**.** Here $\alpha{:}k \in I_k(E)$ directly by the def. of $I_k()$.

Case $E \equiv \sum_{i\in I} E_i$**.** Suppose $\overline{\sum_{i\in I} E_i} \equiv \sum_{i\in I} \overline{E_i} \xrightarrow{\alpha}$. Suppose the transition is inferred by $\overline{E_i} \xrightarrow{\alpha}$ where $i \in I$. Then by induction $\alpha{:}k \in I_k(E_i)$ with $k = \text{pri}(\alpha)$ or $\tau{:}k \in I_k(E_i)$ for some k, and therefore $\alpha{:}k \in I_k(\sum_{i\in I} E_i)$ with $k = \text{pri}(\alpha)$ or $\tau{:}k \in I_k(\sum_{i\in I} E_i)$ for some k by definition of $I_k()$.

Case $E \equiv E_1 \mid E_2$**.** Suppose $\overline{E_1 \mid E_2} \equiv \overline{E_1} \mid \overline{E_2} \xrightarrow{\alpha}$. There are three ways to infer this transition. Suppose $\overline{E_1} \xrightarrow{l}$ and $\overline{E_2} \xrightarrow{\overline{l}}$. Then by induction $l : k \in I_k(E_1)$ and $\overline{l} : k \in I_k(E_2)$ with $k = \text{pri}(l)$, and therefore $\tau : k \in I_k(E_1 \mid E_2)$. In the second and third cases the transition is inferred from the independent action of E_1 or E_2. These cases are similar to the $E \equiv E_1 + E_2$ case.

Case $E \equiv E'\backslash L$**.** Suppose $\overline{E'\backslash L} \equiv \overline{E'}\backslash\{\alpha \mid \alpha{:}k \in L \text{ and } k \in I\!N\} \xrightarrow{\alpha}$. Then $\overline{E'} \xrightarrow{\alpha}$ and $\alpha, \overline{\alpha} \notin \{\alpha \mid \alpha : k \in L \text{ and } k \in I\!N\}$. By induction $\alpha : k \in I_k(E')$ with $k = \text{pri}(\alpha)$ or $\tau : k \in I_k(E')$ for some k. Therefore $\alpha : k \in I_k(E'\backslash L)$ with $k = \text{pri}(\alpha)$ or $\tau{:}k \in I_k(E'\backslash L)$ for some k.

Case $E \equiv E'[f]$**.** Suppose $\overline{E'[f]} \equiv \overline{E'}[f] \xrightarrow{\alpha}$. Then $\overline{E'} \xrightarrow{\beta}$ with $f(\beta) = \alpha$, and by induction $\beta\!:\!k \in I_k(E')$ with $k = \mathrm{pri}(\beta)$ or $\tau\!:\!k \in I_k(E')$ for some k. Therefore $\alpha : k \in I_k(E'[f])$ with $k = \mathrm{pri}(\alpha)$, since $f(\beta : k) = f(\beta) : k = \alpha : k$, or $\tau\!:\!k \in I_k(E'[f])$ for some k. $\square$

To show that an action of $\overline{E}$ can be matched by some action of E we use the following fact: if $\alpha\!:\!k$ is an initial action of E, then either E has an $\alpha\!:\!k$ transition, or E has a $\tau\!:\!j$ transition for some j with higher priority than k.

Lemma 8.3 *Let E be a uniformly prioritized PCCS process. Then*

$$\alpha\!:\!k \in I_k(E) \Rightarrow (E \xrightarrow{\alpha:k}_p \text{ or } E \xrightarrow{\tau:j}_p \text{ for some } j < k).$$

Proof. The proof is by induction on pairs of the form

$$(k, \text{depth of derivation of } I_k(E)).$$

We order the pairs lexicographically, and show that the claim holds for a pair if we assume the claim for all pairs smaller in the order than the one under consideration.

Case $E \equiv A$**.** Here we assume $A \stackrel{\text{def}}{=} F$. Suppose $\alpha\!:\!k \in I_k(A)$. Then $\alpha\!:\!k \in I_k(F)$ by a shorter inference, and by induction $F \xrightarrow{\alpha:k}_p$ or $F \xrightarrow{\tau:j}_p$ with $j < k$, so $A \xrightarrow{\alpha:k}_p$ or $A \xrightarrow{\tau:j}_p$ by def. of PCCS.

Case $E \equiv \alpha.E'$**.** Suppose $\alpha\!:\!k \in I_k(\alpha.E')$. Then $\alpha\!:\!k.E' \xrightarrow{\alpha:k}_p E'$ by def. of PCCS.

Case $E \equiv \sum_{i \in I} E_i$**.** Suppose $\alpha\!:\!k \in I_k(\sum_{i \in I} E_i)$. Then $\alpha\!:\!k \in I_k(E_i)$ for some $i \in I$ by a shorter inference. By induction we have two cases: $E_i \xrightarrow{\alpha:k}_p$ or $E_i \xrightarrow{\tau:j}_p$ with $j < k$. Suppose $E_i \xrightarrow{\alpha:k}_p$. There are two subcases:

i) $\tau \notin I_{<k}(E_{i'})$ for all i' in $I - \{i\}$. Then $\sum_{i \in I} E_i \xrightarrow{\alpha:k}_p$ by def. of PCCS.

ii) $\tau\!:\!j \in I_j(E_{i'})$ for some i' in $I - \{i\}$ and $j < k$. Then $\tau\!:\!j \in I_j(\sum_{i \in I} E_i)$, so by induction $\sum_{i \in I} E_i \xrightarrow{\tau:j'}_p$ for some j' such that $j' \leq j < k$.

The second case is for $E_i \xrightarrow{\tau:j}_p$ with $j < k$. This case is very similar to the first case.

Case $E \equiv E_1 \mid E_2$**.** Suppose $\alpha : k \in I_k(E_1 \mid E_2)$. There are three cases. For the first assume $\alpha = \tau$ and $l \in I_k(E_1)$ and $\bar{l} \in I_k(E_2)$. Then by induction $(E_1 \xrightarrow{l:k}_p$ or $E_1 \xrightarrow{\tau:j_1}_p)$ and $(E_2 \xrightarrow{\bar{l}:k}_p$ or $E_2 \xrightarrow{\tau:j_2}_p)$, with $j_1, j_2 < k$. There are three subcases:

i) $\tau \notin I_{<k}(E_1) \cup I_{<k}(E_2)$. Then $E_1 \xrightarrow{l:k}_p$ and $E_2 \xrightarrow{\bar{l}:k}_p$, so $E_1 \mid E_2 \xrightarrow{\tau:k}_p$.

ii) $\tau : j \in I_j(E_1)$ for some $j < k$. Then $\tau : j \in I_j(E_1 \mid E_2)$, so by induction $E_1 \mid E_2 \xrightarrow{\tau:j'}_p$ for some j' such that $j' \leq j < k$.

iii) Symmetrical to subcase ii.

The second and third main cases are for $\alpha : k \in I_k(E_1)$ and $\alpha : k \in I_k(E_2)$. These are very similar to the $E \equiv \sum_{i \in I} E_i$ case.

Case $E \equiv E' \backslash L$**.** Suppose $\alpha : k \in I_k(E' \backslash L)$, so $\alpha : k \in I_k(E')$ and $\alpha : k \notin L \cup \overline{L}$. By induction $E' \xrightarrow{\alpha:k}_p$ or $E' \xrightarrow{\tau:j}_p$ for some $j < k$. Therefore $E' \backslash L \xrightarrow{\alpha:k}_p$ or $E' \backslash L \xrightarrow{\tau:j}_p$ by def. of PCCS.

Case $E \equiv E'[f]$**.** Suppose $\alpha : k \in I_k(E'[f])$, so $\beta : k \in I_k(E')$ and $f(\beta : k) = \alpha : k$. By induction $E' \xrightarrow{\beta:k}_p$ or $E' \xrightarrow{\tau:j}_p$ for some $j < k$. Therefore $E'[f] \xrightarrow{\alpha:k}_p$ or $E'[f] \xrightarrow{\tau:j}_p$ by def. of PCCS. □

The two preceding lemmas lead directly to the desired result.

Lemma 8.4 *Let E be a uniformly prioritized PCCS process. Then*

$\overline{E} \rightarrow \;\Rightarrow\; E \rightarrow_p$.

Proof. Assume $\overline{E} \xrightarrow{\alpha}$ for some α. Then $\alpha : k \in I_k(E)$ for some k by Lemma 8.2, and so $E \xrightarrow{\alpha:k}_p$ or $E \xrightarrow{\tau:j}_p$ for some $j < k$ by Lemma 8.3. □

For our main result we must compare the behaviour of a PCCS process and a CCS process. To do so we define an operator PF ("priority-free") that elides the priority in transitions between PCCS processes.

$$\frac{E \xrightarrow{\alpha:k}_p E'}{PF(E) \xrightarrow{\alpha} PF(E')}$$

Now we can express that a CCS process ready-simulates any uniformly prioritized version of it.

Theorem 8.3 *Let E be a uniformly prioritized PCCS process. Then*

$$PF(E) \preceq \overline{E}.$$

where $\preceq$ here is the ready-simulates relation over states of the LTS $(\mathcal{P} \cup \{PF(F) \mid F \in \mathcal{P}_p\}, Act, \{\xrightarrow{\alpha} \mid \alpha \in Act\})$.

Proof. Let $\mathcal{R}$ be a relation consisting of all pairs of form $(PF(E), \overline{E})$, where E is a uniformly prioritized PCCS process. Suppose $(PF(E), \overline{E}) \in \mathcal{R}$. By Lemma 8.1 $PF(E) \xrightarrow{\alpha} PF(E')$ implies $\overline{E} \xrightarrow{\alpha} \overline{E'}$. By Lemma 8.4 $PF(E) \not\rightarrow$ implies $\overline{E} \not\rightarrow$. Therefore $\mathcal{R}$ is an anonymous ready simulation. $\square$

Low-priority actions of a PCCS process can sometimes be removed without affecting the behaviour of the process. We noted earlier that

$$a\!:\!1.0 + b\!:\!2.0 \not\sim_p a\!:\!1.0$$

because an action can be preempted by a lower-priority action α only if α is τ. However, it is the case that

$$(a\!:\!1.0 + b\!:\!2.0 \mid \overline{a}\!:\!1.0 + \overline{b}\!:\!2.0)\backslash\{a\!:\!1, b\!:\!2\} \sim_p (a\!:\!1.0 \mid \overline{a}\!:\!1.0)\backslash\{a\!:\!1, b\!:\!2\}.$$

Here the lower priority action $b\!:\!2$ can be removed because $\overline{a}\!:\!1$ can occur whenever $\overline{b}\!:\!2$ can occur. We generalize this idea in the following algebraic law.

Definition 8.4 *Let l be a label, L be a set of labels, and E be a PCCS process. Then l* enables some L *in E if*

$$(E' \xrightarrow{l:i}_p \text{ for some } i) \Rightarrow (E' \xrightarrow{l':j}_p \text{ for some } l' \text{ in } L \text{ and some } j)$$

and

$$(E' \xrightarrow{\bar{l}:i}_p \text{ for some } i) \Rightarrow (E' \xrightarrow{\overline{l'}:j}_p \text{ for some } l' \text{ in } L \text{ and some } j)$$

for all E' in $\mathrm{Rch}(E)$.

We say that l *enables all* L in E if l enables some $\{l'\}$ in E for all l' in L. Notice that if l enables some L in E, then l enables some L in every E' reachable from E.

In proving our algebraic law we use the following fact, which holds directly by definition of $I_{<i}()$.

Lemma 8.5 *Let E and F be PCCS processes, L be a label set, and i be a natural number. Then*

$$I_{<i}(E\backslash L \mid F\backslash L) \subseteq I_{<i}(E \mid F).$$

Proposition 8.2 *Let $\{l\}$, K, and L be label sets such that $\{l\} \cup K \subseteq L \cup \overline{L}$. Let $(E \mid F)\backslash L$ be a value-passing PCCS process over data domain D and priority mapping P such that $P(l') < P(l)$ for all l' in K, and l enables all K in E, and l enables some K in F. Then*

$$(E \mid F)\backslash L \sim_p (E\backslash\{l\} \mid F\backslash\{l\})\backslash L.$$

Proof. Let $\mathcal{R}$ be the relation consisting of pairs of either of the following forms:

$$((E' \mid F')\backslash L \quad , \quad (E'\backslash\{l\} \mid F'\backslash\{l\})\backslash L)$$
$$(G \quad , \quad G)$$

where $E' \in \mathrm{Rch}(E)$ and $F' \in \mathrm{Rch}(F)$. We show that $\mathcal{R}$ is a strong bisimulation. It is enough to consider transitions of the processes in the first kind of pair in $\mathcal{R}$. We begin by looking at transitions of the process on the left.

Suppose $(E' \mid F')\backslash L \xrightarrow{\alpha:i}_p (E'' \mid F')\backslash L$ by independent action of E'. Then by the semantics of PCCS $E' \xrightarrow{\alpha:i}_p E''$ and $\tau \notin I_{<i}(E' \mid F')$ and $\alpha \notin L \cup \overline{L}$. Therefore $E'\backslash\{l\} \xrightarrow{\alpha:i}_p E''\backslash\{l\}$ since $\alpha \notin \{l, \overline{l}\}$, and $(E'\backslash\{l\} \mid F'\backslash\{l\})\backslash L \xrightarrow{\alpha:i}_p (E''\backslash\{l\} \mid F'\backslash\{l\})\backslash L$ since $\tau \notin I_{<i}(E'\backslash\{l\} \mid F'\backslash\{l\})$ by Lemma 8.5 and $\alpha \notin L \cup \overline{L}$. The case for independent action by F' is symmetrical.

Next suppose $(E' \mid F')\backslash L \xrightarrow{\tau:i}_p (E'' \mid F'')\backslash L$ by synchronisation between E' and F'. Then by the semantics of PCCS there exists an l' such that $E' \xrightarrow{l':i}_p E''$ and $F' \xrightarrow{\overline{l'}:i}_p F''$ and $\tau \notin I_{<i}(E' \mid F')$. We now prove $l' \notin \{l, \overline{l}\}$ by contradiction. Suppose $l' \in \{l, \overline{l}\}$. By the enablement conditions on E' and F' we know $E' \xrightarrow{l'':k}_p E'''$ and $F' \xrightarrow{\overline{l''}:k}_p F'''$ for some l'' in K, E''', and F'''. Therefore $\tau \in I_{<i}(E' \mid F')$, a contradiction. So it must be that $l' \notin \{l, \overline{l}\}$. Therefore $E'\backslash\{l\} \xrightarrow{l':i}_p E''\backslash\{l\}$ and $F'\backslash\{l\} \xrightarrow{\overline{l'}:i}_p F''\backslash\{l\}$, and by Lemma 8.5 $\tau \notin I_{<i}(E'\backslash\{l\} \mid F'\backslash\{l\})$, so $(E'\backslash\{l\} \mid F'\backslash\{l\})\backslash L \xrightarrow{\tau:i}_p (E''\backslash\{l\} \mid F''\backslash\{l\})\backslash L$.

Now we look at transitions of the process on the right. Suppose $(E'\backslash\{l\} \mid F'\backslash\{l\})\backslash L \xrightarrow{\alpha:i}_p (E''\backslash\{l\} \mid F'\backslash\{l\})\backslash L$ by independent action of E'. Then by the semantics of PCCS $E' \xrightarrow{\alpha:i}_p E''$ and $\alpha \notin L \cup \overline{L} \cup \{l, \overline{l}\}$ and $\tau \notin I_{<i}(E'\backslash\{l\} \mid F'\backslash\{l\})$. We now prove $\tau \notin I_{<i}(E' \mid F')$ by contradiction. Suppose $\tau \in I_{<i}(E' \mid F')$. Then since $\tau \notin I_{<i}(E'\backslash\{l\} \mid F'\backslash\{l\})$ there exists an $l' \in \{l, \overline{l}\}$ and a $j < i$ such that $l' : j \in I_j(E')$ and $\overline{l'} : j \in I_j(F')$. Therefore $\tau \in I_{<i}(E' \mid F')$, contrary to assumption, so $\tau \notin I_{<i}(E' \mid F')$, and then $(E' \mid F')\backslash L \xrightarrow{\alpha:i}_p (E'' \mid F')\backslash L$ by the semantics of PCCS. The case for independent action by F' is symmetrical.

Next suppose $(E'\backslash\{l\} \mid F'\backslash\{l\})\backslash L \xrightarrow{\tau:i}_p (E''\backslash\{l\} \mid F''\backslash\{l\})\backslash L$ by synchronisation between $E'\backslash\{l\}$ and $F'\backslash\{l\}$. Then by the semantics of PCCS there exists an $l' \notin \{l, \overline{l}\}$ such that $E' \xrightarrow{l':i}_p E''$ and $F' \xrightarrow{\overline{l'}:i}_p F''$ and $\tau \notin I_{<i}(E'\backslash\{l\} \mid F'\backslash\{l\})$. By the same argument as in the previous case $\tau \notin I_{<i}(E' \mid F')$. Therefore $(E' \mid F')\backslash L \xrightarrow{\tau:i}_p (E'' \mid F'')\backslash L$. $\square$

The following algebraic laws for value-passing PCCS processes are also used later. The first laws holds because a priorised action can never occur in the presence of a lower-priority τ action. The other laws are the PCCS counterparts of CCS laws for strong equivalence. They are easy to prove by exhibiting bisimulation relations.

Proposition 8.3 (Value-passing PCCS laws)

$$\begin{aligned}
\tau.E + a(x).F &\sim_p \tau.E \qquad \textit{if } P(\tau) < P(a) && (8.1)\\
\tau.E + \overline{a}(e).F &\sim_p \tau.E \qquad \textit{if } P(\tau) < P(a) && (8.2)\\
(E \mid F)\backslash L &\sim_p E\backslash L \mid F\backslash L \qquad \textit{if } \mathcal{L}(E) \cap \overline{\mathcal{L}(F)} \cap (L \cup \overline{L}) = \emptyset && (8.3)\\
(E + F)\backslash L &\sim_p E\backslash L + F\backslash L && (8.4)\\
E\backslash(L_1 \cup L_2) &\sim_p E\backslash L_1\backslash L_2 && (8.5)\\
E\backslash L &\sim_p E \qquad \textit{if } \mathcal{L}(E) \cap (L \cup \overline{L}) = \emptyset && (8.6)\\
a(x).E\backslash L &\sim_p 0 \qquad \textit{if } a \in L \cup \overline{L} && (8.7)\\
\overline{a}(e).E\backslash L &\sim_p 0 \qquad \textit{if } a \in L \cup \overline{L} && (8.8)
\end{aligned}$$

8.5 Ben-Ari's Garbage Collection Algorithm

Traditional imperative programming languages force programmers to manage the memory use of their programs. In particular, a programmer must allocate memory when it is needed and free memory when it is no longer needed. Some programming languages relieve programmers of this chore. The usual approach is for the run-time system of such a language to occasionally interrupt program execution and run a *garbage-collection* algorithm, which discovers inaccessible memory elements and frees them for later use. Program execution is then resumed. With a *concurrent garbage collection algorithm* (see Fig. 8–1) the program and garbage-collection algorithm are instead run concurrently, permitting the memory state to change while garbage collection is underway. One of the first concurrent garbage collection algorithms was presented in [33]. The algorithm uses a three-colour marking scheme and has a complex correctness proof. Ben-Ari devised an algorithm with a two-colour marking scheme and a simpler correctness proof [5]. However, Ben-Ari's proofs and some of his claims were shown to be erroneous [81,76]. Here we show how one of these erroneous claims can be exposed through model checking once the algorithm is abstracted.

Ben-Ari's algorithm describes the actions of two concurrent processes: a *mutator*, which represents program operations on memory, and a *collector*, which collects garbage. Program memory is organised as an array of elements called *nodes*. Each node has a fixed number of pointers to other nodes. The nodes pointed to from a node are called its *children*. A fixed set of nodes are identified as the *roots*. The *accessible* nodes in a state of memory are the roots and all nodes reachable by the roots via pointers. Among the roots is the node that points to the *free list*, which is a linked list of unallocated nodes. Each node also has a *marking*, which is either black or white. Initially, all nodes are linked on the free list and all nodes are marked white.

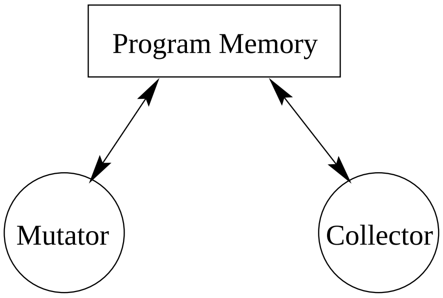


Figure 8–1: Concurrent garbage collection

The *mutator* process represents program operations on memory. Memory reads and writes are not modelled because garbage collection ignores the data stored in nodes. Memory pointer operators are modelled (following [33]) as a single abstract operation that redirects a pointer from one accessible node to another accessible node. The pointer-redirection operation cannot make an inaccessible node accessible, but can make an accessible node inaccessible by directing a pointer away from it. The mutator process also contains an operation to help the garbage collector. After a pointer is redirected, its new target is blackened.

Mutator

1. Redirect a pointer from one accessible node to another accessible node.

2. Blacken the new target node.

The *collector* process models the operations of the garbage collector. It marks the root nodes black, and then propagates the marking through the memory pointers until no further marking is possible. All white nodes are then collected by inserting them into the free list.

Collector

1. Blacken the root nodes.

2. Examine each node in memory. If the node is black, then mark all children of the node as black.

3. Count the number of black nodes. If the count has increased since the last propagation cycle, return to step 1.

4. Examine each node in memory. If the node is white, insert it into the free list. If the node is black, mark it as white.

Since the free list is pointed to by a root, inserting a node into the free list makes the node accessible. Thus collecting an inaccessible node makes it accessible again.

The algorithm is difficult to understand because of the possible interleavings of the mutator and collector. In particular, the mutator can redirect pointers while the collector marks memory nodes. Notice that memory is viewed in different ways by the mutator and collector. The mutator sees memory as a directed graph of nodes and pointers, while the collector sees memory as an array.

The second mutator step is subtle and may not seem necessary. We show that it is necessary with an example that was first presented in [15]. The leftmost diagram of Figure 8–2 is a snapshot of part of memory during garbage collection. Nodes A and B have already been blackened, and the collector is about to begin another round of propagating the markings. The collector examines node A, which has no white children. The mutator then changes A's pointer so that it points to node C, as shown in the middle picture. This mutator operation is possible because A and C are both accessible. The mutator then changes B's pointer so that it no longer points to C, as shown in the right-most picture. The collector examines node B, which also has no white children, and node C, which is white. Since this round of propagation by the collector has not increased the number of black nodes, propagation ends. Node C is then collected, although it is accessible.

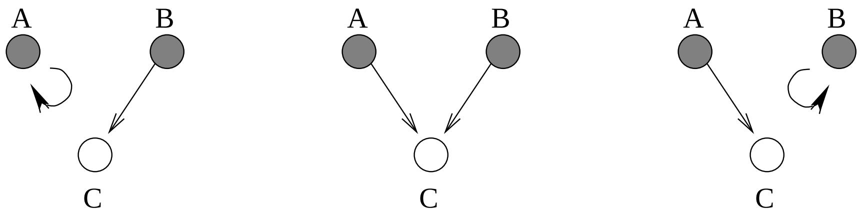


Figure 8–2: A problem that arises if mutator step 2 is omitted

Ben-Ari claimed that his algorithm is *safe* and *live*, where here safety means that only garbage is collected, and liveness means that eventually all garbage is collected. Furthermore, Ben-Ari claimed that safety and liveness of his algorithm would hold even if the two mutator steps were reversed. In a previous study of Ben-Ari's algorithm [15] we were able to show that safety does not hold if the second mutator step is eliminated. However, we could not show that safety does not hold if the mutator steps are reversed, because this problem was not exposed in the largest model we were able to check with the Concurrency Workbench. When we attempted to extend our model by allowing each node to have two sons instead of one, the state space of the model grew so much that model checking with the Concurrency Workbench was impossible.

8.6 Modelling Ben-Ari's Algorithm in CCS

In this section we present a value-passing CCS model of Ben-Ari's algorithm with the mutator operations reversed. Our model consists of three processes: one for memory, one for the mutator, and one for the collector. We describe the memory process *Mem* first. There are n memory nodes, each of which has k sons. The root nodes are assumed to be the first r nodes of memory, with node fl the root node that points to the head of the free list. We write N as an abbreviation for the set $\{1, \ldots, n\}$ of node indices, R as an abbreviation for the set $\{1, \ldots, r\}$ of root indices, and S as an abbreviation for the set $\{1, \ldots, k\}$ of son indices. Here we fix k to be 2 for simplicity.

Process *Mem* has parameters *Colors* and *Sons*, which model the marking and pointers of memory nodes. *Colors* is an n-tuple of values from $\{black, white\}$. The ith element of *Colors* represents the colour of the ith node. *Sons* is a pair of n-tuple of values from $\{1, \ldots, n\}$. The ith element of the first (second) n-tuple represents the first (second) son of the ith node. If t is a tuple, we write $t[i \rightarrow x]$ for t with its ith element replaced by x, and t_i for the ith element of t.

The accessible nodes in memory state $(Colors, Sons)$ are the nodes of the smallest set including R that is closed under pointer accessibility. We write *Acc* in the definition of *Mem* to stand for the accessible nodes in the current memory state.

Memory operations are modelled by the actions of process *Mem*: $\mathtt{setcolor}_i(c)$ (set the colour of node i to c), $\overline{\mathtt{color}}_i(c)$ (output the colour of node i), $\overline{\mathtt{son}}_i(j)$ (output the sons of node i), $\mathtt{mutate}_{i,s,j}$ (make son s of accessible node i point to accessible node j), $\mathtt{bdest}_{i,j}$ (blacken node j if i and j are accessible), $\mathtt{free}(i)$ (node i is garbage and can be freed), `fail` (an accessible node was flagged as garbage), and `coll` (put a node on the free list).

$$
\begin{aligned}
&Mem(Colors, Sons) \stackrel{\text{def}}{=} \\
&\quad \sum_{i\in N} \texttt{setcolor}_i(c).Mem(Colors[i \to c], Sons) + \\
&\quad \sum_{i\in N} \overline{\texttt{color}}_i(Colors_i).Mem(Colors, Sons) + \\
&\quad \sum_{i\in N} \overline{\texttt{son}}_i(((Sons_1)_i, (Sons_2)_i)).Mem(Colors, Sons) + \\
&\quad \sum_{(i,s,j)\in Acc\times S\times Acc} \texttt{mutate}_{i,s,j}\,.Mem(Colors, Sons[s \to Sons_s[i \to j]]) + \\
&\quad \sum_{(i,j)\in Acc\times Acc} \texttt{bdest}_{i,j}\,.Mem(Colors[j \to black], Sons) + \\
&\quad \texttt{free}(i).\text{if } i \in Acc \text{ then } \texttt{fail}\,.0 \\
&\quad\qquad\qquad\qquad\quad \text{else } (\tau.Mem(Colors, Sons) + \\
&\quad\qquad\qquad\qquad\qquad \texttt{coll}\,.Mem(Colors, \text{insert}(i, Sons))
\end{aligned}
$$

The details of inserting a node into the free list are not specified by Ben-Ari in [5]. In our model, action `free` allows memory to collect the specified node i by putting it at the head of the free list. The manipulation of the free list is done by function insert. Notice that we include in our model the possibility that node i is not collected after $\texttt{free}(i)$ occurs, even if the node is inaccessible. This models that garbage collection has been deferred for this node.

Process *Mutator* models the mutator with the mutator steps reversed. First node j is blackened, and then son s of node i is redirected to node j. The treatment of actions `bdest` and `mutate` in process *Mem* ensures that *Mutator* can only mutate accessible nodes.

$$
Mutator \stackrel{\text{def}}{=} \sum_{(i,j)\in N\times N} \overline{\texttt{bdest}}_{i,j}\,. \sum_{s\in S} \overline{\texttt{mutate}}_{i,s,j}\,.Mutator
$$

Process *Collector* models the collector. Process C_i models step i in the informal description of the collector in the previous section.

$$
\begin{array}{rcl}
\mathit{Collector} & \stackrel{\text{def}}{=} & C_1(1) \\[2ex]
C_1(i) & \stackrel{\text{def}}{=} & \text{if } i > r \text{ then } C_2(0,1) \\
 & & \qquad\quad \text{else } \overline{\texttt{setcolor}_i}(\mathit{black}).C_1(i+1) \\[2ex]
C_2(\mathit{old}, i) & \stackrel{\text{def}}{=} & \text{if } i > n \text{ then } C_3(\mathit{old}, 0, 1) \\
 & & \qquad\quad \text{else } \texttt{color}_i(c).\text{if } c = \mathit{black} \\
 & & \qquad\qquad\qquad\qquad \text{then } \texttt{son}_i((j,k)). \\
 & & \qquad\qquad\qquad\qquad\qquad \overline{\texttt{setcolor}_j}(\mathit{black}). \\
 & & \qquad\qquad\qquad\qquad\qquad \overline{\texttt{setcolor}_k}(\mathit{black}).C_2(\mathit{old}, i+1) \\
 & & \qquad\qquad\qquad\qquad \text{else } C_2(\mathit{old}, i+1) \\
C_3(\mathit{old}, \mathit{new}, i) & \stackrel{\text{def}}{=} & \text{if } i > n \text{ then } (\text{if } \mathit{old} = \mathit{new} \text{ then } C_4(1) \\
 & & \qquad\qquad\qquad\qquad\quad \text{else } C_2(\mathit{new}, 1)) \\
 & & \qquad\quad \text{else } \texttt{color}_i(c).\text{if } c = \mathit{black} \\
 & & \qquad\qquad\qquad\qquad \text{then } C_3(\mathit{old}, \mathit{new}+1, i+1) \\
 & & \qquad\qquad\qquad\qquad \text{else } C_3(\mathit{old}, \mathit{new}, i+1) \\
C_4(i) & \stackrel{\text{def}}{=} & \text{if } i > n \text{ then } C_1(1) \\
 & & \qquad\quad \text{else } \texttt{color}_i(c).\text{if } c = \mathit{black} \\
 & & \qquad\qquad\qquad\qquad \text{then } \overline{\texttt{setcolor}_i}(\mathit{white}).C_4(i+1) \\
 & & \qquad\qquad\qquad\qquad \text{else } \overline{\texttt{free}}(i).C_4(i+1)
\end{array}
$$

In each step the collector iterates over the nodes starting at the first node. At the beginning of each step the index i, representing the current point in the iteration, is checked to see if the next step should begin. The parameters *old* and *new* represent the old and new count of the black nodes.

Process GC models the complete algorithm. We write L for the set $\bigcup_{(i,s,j)\in N\times S\times N}$ $\{\texttt{setcolor}_i, \texttt{color}_i, \texttt{son}_i, \texttt{mutate}_{i,s,j}, \texttt{bdest}_{i,j}, \texttt{free}\}$ of names.

$$GC \stackrel{\text{def}}{=} (Mem(Colors_0, Sons_0) \mid Mutator \mid Collector)\backslash L$$

$Colors_0$ and $Sons_0$ are the initial values of the memory parameters. Figure 8–3 is a flow diagram of process GC showing the concurrent components and their connections.

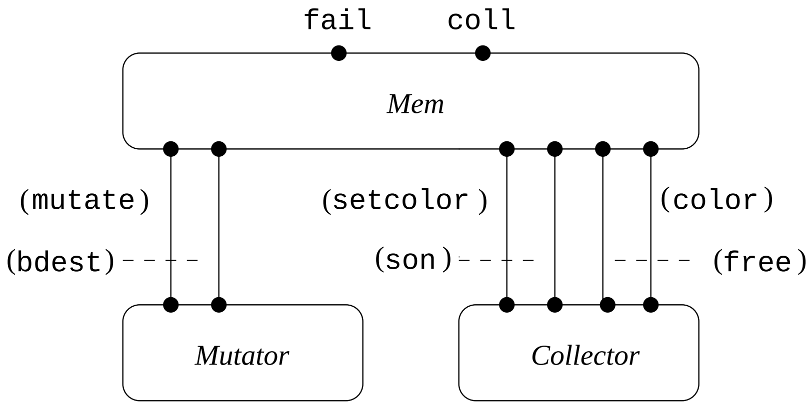


Figure 8–3: Flow diagram of process GC

We want to show that the collector never attempts to collect an accessible node. Our CCS model should therefore satisfy the formula expressing that action `fail` can never occur.

$$Safe \stackrel{\text{def}}{=} \nu X.[\texttt{fail}]\,\texttt{ff} \wedge [-]X$$

In the instance of the model we examine there are $n = 4$ memory nodes, $r = 1$ roots, and $k = 2$ sons for each node. The initial values are $Colors_0 = (white, white, white, white)$ and $Sons_0 = ((4, 2, 2, 3), (4, 2, 3, 3))$. It is hard to estimate the number of states in this instance of the model. We verify value-passing CCS processes by first translating them to basic CCS, and here our translator was not even able to produce the basic CCS version of the model. Process *Mem* has about 10^6 states, process *Collector* has about 500 states, and process *Mutator* has about 15 states, so process GC may have as many as 10^9 states.

8.7 Abstracting Ben-Ari's Algorithm

We now abstract Ben-Ari's algorithm by first prioritizing the actions of process GC and then performing some algebraic simplification.

8.7.1 Adding Priorities

The main reason that our model of Ben-Ari's algorithm has a large state space is that there are many possible memory states. In particular, there are many possible ways in which the memory nodes can be connected by pointers. Another reason is that the actions of components *Mem*, *Mutator*, and *Collector* can be interleaved in many possible ways. We now prioritize the actions of GC to reduce its state space.

The priority mapping P that we use is defined as follows. Let $B = \{(1,2,1), (1,2,2), (1,2,4), (2,1,2), (3,1,2), (3,1,3), (4,1,3)\}$ and let $B' = (N \times S \times N) - B$. Let $C = \{(i,j) \mid (i,s,j) \in B\}$ and let $C' = (N \times N) - C$.

$$P(\alpha) = \begin{cases} 0 & \text{if } \alpha \in \bigcup_{i \in N} \{\texttt{fail}, \texttt{setcolor}_i, \texttt{son}_i, \texttt{free}_i\} \\ 1 & \text{if } \alpha \in \bigcup_{i \in N} \{\texttt{color}_i\} \cup \bigcup_{(i,s,j) \in B} \{\texttt{bdest}_{i,j}, \texttt{mutate}_{i,s,j}\} \\ 3 & \text{if } \alpha = \texttt{coll} \\ 2 & \text{otherwise} \end{cases}$$

Actions `fail`, `setcolor`, `son`, and `free` are given high priority to reduce interleavings. For example, the first step of the collector, in which the roots are blackened, cannot be interleaved with mutator operations in our prioritized version of the model.

Some `bdest` and `mutate` actions are given lower priority to eliminate some of the possible memory configurations. Figure 8–4 shows the links between nodes that are possible through higher-priority `mutate` actions. The edges on the top half of the graph represent pointers to first sons; the edges on the bottom half represent pointers to second sons. To decide which indexed versions of `bdest` and `mutate` should be given low priority, we considered some features our model should have for the safety property to be violated. First, it should allow all nodes to be reached at all times, as only reachable nodes can be collected. Secondly, it should not have fixed connections making all nodes reachable, as then the collector would always blacken all nodes. Finally, it should have memory pointers that run from higher-numbered nodes to lower-numbered ones, as this allows the most cycles of blackening in the collector.

Action `coll` is given low priority to prevent process *Mem* from putting garbage on the free list. The pointer manipulation involved in putting garbage on the free list can introduce memory configurations that we attempted to make unreachable by our assignment of priorities to `bdest` and `mutate` actions.

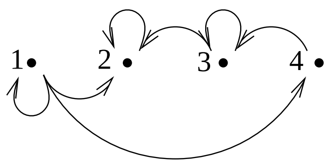


Figure 8–4: Pointers of High-Priority Mutate Actions

8.7.2 Algebraic Simplification

In Section 8.6 we stated that it is impracticable to translate our value-passing model of Ben-Ari's algorithm to basic CCS because of the many possible parameter values of process *Mem*. Adding priorities to the model does not solve this problem. However, through algebraic simplification we can greatly reduce the number of parameter values and hence the number of process definitions that are created. To do this we first rewrite *Mutate* and *Mem* slightly to separate the `mutate` and `bdest` actions having different priority levels.

$$Mem(Colors, Sons) \stackrel{\text{def}}{=}$$

$$\sum_{i \in N} \mathtt{setcolor}_i(c).Mem(Colors[i \to c], Sons) +$$

$$
\begin{aligned}
&\sum_{i \in N} \overline{\texttt{color}}_i(\mathit{Colors}_i).\mathit{Mem}(\mathit{Colors}, \mathit{Sons}) + \\
&\sum_{i \in N} \overline{\texttt{son}}_i(((\mathit{Sons}_1)_i, (\mathit{Sons}_2)_i)).\mathit{Mem}(\mathit{Colors}, \mathit{Sons}) + \\
&\sum_{(i,s,j) \in B} \text{if } \{i,j\} \subseteq \mathit{Acc} \text{ then } \texttt{mutate}_{i,s,j}\,.\mathit{Mem}(\mathit{Colors}, \mathit{Sons}[s \rightarrow \mathit{Sons}_s[i \rightarrow j]]) + \\
&\sum_{(i,s,j) \in B'} \text{if } \{i,j\} \subseteq \mathit{Acc} \text{ then } \texttt{mutate}_{i,s,j}\,.\mathit{Mem}(\mathit{Colors}, \mathit{Sons}[s \rightarrow \mathit{Sons}_s[i \rightarrow j]]) + \\
&\sum_{(i,j) \in C} \text{if } \{i,j\} \subseteq \mathit{Acc} \text{ then } \texttt{bdest}_{i,j}\,.\mathit{Mem}(\mathit{Colors}[j \rightarrow \mathit{black}], \mathit{Sons}) + \\
&\sum_{(i,j) \in C'} \text{if } \{i,j\} \subseteq \mathit{Acc} \text{ then } \texttt{bdest}_{i,j}\,.\mathit{Mem}(\mathit{Colors}[j \rightarrow \mathit{black}], \mathit{Sons}) + \\
&\texttt{free}(i).\text{if } i \in \mathit{Acc} \text{ then } \texttt{fail}\,.0 \\
&\qquad\qquad\qquad\qquad \text{else } (\tau.\mathit{Mem}(\mathit{Colors}, \mathit{Sons}) + \\
&\qquad\qquad\qquad\qquad\qquad \texttt{coll}\,.\mathit{Mem}(\mathit{Colors}, \text{insert}(i, \mathit{Sons}))
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Mutator} \stackrel{\text{def}}{=} & \sum_{(i,j) \in C} \overline{\texttt{bdest}}_{i,j}\,.\sum_{s \in S} \overline{\texttt{mutate}}_{i,s,j}\,.\mathit{Mutator} + \\
& \sum_{(i,j) \in C'} \overline{\texttt{bdest}}_{i,j}\,.\sum_{s \in S} \overline{\texttt{mutate}}_{i,s,j}\,.\mathit{Mutator}
\end{aligned}
$$

Next we apply Prop. 8.2 to restrict low-priority `bdest` actions. We instantiate Prop. 8.2 by letting E be process $\mathit{Mutator} \mid \mathit{Collector}$, F be process Mem, label l be $\texttt{bdest}_{i,j}$ for an arbitrary (i,j) in C', label set K be $\{\texttt{bdest}_{i',j'} \mid (i',j') \in C\}$, and label set L be just the label set L we have already defined for the top level restriction in process GC. The only non-trivial conditions of the proposition are the enablement conditions of E and F, given by the following proposition.

Proposition 8.4 *Suppose $(i,j) \in C'$. Then*

1. $\{\texttt{bdest}_{i,j}\}$ *enables all* $\{\texttt{bdest}_{i',j'} \mid (i',j') \in C\}$ *in* $\mathit{Mutator} \mid \mathit{Collector}$*, and*

2. $\{\texttt{bdest}_{i,j}\}$ *enables some* $\{\texttt{bdest}_{i',j'} \mid (i',j') \in C\}$ *in* $\mathit{Mem}(\mathit{Colors}_0, \mathit{Sons}_0)$*.*

Proof. The first condition holds by definition of *Mutator*. For the second, suppose $\mathit{Mem}(C,S) \in \text{Rch}(\mathit{Mem}(\mathit{Colors}_0, \mathit{Sons}_0))$ and $\mathit{Mem}(C,S) \xrightarrow{\texttt{best}_{i,j}:2}_p E$ for some E. We must show that $Mem(C,S) \xrightarrow{\texttt{best}_{i',j'}:1}_p E'$ for some E' and some (i',j') in C. By the def. of Mem it suffices to find an (i',j') in C such that $(i',j') \in Acc \times Acc$.

By the assumption $Mem(C,S) \xrightarrow{\texttt{best}_{i,j}:2}_p E$ and the def. of Mem we know that $(i,j) \in Acc \times Acc$. Then $\{(i,i),(i,j),(j,i),(j,j)\} \subseteq Acc \times Acc$ by simple set theory. By this observation we can find an (i',j') in $C \cap (Acc \times Acc)$ for every (i,j) in C' except element $(4,4)$. So suppose $(i,j) = (4,4)$. We know $4 \in Acc$ because $(i,j) \in Acc \times Acc$ and $1 \in Acc$ because 1 is a root node. Therefore $(1,4)$ of C is in $Acc \times Acc$. $\square$

Thus if $(i,j) \in C'$ then action $\texttt{bdest}_{i,j}$ can be restricted in processes Mem and $Mutator \mid Collector$. We can similarly restrict low-priority `mutate` actions. Let L_2 be set $\{\texttt{bdest}_{i,j} \mid (i,j) \in C'\} \cup \{\texttt{mutate}_{i,s,j} \mid (i,s,j) \in B'\}$ of low-priority `bdest` and `mutate` actions, and let L_1 be $L - L_2$. Then by Prop. 8.2 and PCCS laws 8.3, 8.5, and 8.6 we have

$$\begin{array}{lll}
 & (Mem(Colors_0, Sons_0) \mid Mutator \mid Collector)\backslash L & \\
\sim_p & (Mem(Colors_0, Sons_0) \mid Mutator \mid Collector)\backslash L_2 \backslash L_1 & \text{(by 8.5)} \\
\sim_p & (Mem(Colors_0, Sons_0)\backslash L_2 \mid (Mutator \mid Collector)\backslash L_2)\backslash L_1 & \text{(by 8.3)} \\
\sim_p & (Mem(Colors_0, Sons_0)\backslash L_2 \mid Mutator\backslash L_2 \mid Collector\backslash L_2)\backslash L_1 & \text{(by 8.3)} \\
\sim_p & (Mem(Colors_0, Sons_0)\backslash L_2 \mid Mutator\backslash L_2 \mid Collector)\backslash L_1 & \text{(by 8.6)}
\end{array}$$

Processes $Mem(Colors_0, Sons_0)\backslash L_2$ and $Mutator\backslash L_2$ can be rewritten as follows using PCCS laws 8.4, 8.6, 8.7, and 8.8.

$$\begin{array}{l}
Mem(Colors, Sons) \stackrel{\text{def}}{=} \\
\quad \sum_{i \in N} \texttt{setcolor}_i(c).Mem(Colors[i \to c], Sons) + \\
\quad \sum_{i \in N} \overline{\texttt{color}}_i(Colors_i).Mem(Colors, Sons) + \\
\quad \sum_{i \in N} \overline{\texttt{son}}_i(((Sons_1)_i, (Sons_2)_i)).Mem(Colors, Sons) + \\
\quad \sum_{(i,s,j) \in B} \text{if } \{i,j\} \subseteq Acc \text{ then } \texttt{mutate}_{i,s,j}.Mem(Colors, Sons[s \to Sons_s[i \to j]]) + \\
\quad \sum_{(i,j) \in C} \text{if } \{i,j\} \subseteq Acc \text{ then } \texttt{bdest}_{i,j}.Mem(Colors[j \to black], Sons) +
\end{array}$$

$$\begin{aligned}&\texttt{free}(i).\text{if } i \in Acc \text{ then } \texttt{fail}.0\\ &\qquad\qquad \text{else } (\tau.Mem(Colors, Sons) +\\ &\qquad\qquad\quad \texttt{coll}.Mem(Colors, \text{insert}(i, Sons))\end{aligned}$$

$$Mutator \stackrel{\text{def}}{=} \sum_{(i,j)\in C} \overline{\texttt{bdest}}_{i,j}\,.\sum_{s\in S} \overline{\texttt{mutate}}_{i,s,j}\,.Mutator$$

Next we apply PCCS law 8.1 to remove the part of *Mem* prefixed by `coll`.

$$\begin{aligned}&Mem(Colors, Sons) \stackrel{\text{def}}{=}\\ &\quad \sum_{i\in N} \texttt{setcolor}_i(c).Mem(Colors[i \to c], Sons) +\\ &\quad \sum_{i\in N} \overline{\texttt{color}}_i(Colors_i).Mem(Colors, Sons) +\\ &\quad \sum_{i\in N} \overline{\texttt{son}}_i(((Sons_1)_i, (Sons_2)_i)).Mem(Colors, Sons) +\\ &\quad \sum_{(i,s,j)\in B} \text{if } \{i,j\} \subseteq Acc \text{ then } \texttt{mutate}_{i,s,j}\,.Mem(Colors, Sons[s \to Sons_s[i \to j]]) +\\ &\quad \sum_{(i,j)\in C} \text{if } \{i,j\} \subseteq Acc \text{ then } \texttt{bdest}_{i,j}\,.Mem(Colors[j \to black], Sons) +\\ &\quad \texttt{free}(i).\text{if } i \in Acc \text{ then } \texttt{fail}.0\\ &\qquad\qquad \text{else } \tau.Mem(Colors, Sons)\end{aligned}$$

Our final step is to reduce the set of values over which parameter *Sons* can range. Define the set C_i of nodes that can be pointed to by node i as $\{j \mid (i,j) \in C\}$. For example, Figure 8–4 shows that $C_1 = \{1, 2, 4\}$. Abstract process $Mem(Colors_0, Sons_0)$ can reach only processes of form $Mem(C, (S_1, S_2))$, where S_1 and S_2 range over $C_1 \times C_2 \times C_3 \times C_4$. Therefore we redefine *Mem* so that $Sons_1$ and $Sons_2$ of *Mem* range over only $C_1 \times C_2 \times C_3 \times C_4$ in GC.

8.7.3 Checking the Abstract Process

We analysed our abstract model of Ben-Ari's algorithm with the Concurrency Workbench of North Carolina [25] because it supports the analysis of value-passing CCS processes. Our abstract model has about 6500 states, and was shown by the Concurrency Workbench not to satisfy formula *Safety*. Since *Safety* is a μLr formula, we know by Propositions 8.2 and 8.3, and Theorems 8.2 and 8.3, that the concrete version of GC also fails to satisfy *Safety*.

In prioritizing the actions in our model of Ben-Ari's algorithm we were guided by our assumptions about the kinds of memory configurations that would be needed to expose a violation of the algorithm's safety property. The results from the Concurrency Workbench show not only that the concrete algorithm violates the property, but also that our assumptions were correct. Had our assumptions been too strong, our abstract model would have satisfied the safety property even though the concrete version does not.

We could have shown that Ben-Ari's algorithm fails to satisfy property *Safe* using the simulates preorder instead of the ready-simulates preorder. Lemma 8.1 shows that a CCS process simulates every prioritized version of it. Then, since the simulates relation preserves formulas of $\mu L\Box$ (Theorem 3.5), and *Safe* is a formula of $\mu L\Box$, one can prove that our model of Ben-Ari's algorithm fails to satisfy *Safe* by proving that the prioritized model fails to satisfy it. We proved results about the ready-simulates preorder for two reasons. First, Theorem 8.3 is interesting on its own. It shows that deadlocks cannot be introduced in a process by adding priorities in a uniform way. Secondly, the results of this chapter can be used on examples in which all diamond operators in the μL formula to be verified are of the form $\langle - \rangle\, \mathtt{tt}$. Such formulas are commonly used to express properties of systems. Two examples are $\nu X.\langle - \rangle\, \mathtt{tt} \wedge [-]X$, which expresses deadlock freedom, and $\mu X.[-a]X \wedge \langle - \rangle\, \mathtt{tt}$, which expresses that action a will eventually occur.

8.8 Dekker's Algorithm

We now present a second application of abstraction with priorities that could not be performed using only the simulates preorder. We show that our model of Dekker's algorithm, presented in Section 7.1, fails to satisfy a basic liveness property. In Chapter 7 we showed that our model satisfies a liveness property under a fairness assumption. The liveness property was that if a process requests entry to its critical section, then it cannot wait forever while the other process continues to enter its own critical section. This property can be expressed without the use of the diamond operator.

Here we examine a stronger liveness property: a process that requests entry to its critical section will eventually enter its critical section. The property can be expressed in μLr as follows.

$$\mathit{Live} \stackrel{\text{def}}{=} \nu X.[\mathtt{req1}](\mu Y.\langle - \rangle\, \mathtt{tt} \wedge [-\,\mathtt{enter1}]Y) \wedge [-]X$$

Paraphrasing the formula gives: whenever `req1` action occurs, action `enter1` must eventually occur afterwards. This formula incorporates no fairness assumptions.

To prove that property *Live* does not hold of process *Dekker* we show that it does not hold of a uniformly prioritized version of *Dekker*. Each action of *Dekker* is assigned priority 0, 1, or 2 (recall that priority value 0 represents high priority). We attach priority 0 to action τ and to actions that represent writes to shared variables b_1 and b_2. This reduces the interleaving of actions in *Dekker*. We attach priority 2 to actions `enter1` and `enter2`. This prevents the processes from entering their critical sections unless no other action is possible. We attach intermediate priority 1 to the remaining actions.

While process *Dekker* has 197 states and 400 transitions, the prioritized version of *Dekker* has only 23 states and 36 transitions. Using the Concurrency Workbench of North Carolina we find that the prioritized version of *Dekker* does not satisfy formula *Live*. By Theorems 8.2 and 8.3 we therefore know that the *Dekker* itself also fails to satisfy formula *Live*. The property fails to hold because process 2 can indefinitely read variable k repeatedly and skip while process 1 is ready to enter its critical section.

8.9 Discussion

In this chapter we showed how priorities can be used to abstract CCS processes. We showed that a CCS process ready simulates any uniformly prioritized version of it, and showed that fragment μLr of the modal mu-calculus logically characterizes the ready-simulates preorder. We also described some algebraic laws for CCS with priorities. Finally, we looked at two examples. By using priorities we showed that a version of Ben-Ari's garbage collection algorithm fails to satisfy an important safety property. Abstraction reduced the state space of the algorithm from possibly millions of states to only about 6500 states. We also used priorities to show that Dekker's mutual exclusion algorithm fails to satisfy a liveness property.

Unlike the abstraction technique of Chapter 6 based on the weak simulation preorder, an abstract process here can intuitively do less than its corresponding concrete process. A possible criticism of the technique in this chapter is that to add priorities one must know where problems are likely to be found. In defence of the technique we can say that we added priorities to Ben-Ari's algorithm using only basic insights into the algorithm. The known counterexample [76] to the version of the algorithm we studied cannot be produced by our abstract model, because the counterexample involves memory configurations not reachable in our model. This shows that we did not add priorities just to allow the known counterexample to be produced.

A version of PCCS is used in [27] to model a railway signalling system. Temporal properties of the model are then checked using the Concurrency Workbench of North Carolina. The primary motivation given in the paper for the use of priorities is to make modelling more intuitive. It is also shown that the prioritized version of the model is smaller than the original version. However, it is not shown that properties of the prioritized model are also properties of the original model.

Chapter 9

Conclusions

Abstraction is not a specific technical topic. It is rather a broad style of verification in which one begins by removing irrelevant details from a model. The techniques developed in this thesis differ from most other work on process abstraction in that they are applied manually and work on process syntax. Thus these techniques can be viewed as an attempt to provide a formal justification of the simplifications that are commonly done to models prior to their verification.

In this thesis we have considered two main abstraction techniques. In one an abstract process is obtained by applying process operators to the concrete process; in the other the abstract and concrete processes are related by a process preorder. We have attempted to show the value of the techniques in two ways. First, we have shown that the techniques are general by using them to derive other verification techniques. Secondly, we have shown that the techniques have a reasonable scope of application by using them to help verify a selection of examples.

9.1 Summary of Main Results

For the first abstraction technique we showed how a class of process operators can serve as property transformers. In verifying with this technique one applies process operators to a process, proves that the abstract process satisfies a property, and proves that the transformed property is stronger than the property of interest. We defined an extension to the modal mu-calculus and showed how every static unary process operator transforms a mu-calculus formula to a formula of this new logic. We also showed that the transformed formula is a modal mu-calculus formula if the abstraction operator is an action mapping, a trace mapping, or a priority operator based on an ordering on actions.

To illustrate the generality of the first technique, we showed that the Interface Rule [22] is an application of it, as are Wolper's results on data-independent processes [97]. We also used the technique to show that one can hide all actions of a process not mentioned in a formula to be verified if the formula is written in the weak, diamond-free modal mu-calculus.

For the second abstraction technique we showed how the weakly simulates and ready simulates preorders can be used to relate abstract processes to concrete ones. In verifying with the weakly simulates preorder one abstracts a process by applying algebraic laws and then proves that the abstract process satisfies the property of interest. We showed that the weak, diamond-free modal mu-calculus characterizes the weak simulation preorder. This has also been shown in [6], but by use of a modal mu-calculus with infinitary disjunction and conjunction instead of fixed-point operators. Our proof used the weak, diamond-free modal mu-calculus directly. We also showed that the weak simulation preorder is a precongruence. We then defined abstraction operations as algebraic laws for the weak simulation preorder. We used the technique along with the first technique to verify properties of Dekker's algorithm. We abstracted the algorithm relative to a liveness property and a safety property, yielding two different abstract algorithms.

In verifying with the ready simulates preorder one abstracts a process by adding priorities to it and then proves that the abstract process fails to satisfy the property of interest. We showed that a process ready simulates any uniformly-prioritized version of it, and that the positive modal mu-calculus with diamond subformulas of form $\langle - \rangle\, \mathtt{tt}$ characterizes the ready simulates preorder. We also developed an algebraic law for value-passing processes with priorities that allows processes with low-priority prefixes to be removed. We used the technique to show that a version of Ben-Ari's concurrent garbage-collection algorithm fails to satisfy a safety property that Ben-Ari claimed it did satisfy. We also used the technique to show that Dekker's algorithm does not satisfy the liveness property that a waiting process will eventually enter its critical section.

9.2 Topics for Further Work

We chose CCS as a process notation because of its simplicity. Since our abstraction techniques are applied manually we would like to support abstraction of processes written in notations used by designers of distributed systems. The concurrent while language defined in Chapter 8 of [72] might be a good candidate. This language contains shared variables. Our work in abstracting Dekker's algorithm shows that removal of a shared variable is a natural abstraction operation.

Our experience in modelling systems with CCS suggests that techniques to abstract process data, such as process parameters, are useful. In Section 5.4 we showed how our first technique can be applied to this problem. Cleaveland and Riely [29] have shown how abstract interpretation can be applied to it. Our attempts to define algebraic laws for abstracting value-passing processes relative to the weakly simulates preorder have not been very successful. We found that some abstraction ideas would result in processes that could not be defined in value-passing CCS.

Our experience with CCS also shows that systems are commonly modelled in a sub-language of CCS. In this language the components of a process are sequential and are composed by restricting on all actions found in more than one component. There may be abstraction operations that are especially useful for these processes.

It would be useful to see how other verification techniques could be formulated as abstraction techniques. The stubborn-set technique in [92] and the partial-order techniques in [45,43] seem natural candidates. In both, commutativity of transitions are used to allow certain transitions of a process to be removed. However, the techniques do not characterize the class of properties for which this abstraction is sound. An abstraction-based account would treat removal of the transitions as an abstraction operation allowed for a particular class of temporal properties.

Most of our algebraic laws for abstraction abstract a single component of a process. Sometimes one would like to abstract two or more components at once. For example, Lynch [69] abstracts the alternating-bit protocol and shows (by exhibiting a bisimulation relation) that the abstract version simulates the concrete version. In the abstract protocol both the sender and receiver component are modified. We would like to find algebraic laws that could be used to obtain the abstract protocol.

Our ultimate goal is to support the greatest possible syntactic abstraction of a process relative to a property to be verified. Our two techniques fall short of this goal in different ways. The first technique is fine-grained in that the abstraction operations it allows depend on the particular property to be proved, but it provides for no direct algebraic manipulation. The second technique supports direct algebraic manipulation, but only for the large, fixed class of properties preserved by the preorders. The task of finding algebraic laws for abstraction relative to a particular class of formulas is difficult because showing a relation to be a precongruence is difficult. Existing work [8,47] shows how to prove that a relation is a congruence for a large class of process languages. For this goal we would like instead a result showing that a large class of preorders are precongruences for a single process language.

Bibliography

[1] Samson Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53:225–241, 1987.

[2] Henrik Reif Andersen and Glynn Winskel. Compositional checking of satisfaction. *Formal Methods in System Design*, 1(4), December 1992.

[3] J.C.M. Baeten and J.A. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3:142–188, 1991.

[4] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Syntax and defining equations for an interrupt mechanism in process algebra. Technical Report CS-R8503, CWI, Amsterdam, 1985.

[5] Mordechai Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on Programming Languages and Systems*, 3(6):333–344, 1984.

[6] S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *Proceedings of CAV '92, LNCS 663*, pages 260–273, 1992.

[7] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60:109–137, 1984.

[8] Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. *Journal of the ACM*, 42(1):232–268, 1995.

[9] T. Bolognesi and E. Brinksma. Introduction to the specification language LOTOS. In van Eijk, Vissars, and Diaz, editors, *The Formal Description Technique LOTOS*. Elsevier, 1989.

[10] A. Bouajjani, J.C. Fernandez, and N. Halbwachs. Minimal model generation. In *Proceedings of CAV '90, LNCS 531*, pages 197–203, 1990.

[11] Gérard G. Boudol. Notes on algebraic calculi of processes. Technical Report 395, INRIA – Sophia Antipolis, 06560 Valbonne, France, April 1985.

[12] J. C. Bradfield. *Verifying Temporal Properties of Systems*. Birkhäuser, Boston, Mass., 1991.

[13] M.C. Browne, E.M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 81:13–31, 1989.

[14] Glenn Bruns. A practical technique for process abstraction. In *Proceedings of CONCUR '93, LNCS 715*, pages 37–49, 1993.

[15] Glenn Bruns. *Distributed Systems Analysis with CCS*. Prentice Hall, 1997.

[16] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *Computing Surveys*, 24(3):293–318, September 1992.

[17] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[18] Ivan Christoff. Testing equivalences and fully abstract models for probabilistic processes. In *Proceedings of CONCUR '90, LNCS 458*, 1990.

[19] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages*, pages 343–354, 1992.

[20] 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*, 8(2):244–263, 1986.

[21] E.M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Proceedings of CAV '93, LNCS 697*, pages 450–461, 1993.

[22] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 353–362. IEEE Computer Society Press, 1990.

[23] R. Cleaveland. An operational semantics of value passing. In B. Bloom, editor, *Proceedings* 2^{nd} *North American Process Algebra Workshop, Ithaca, New York*, 1993.

[24] Rance Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27:725–747, 1990.

[25] Rance Cleaveland. The NCSU concurrency workbench. In *Proceedings of CAV '96, LNCS 1102*, pages 394–397, 1996.

[26] Rance Cleaveland and Matthew Hennessy. Priorities in process algebra. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1988.

[27] Rance Cleaveland, Gerard Lüttgen, V. Natarajan, and Steve Sims. Modeling and verifying distributed systems using priorities: A case study. *Software Concepts and Tools*, 15:50–62, 1996.

[28] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, 1993.

[29] Rance Cleaveland and James Riely. Testing-based abstractions for concurrent systems. In *Proceedings of CONCUR '94, LNCS 836*, pages 417–432, 1994.

[30] Rance Cleaveland and Bernhard Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2(2):121–147, April 1993.

[31] W. de Roever. The quest for compositionality – a survey of assertion-based proof systems for concurrent programs. part 1: Concurrency based on shared variables. In E. Neuhold, editor, *The Role of Abstract Models in Information Processing*. North Holland, 1985.

[32] Edsger Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[33] Edsger W. Dijkstra, Leslie Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, 1978.

[34] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[35] E. A. Emerson and J. Y. Halpern. 'Sometimes' and 'Not Never' revisited: on branching time versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[36] E. Allen Emerson and Chin-Luang Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the 1st Annual Symposium on Logic in Computer Science*, pages 267–278. IEEE Computer Society Press, 1986.

[37] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. In *Proceedings of CAV '93, LNCS 697*, pages 463–478, 1993.

[38] Javier Esparza and Glenn Bruns. Trapping mutual exclusion in the box calculus. *Theoretical Computer Science*, 153:95–128, 1996.

[39] J.C. Fernandez and L. Mounier. Verifying bisimulation on the fly. In *Third International Conference on Formal Description Techniques, FORTE '90*, 1990.

[40] Jean-Claude Fernandez and Laurent Mounier. "On the fly" verification of behavioural equivalences and preorders. In *Proceedings of CAV '91, LNCS 575*, 1991.

[41] M. Fischer and R. Ladner. Propositional dynamic logic of regular programs. *Journal of Computing and System Sciences*, 18:194–211, 1979.

[42] Mahesh Girkar and Robert Moll. New results on the analysis of concurrent systems with an indefinite number of processes. In *Proceedings of CONCUR '94, LNCS 836*, pages 65–80, 1994.

[43] Patrice Godefroid. Using partial orders to improve automatic verification methods. In *Proceedings of CAV '90, LNCS 531*, pages 176–185, 1990.

[44] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032 of *LNCS*. Springer-Verlag, 1996.

[45] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of CAV '91, LNCS 575*, pages 332–342, 1991.

[46] Robert Goldblatt. *Logics of Time and Computation.* Number 7 in CSLI Lecture Notes. Center for the Study of Language and Information, 1992.

[47] J. F. Groote and F. W. Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, October 1992.

[48] Jan Friso Groote. Transition system specifications with negative premises. In *Proceedings of CONCUR '90, LNCS 458*, pages 332–341, 1990.

[49] Jan Friso Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118:263–299, 1993.

[50] David Harel. *First-Order Dynamic Logic.* Springer-Verlag, 1979. LNCS 68.

[51] M. Hennessy and A. Ingólfsdóttir. A theory of communicating processes with value-passing. In *Proceedings of ICALP '90, LNCS 443*, pages 209–219.

[52] M.C. Hennessy and A.J.R.G. Milner. Algebraic laws for non-determinism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[53] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[54] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, 1985.

[55] INRIA. Hyman's algorithm example. Distributed with the AUTO tool.

[56] Claude Jard and Thierry Jeron. On-line model-checking for finite linear temporal logic specifications. In *Proceedings of the 1st Intl. Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407*, 1989.

[57] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86:43–68, 1990.

[58] D.E. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5), 1966.

[59] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[60] R.P. Kurshan and K. McMillan. A structural induction theorem for processes. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, pages 239–247, 1989.

[61] R.P. Kurshan, M. Merritt, A. Orda, and S.R. Sachs. A structural linearization principle for processes. In *Proceedings of CAV '93, LNCS 697*, pages 491–504, 1993.

[62] Y. S. Kwong. On reduction of asynchronous systems. *Theoretical Computer Science*, 5:25–50, 1977.

[63] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 344–352, 1989.

[64] K. G. Larsen and B. Thomsen. A modal process logic. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science,* Edinburgh, pages 203–210. IEEE Computer Society Press, 1988.

[65] Kim G. Larsen and Liu Xinxin. Compositionality through an operational semantics of contexts. *Journal of Logic and Computation*, 1(6):761–795, 1991.

[66] Kim Gulstrand Larsen. Modal specifications. In *Proceedings of the 1st Intl. Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407*, 1989.

[67] David E. Long, Anca Browne, Edmund M. Clarke, Somesh Jha, and Wilfredo R. Marrero. An improved algorithm for the evaluation of fixpoint expressions. In *Proceedings of CAV '94, LNCS 818*, pages 338–350, 1994.

[68] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.

[69] Nancy A. Lynch. Multivalued possibilities mapping. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, pages 519–543, 1989. LNCS 430.

[70] Anthony McIsaac. A formalization of abstraction in LAMBDA. In *Proceedings of the HOL User's Group Workshop*, 1993.

[71] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100(1):1–77, 1992.

[72] Robin Milner. *Communication and Concurrency.* Prentice Hall International, 1989.

[73] F. Moller and C. Tofts. A temporal calculus of communicating systems. In *Proceedings of CONCUR '90, LNCS 458*, pages 401–415, 1990.

[74] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[75] J.L. Peterson and A. Silberschatz. *Operating System Concepts.* Addison Wesley, 1985.

[76] C. Pixley. An incremental garbage collection algorithm for multi-mutator systems. *Distributed Computing*, 3:41–50, 1988.

[77] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Department, Åarhus University, Denmark, 1981.

[78] V. R. Pratt. A decidable mu-calculus. In 22^{th} *Annual Symposium on Foundations of Computer Science,* Nashville, Tennessee, pages 421–427, New York, 1981. IEEE.

[79] Vaughan Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings of the 17th IEEE Symposium on Foundations of Computer Science*, pages 109–121, 1976.

[80] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[81] David M. Russinoff. A mechanically verified incremental garbage collector. *Formal Aspects of Computing*, 6:359–390, 1994.

[82] D. Sangiorgi. The lazy lambda calculus in a concurrency scenario (extended abstract). In *Proceedings of the 7th Annual Symposium on Logic in Computer Science*, pages 102–109. IEEE Computer Society Press, 1992.

[83] Joseph Sifakis. Property preserving homomorphisms of transition systems. In *Logics of Programs*, pages 458–473, 1983. LNCS 164.

[84] A.P. Sistla and S.M. German. Reasoning with many processes. In *Proceedings of the 2nd Annual Symposium on Logic in Computer Science*, pages 138–152. IEEE Computer Society Press, 1987.

[85] Colin Stirling. Modal logics for communicating systems. *Theoretical Computer Science*, 49:331–347, 1987.

[86] Colin Stirling. An introduction to modal and temporal logics for CCS. In A. Yonezawa and T. Ito, editors, *Concurrency: Theory, Language, and Architecture*, pages 2–20, 1989. LNCS 491.

[87] Colin Stirling. Modal and temporal logics. In *Handbook of Logic in Computer Science*, pages 477–563. Oxford University Press, 1991.

[88] Colin Stirling. Modal and temporal logics for processes. Notes for Summer School in Logic Methods in Concurrency, Computer Science Department, Åarhus University, Denmark, 1993.

[89] Colin Stirling and David Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, 1991.

[90] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. of Maths*, 5:285–309, 1955.

[91] C. Tofts. A synchronous calculus of relative frequency. In *Proceedings of CONCUR '90, LNCS 458*, pages 467–480, 1990.

[92] Antti Valmari and Matthew Clegg. Reduced labelled transition systems save verification effort. In *Proceedings of CONCUR '91, LNCS 527*, pages 526–540, 1991.

[93] J. van Bentham. Correspondence theory. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic Vol. II.* Reidel, 1984.

[94] R.J. van Glabbeek. The linear time – branching time spectrum; the semantics of sequential systems with silent moves. In *Proceedings of CONCUR '93, LNCS 715*, pages 66–81, 1993.

[95] D. Walker. Automated analysis of mutual exclusion algorithms using CCS. *Formal Aspects of Computing*, 1:273–292, 1989.

[96] Glynn Winskel. A note on model checking the modal ν-calculus. In *Proceedings of ICALP '89, LNCS 372*, pages 761–771, 1989.

[97] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages*, 1986.

[98] Pierre Wolper and Vinciane Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the 1st Intl. Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407*, 1989.